Conditional Preference Networks: Efficient Dominance Testing and Learning

Kathryn Laing

School of Mathematics

University of Leeds

Submitted in accordance with the requirements for the degree of

Doctor of Philosophy

June 2020

The candidate confirms that the work submitted is her own, except where work which has formed part of a jointly authored publication has been included. The contribution of the candidate and the other authors to this work has been explicitly indicated below. The candidate confirms that appropriate credit has been given within the thesis where reference has been made to the work of others.

Chapter 2 of this thesis (as well as the associated appendices and proofs) is based upon material that has appeared in publication as follows:

> Laing, K., Thwaites, P. A., and Gosling, J. P. (2019). Rank pruning for dominance queries in CP-nets. *Journal of Artificial Intelligence Research,* 64:55–107.

All explanatory text, results, proofs, and experiments contained within this paper were contributed by the candidate, Kathryn Laing. Some of the initial concepts for this paper were conceived in collaboration with the other authors.

Acknowledgements

I would first like to thank both of my supervisors, Peter and JP, for all of their help and support. Thank you for your patience, encouragement, and guidance over the years. Without you, I would never have embarked upon a PhD and I could not have asked for better supervisors to help guide me through it. I would also like to take this opportunity to thank the many members of staff in the School of Maths who have made both my undergraduate and postgraduate time at Leeds the most amazing experience.

Thank you to my family, who support me in everything I do. In particular, thank you to my mum, who always believes in me, even when I cannot see it for myself. To my nanna, I give all the credit for this achievement, as previously agreed.

Finally, thank you to Kasia, who saw this PhD through alongside me. I cannot possibly list everything you have done for me over the last four years, so I will simply say thank you for everything.

All experiments in this thesis were undertaken on ARC3, part of the High Performance Computing facilities at the University of Leeds, UK.

Abstract

Modelling and reasoning about preference is necessary for applications such as recommendation and decision support systems. Such systems are becoming increasingly prevalent in all aspects of our daily lives as technology advances. Thus, preference representation is a wide area of interest within the Artificial Intelligence community. Conditional preference networks, or CP-nets, are one of the most popular models for representing a person's preference structure. In this thesis, we address two issues with this model that make it difficult to utilise in practice. First, answering dominance queries efficiently. Dominance queries ask for the relative preference between a given pair of outcomes. Such queries are natural and essential for effectively reasoning about a person's preferences. However, they are complex to answer given a CP-net representation of preference. Second, learning a person's CP-net from observational data. In order to utilise a CP-net representation of a person's preferences, we must first determine the correct model. As direct elicitation is not always possible or practical, we must be able to learn CP-nets passively from the data we can observe.

We provide two distinct methods of improving dominance testing efficiency for CP-nets. The first utilises a quantitative representation of preference in order to prune the associated search tree. The second reduces the size of a dominance testing problem by preprocessing the CP-net. Both methods are shown experimentally to significantly improve dominance testing efficiency. Furthermore, both are shown to outperform existing methods. These techniques can be combined with one another, and with the existing methods, in order to further improve efficiency.

We also introduce a new, score-based learning technique for CP-nets. Most existing work on CP-net learning uses pairwise outcome preferences as data. However, such preferences are often impossible to observe passively from user actions, particularly in online settings, where users typically choose from a variety of options. Contrastingly, our method assumes a history of user choices as data, which is observable in a wide variety of contexts. Experimental evaluation of this method finds that the learned CP-nets show high levels of agreement with the true preference structures and with previously unseen (future) data.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation and Contributions

As we increasingly rely upon technology in all aspects of our lives, we have come to expect that it will anticipate our needs and be a personalised experience that is most helpful to us. This can be seen in content recommendation on platforms such as Netflix or Spotify, personalised news feeds on our phones, and online shopping suggestions. Even our autofill prompts when typing are tailored to match personal writing styles and phrases. We are also developing progressively advanced technology to assist or automate human decision making. Smart home technology and route planners are already commonplace, and driverless cars are becoming an everyday reality. We are even in the process of creating systems to assist or automate medical diagnoses. In order to provide helpful recommendations or decision support, these systems must understand the preferences of the intended user. Hence, modelling and reasoning with human preferences is an important topic in artificial intelligence.

There have been many methods proposed in the literature for modelling preference. The most prominent of the existing models are as follows. Conditional preference networks (Boutilier et al., 2004a), or CP-nets, are a graphical representation of qualitative preferences. There have also been many extensions and generalisations of CP-nets proposed, which we shall discuss below. Conditional importance networks (Bouveret et al., 2009), or CI-nets, are another graphical model, which represent qualitative preferences over sets of outcomes. We can also represent qualitative preferences via lexicographic preference trees (Booth et al., 2010), where the edges represent choices over features, or a more generalised model, preference trees (Liu and Truszczynski, 2014), where edges represent the satisfaction of more complex conditions over the features (represented by propositional formulae). Qualitative preferences over a set of possibilities can also be expressed via propositional logic languages, using formulae to dictate desired properties (Coste-Marquis et al., 2004). Another graphical representation is π-pref nets (Amor et al., 2015), which model preference via a possibilistic network and can represent both qualitative and quantitative preference information. Quantitative preferences can be represented as soft constraints in a constraint satisfaction problem (Bistarelli et al., 1997; Schiex et al., 1995). Ordinal conditional function networks (Eichhorn et al., 2016) are another graphical structure that we can use to represent qualitative preference information. These structures are similar to Bayesian networks, however, they express degrees of implausibility rather than probability. Naturally, one can also represent quantitative preference via a multi-attribute utility function over the set of alternatives. There are also several graphical representation of such utilities: CAI-nets (Bacchus and Grove, 1995), GAI-nets (Gonzales and Perny, 2005), utility diagrams (Abbas and Howard, 2005), CUI networks (Engel and Wellman, 2008), utility difference networks (Brafman and Engel, 2009), and bidirectional utility diagrams (Abbas, 2010). These are more compact representations, which are easier to elicit and reason with, though they rely on assumptions regarding independence and the decomposability of the utility function.

In this thesis, our focus will be on CP-net models of preference. From here on, we shall refer to the person whose preferences we are modelling as the *user*, as they are the intended user of the system. CP-nets are a compact graphical model of user preferences over a large combinatorial domain of possible outcomes. They are based upon natural, qualitative *ceteris paribus* preference statements, which make them simple to elicit from non-expert users. A *ceteris paribus* preference is a preference made under the assumption of ‘all else being equal’. This is the typical assumption people make when specifying their preferences; for example, if a person specified that they ‘would rather have a leather sofa than a fabric sofa’, then they are implicitly assuming that everything else about these two hypothetical sofas are the same. Such a statement does not imply that a small, second-hand leather sofa would be preferable to a large, brand new fabric sofa.

Using qualitative preference statements rather than quantitative also improves elicitation and applicability of CP-nets. Firstly, relative preferences are simpler and, thus, more likely to be accurately specified by the user. Secondly, while quantitative preference information can always be simplified into qualitative preferences, the reverse is not always possible – there are scenarios where there is no numerical value associated with the possible choices and the user cannot accurately quantify their preference.

CP-nets can also express conditional (*ceteris paribus*) preference statements, such as 'unless it is a recliner, I would rather have a leather sofa than fabric'. This allows CP-nets to represent more complex preference structures, despite being based on natural preference statements.

Preference optimisation is also simple given a CP-net model of user preference. The globally optimal (most preferred) outcome can be found in linear time, as can the optimal outcome under certain types of constraint. This is important as it means that we can quickly identify the best choice for the user.

As well as making CP-nets simple to elicit, their compact, qualitative nature and simple interpretations allow CP-net models (and any subsequent reasoning) to be clearly explained to non-experts. As artificial intelligence systems are being applied to tasks of increasing responsibility, such as medical diagnoses and fraud detection, the explainability of these systems is becoming more important. CP-nets offer simple interpretations and transparency in their reasoning, which is not always possible for other techniques used for recommendation and decision support such as neural networks. For CP-nets, it is simple to explain to the user (and correct if necessary) the assumptions and methods used in our reasoning.

These many desirable properties have made CP-nets one of the most popular models for preference representation in the literature. CP-nets were first introduced in Boutilier et al. (1999), followed by a more comprehensive introduction in Boutilier et al. (2004a). Since then, many extensions and generalisations of the CP-net model have been proposed, most notably:

- **UCP-Nets:** UCP-nets (Boutilier et al., 2001) add quantitative preference information to CP-nets. Instead of local (conditional) preference rules, UCP-nets have local (conditional) utility functions.

- **mCP-Nets:** mCP-nets (Rossi et al., 2004) represent the aggregation of the preferences of multiple users (who all have their own CP-net representations, which may be dependent upon one another).

- **CP-theories:** Conditional preference theories (Wilson, 2004b) are a generalisation of CP-nets. Unlike CP-nets, CP-theories do not have a graphical representation, rather, they are theories of a defined conditional preference logic. These theories consist of stronger conditional preference statements (in the associated formal preference language) that do not require the CP-net *ceteris paribus* assumption. CP-theories are strictly more expressive than

CP-nets – all preference structures representable by a CP-net can be represented by a CP-theory, but not vice versa. Wilson (2004a) shows that all TCP-nets can also be expressed via CP-theories.

- **TCP-Nets:** Tradeoff-enhanced CP-nets (Brafman et al., 2006) extend CP-nets to express (conditional) relative importance statements as well as relative preference.
- **LCP-Nets:** Linguistic CP-nets (Châtel et al., 2008) extend TCP-nets by assigning a linguistic term to each possible feature choice, giving qualitative local utilities. That is, where (T)CP-nets give local preference orderings over the possibilities, LCP-nets give each value a linguistic expression of preference such as "very high" or "low". This provides more information about the relative preferences whilst keeping the preferences qualitative.
- **WCP-Nets:** In weighted CP-nets (Wang et al., 2012), the dependency structure and local preferences of CP-nets are annotated with weights. These weights reflect the degree of relative variable importance and relative preference between values, respectively. Each weight is obtained by assigning one of five qualitative degrees of importance and then translating this into a value between 1 and 5.
- **PCP-Nets:** Probabilistic CP-nets (Bigot et al., 2013; Cornelio et al., 2013) enable CP-nets to encode uncertainty over user preferences – essentially, PCP-nets represent a probability distribution over a class of CP-nets. They give a probability distribution over all possibilities for each local preference rule. In Cornelio et al. (2013), they present a more generalised framework where one can also encode uncertainty over the preferential dependency structure.

In this thesis, however, we address some outstanding problems with the original CP-net model.

Many applications of CP-nets have also been suggested and explored experimentally in the literature. Wicker (2006) suggests using CP-nets to model user profiles on social media platforms, in order to evaluate a degree of interest matching between users (for the purpose of suggesting potential 'friends' or content). Bistarelli et al. (2007) propose a method of using CP-nets to assist in the selection of appropriate countermeasures, in order to mitigate the risk of potential cyber-attacks. Boubekeur et al. (2007) introduce an information retrieval system based on CP-net models. The idea is to identify the most relevant documents from a user's natural language query. Li et al. (2011b) consider the problem of social choice when voters preferences are represented via CP-nets. Both Alanazi et al. (2012) and Mohammed et al. (2015) propose using CP-nets to represent qualitative preferences in a personalised online shopping procedure. This procedure can also be applied to content recommendation on platforms like Netflix and YouTube. Aydŏgan et al. (2013) utilises CP-nets as preference representations for automated negotiations. Cafaro et al. (2013) introduces a method for utilising CP-nets in grid scheduling – allocating computational resources to submitted job requests. Wang et al. (2016) and Wang et al. (2019) both propose recommender systems for web service selection based upon CP-net representations of user preferences. Both evaluate their systems on real data and Wang et al. (2016) uses human subjects to evaluate the recommendation quality. Khoshkangini et al. (2018) propose learning CP-nets of a particular form in order to use them as recommender systems. They test this procedure on two real-world data sets. Haqqani et al. (2018) learn CP-net models over various aspects of transportation from real data, in order to offer personalised journey planning recommendations. They find that the CP-net models closely match the true user preference orderings and perform better than other techniques for preference learning, which do not allow for conditional preferences. These examples are only a sample of the wide variety of proposed applications for CP-nets. However, despite their popularity and many proposed uses, CP-nets have not yet been applied in practice, as far as we are aware.

While CP-nets have many desirable properties and have been popular in academic research, there remain unresolved issues with these models, which may explain their lack of adoption in practical applications.

Firstly, accurate preference comparisons are complex when using CP-net models. CP-nets make certain preference reasoning tasks simple; globally optimal outcomes can be found in linear time, as can optimal outcomes under certain plausibility constraints (Boutilier et al., 2004a). Weak preferential comparison is also possible in linear time (Boutilier et al., 2004a). That is, given two possible outcomes, a and b, we can determine an ordering that is not contradicted by the CP-net. Such pairwise comparisons also enable us to obtain non-contradictory orderings of outcome sets. However, a weak preference $a \succ b$ means only that the CP-net model does not include the information 'b is preferred to a'. As CP-nets do not typically encode a preference between every outcome pair, weak preferences may still be incorrect. In order to assess the relative preference of outcome pairs accurately, we must use dominance queries. These queries tell us precisely what is known (encoded by the CP-net) about the user's relative preference between a and b.

1. Introduction

If CP-nets are to be used for decision support, then it is essential that we are able to extract the relative preference of the possible options. Boutilier et al. (2004b) have shown that evaluating optimal outcomes under arbitrary constraints requires dominance queries. However, answering dominance queries is a complex task given a CP-net representation of user preference. For general CP-nets, answering dominance queries has been shown to be PSPACE-complete (Goldsmith et al., 2008) (see Appendix F for definition). Thus, answering these queries efficiently, particularly for larger CP-nets, is a difficult task. In §2.2.3, we review the existing work on this problem.

Secondly, there is insufficient work on determining a user's CP-net. In order to use CP-nets to represent (and reason with) user preferences in practical applications, one first needs to know what the user's CP-net looks like. Whilst CP-nets are simple to elicit directly even from non-experts (Boutilier et al., 2004a), it is not always possible or practical to obtain a user's CP-net model via direct queries. Consider existing recommendation systems for services such as Netflix or Amazon. These platforms, among many others, approximate user preference without requiring any direct user input about their preferences. A system that requires users to first specify their preference structure may be off-putting by comparison. Furthermore, users may be unwilling to reveal their preferences in certain contexts such as auctions or games played against an adversary. Another disadvantage to user specified models is that they may be inconsistent or inaccurate due to human error (particularly for larger systems) or if their abstract preferences do not match up with their actions when using the system. Preferences are also liable to change over time, meaning that a user specified preference structure will not remain accurate.

Hence, for CP-nets to be a plausible model for practical applications, we must be able to learn a user's CP-net from observable data, rather than via direct elicitation. Most existing work on learning CP-nets (reviewed in §4.2) assumes pairwise preferences as training data. However, in many contexts (including most online scenarios), users are not presented with choices between pairs of options. Rather, they typically make a choice out of a number of options, which can vary between tens and thousands (consider selecting out of recommended videos on YouTube vs selecting a movie on Netflix). Thus, we observe only which outcome was successful (the item the user chose to watch or buy). Despite the fact that pairwise preferences are unrealistic to passively observe in many contexts, learning from other types of data has received little attention thus far.

In this thesis, we address both of these problems. We propose two distinct approaches to improving dominance testing efficiency. The first introduces a quantitative representation of preference, which can be used to improve the efficiency of answering dominance queries by pruning the associated search. The second method reduces the size of the dominance testing problem by preprocessing the CP-net (and query) prior to answering. Both methods are shown to improve dominance testing efficiency significantly and more effectively than existing methods. These methods can also be combined (with one another and existing methods) for further efficiency. Our third contribution is a novel technique for learning CP-nets from user choice data – that is, a history of which outcomes were successful (chosen by the user). Experimental analysis finds that this algorithm learns CP-nets that agree strongly with both the user's true preference structure and test sets of previously unseen (future) data.

1.2 Thesis Overview

In the remainder of Chapter 1, we provide the necessary background on CP-nets and a glossary of common notation.

In Chapters 2 and 3, we address the problem of efficient dominance testing. In Chapter 2, we introduce a quantitative measure of user preference for a given CP-net, called outcome ranks. These rank values can be used both to obtain a preference ordering over a set of possible outcomes and to prune dominance query search trees in order to improve efficiency. We experimentally evaluate the performance of rank pruning in comparison to the existing pruning methods, also considering all possible combinations of these methods. We then go on to generalise our results to the case of CP-nets that allow indifference.

In Chapter 3, we improve dominance testing efficiency from a different perspective; rather than making our dominance testing methods faster, we reduce the size of the dominance query problem. We introduce a novel method of preprocessing the CP-net, given a specific dominance query, in order to simplify the problem and make dominance testing more efficient. We evaluate experimentally the effect of our preprocessing on dominance testing complexity in the binary case. Further, we compare these results to both the existing preprocessing technique and the combination of the two. In these experiments, we answer the (original and preprocessed) queries using an efficient pruning schema from Chapter 2, showing that preprocessing can further improve upon the efficiency we obtained in Chapter 2 via pruning.

In Chapter 4, we introduce a novel method of learning binary CP-nets from user choice data. In order to learn a user's CP-net, we first construct a score that measures the agreement between the observed data and a given CP-net candidate model. We then attempt to maximise this score over the space of candidate models. We evaluate the performance of our learning algorithm experimentally, using simulated choice data.

The relevant existing literature on these topics is reviewed at the beginning of Chapters 2 and 4, respectively (§2.2 and §4.2). The relevant literature for Chapter 3 is covered by the Chapter 2 review. In the final sections of Chapters 2, 3, and 4, we provide a discussion of our results from that chapter and related future work.

Finally, in Chapter 5, we provide a summary of our contributions and conclusions.

All proofs can be found in the appendices, except those which can be given concisely and without significantly disrupting the text. We have also located all non-essential details about our algorithms (necessary for implementation but not understanding) in the appendices. Proofs of algorithm completeness as well as secondary algorithms (minor algorithms which are called in turn by our primary algorithm of interest) can also be found in the appendices. Similarly, additional details and further results from our experiments are given in the appendices. These additional details are of interest but are not necessary to understand the outcomes of our experiments with respect to our original question of interest. In the case of Appendix A, we have consigned a novel theoretical result to the appendices as it is tangential to our other work and too extensive to keep in the main text without disruption. Appendix F contains a glossary of additional, non-essential terminology used in the thesis. These are typically terms used in describing the work of others, which we do not define in the main text. More specific locations of these additional details are provided where relevant throughout the main text of the thesis.

In order to evaluate the novel methods we introduce in this thesis, we conducted several experimental evaluations. This required us to write code from scratch both for the implementation of our methods and for the encoding and general handling of CP-nets. In our previous publication, Laing et al. (2019), all code was written in R. In this thesis, all experiments were conducted in C++ (in some cases via the R package Rcpp). Throughout this thesis, we provide all essential details for understanding our methods and experiments, including pseudocode algorithms and some discussion of CP-net encoding. However, for the purposes of implementation, all of our code (in both languages) has been made available at `github.com/KathrynLaing`. Further practical details of our CP-net encodings, experiments, and our results can also be found here. One of the main practical issues with handling CP-nets computationally is their exponential size and the constraints of limited accuracy and storage enforced by computers. Some discussion of this is given here, particularly in Chapter 4, but the full details of our practical solutions to these issues are given in the online repository.

1.3 CP-Net Preliminaries

In this section, we give the necessary background on CP-nets, as defined in Boutilier et al. (2004a).

Definition 1.1. A *conditional preference network (CP-net)*, N, is a graphical representation of user preference over a finite set of discrete variables, V. We assume the domain of each variable, $X \in V$, denoted $\mathrm{Dom}(X)$, to be finite. The CP-net, N, consists of two parts. First, a directed graph, G, with nodes V. We say G is the *structure* of N. Second, each node, $X \in V$, is annotated with a *conditional preference table (CPT)*, denoted $\mathrm{CPT}(X)$. Let $\mathrm{Pa}(X)$ denote the *parent set* of X in G. Then, for each possible assignment of values to $\mathrm{Pa}(X)$, $\mathrm{CPT}(X)$ gives the user's *ceteris paribus* preference order over all possible values of X, $\mathrm{Dom}(X)$. (Boutilier et al., 2004a)

The *ceteris paribus* preference $a \succ b$ means that, all else being equal, the user prefers a to b. In $\mathrm{CPT}(X)$, the preference orders over $\mathrm{Dom}(X)$ hold under the assumption that all other variables, $V \backslash X$, remain fixed.

The structure, G, is a *preferential dependency graph*. The edge $X \to Y$ means that the user's preference over Y is dependent upon the value taken by X. CP-net semantics assume that, for every pair of variables, $X, Y \in V$, X is *preferentially independent* of Y given $\mathrm{Pa}(X)$. That is, once $\mathrm{Pa}(X)$ is assigned a set of values, the preference order over $\mathrm{Dom}(X)$ is fully determined and does not depend upon the value taken by Y.

We illustrate these concepts using the following example from our paper, Laing et al. (2019).

Example 1.2. Suppose that we are modelling a user's preferences over aeroplane seats. The variables we might take into account, and their respective domains, are as follows.

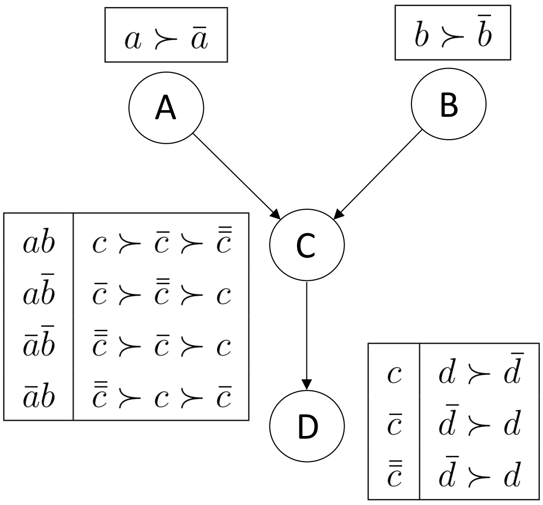

$a \succ \bar{a}$
$b \succ \bar{b}$
A
B
C
D
ab $c \succ \bar{c} \succ \bar{\bar{c}}$
$a\bar{b}$ $\bar{c} \succ \bar{\bar{c}} \succ c$
$\bar{a}\bar{b}$ $\bar{\bar{c}} \succ \bar{c} \succ c$
$\bar{a}b$ $\bar{\bar{c}} \succ c \succ \bar{c}$
c $d \succ \bar{d}$
$\bar{c}$ $\bar{d} \succ d$
$\bar{\bar{c}}$ $\bar{d} \succ d$

Figure 1.1: Example CP-Net

$A =$ Flight Length	$\text{Dom}(A) = \{a : \text{short}, \bar{a} : \text{long-haul}\}$
$B =$ School Term Time	$\text{Dom}(B) = \{b : \text{term}, \bar{b} : \text{holiday}\}$
$C =$ Class	$\text{Dom}(C) = \{c : \text{economy}, \bar{c} : \text{business}, \bar{\bar{c}} : \text{first}\}$
$D =$ Pay Extra for Wi-Fi	$\text{Dom}(D) = \{d : \text{no}, \bar{d} : \text{yes}\}$

One example of a possible CP-net over these variables is given in Figure 1.1. The CPTs of this CP-net show that the user has a strict preference for short flights over long-haul flights (*ceteris paribus*, that is, given B, C, D take the same values) and for flying in term time over flying in holiday time (*ceteris paribus*, that is, given A, C, D take the same values). These preferences are unaffected by the values taken by any other variable. However, the user's preference over which class they fly in is dependent (conditional) upon the values taken by A and B (Flight Length and School Term Time). If it is a short flight in term time, then the user prefers economy to business to first class (*ceteris paribus* – given that D takes the same value). However, if it is a short flight in holiday time, then the user prefers business to first to economy class. Once the values of A and B are determined, these preferences over C (Class) are fixed and do not change (regardless of the value taken by D), by our preferential independence assumption above. Similarly, the user's preference over D (Pay Extra for Wi-Fi) depends upon the value taken by C, but these preferences are independent of the values taken by A and B.

CP-nets may contain irrelevant edges that do not contribute additional preference information. Such edges occur when a variable has a parent that it is not preferentially dependent upon. Suppose a CP-net contains the edge $X \to Y$. We say this edge is *degenerate*, or that X is a degenerate parent of Y, if changing the value of X does not affect the user's preference over Y. More formally, for any two assignments of values to $\mathrm{Pa}(Y)$, say $\mathbf{u}_1$ and $\mathbf{u}_2$, that differ only on the value of X, the preferences over $\mathrm{Dom}(Y)$ given in $\mathrm{CPT}(Y)$ must be the same under both $\mathrm{Pa}(Y) = \mathbf{u}_1$ and $\mathrm{Pa}(Y) = \mathbf{u}_2$. For example, if every entry in $\mathrm{CPT}(D)$ was $\bar{d} \succ d$ in Example 1.2, then C would be a degenerate parent of D. Such degenerate edges represent a trivial parent-child relation. As this does not add any information to the CP-net preference structure, degenerate edges can be removed without changing the CP-net semantics. The child's CPT can be trivially simplified as preference is not dependent upon the removed parent.

If all variables in a CP-net are binary, then we say that it is a *binary CP-net*. In Chapters 2 and 3, we allow all CP-nets to contain multivalued variables. In Chapter 4, our learning is restricted to binary CP-nets and, thus, all CP-nets considered in this chapter are assumed to be binary.

We assume that all CPT preference orders are strict total orderings of the appropriate domain, except in §2.5, where we consider more general CP-nets with indifference statements. In this case, we assume CPT preference orders to be total preorders (transitive and complete). For example, we could have the non-strict preference ordering $\bar{\bar{c}} \succ c \sim \bar{c}$ for variable C, meaning that the user would rather be sat in first class, but is indifferent between economy and business. Such indifferences are natural preference statements that we may reasonably expect to find in real world preference structures. This generalisation is therefore an important CP-net extension to consider, in order to improve the applicability of our results.

CP-nets consist of (conditional) preference rules over the variable domains. However, we are primarily interested in the user's preferences over the outcomes – the products or scenarios the user is ultimately deciding between.

Definition 1.3. Let N be a CP-net over variables $V = \{X_1, X_2, ..., X_n\}$. An *outcome*, o, is an n-tuple representing an assignment to each variable. The set of outcomes associated with N is

$$\Omega = \mathrm{Dom}(X_1) \times \mathrm{Dom}(X_2) \times \cdots \times \mathrm{Dom}(X_n),$$

where × denotes the Cartesian product. We denote the number of outcomes associated with a CP-net by $\mathcal{O} = |\Omega|$.

For the CP-net given in Example 1.2, an outcome is a fully specified flight such as $\bar{a}b\bar{\bar{c}}\bar{d}$ (a long-haul flight in holiday time, sat in first class with wi-fi). For this example, there are 24 possible outcomes. In general, $\mathcal{O} \geq 2^n$, with equality only in the case of binary CP-nets. The user's preference structure over the outcome set is given explicitly by the associated preference graph.

Definition 1.4. Let N be a CP-net over variables V with associated outcome set Ω. The *preference graph* induced by N, denoted G_N, is a directed graph over Ω. Two outcomes, say $o, o' \in \Omega$, are connected by an edge $o \rightarrow o'$ if and only if the following conditions hold. First, o and o' differ on exactly one variable, say $X \in V$ (such outcome pairs are called a *variable flip*). Let $\mathbf{u}$ be the assignment of values to $\mathrm{Pa}(X)$ in both o and o'. Second, $o'[X]$ (the value assigned to X in o') is preferred to $o[X]$ according to the $\mathrm{CPT}(X)$ preference over $\mathrm{Dom}(X)$ under the assignment $\mathrm{Pa}(X) = \mathbf{u}$ (making $o \rightarrow o'$ an *improving flip*). (Boutilier et al., 2004a)

As the preference orders in CPTs are *ceteris paribus*, they imply preferences between outcomes that differ on exactly one variable. In particular, the preferences encoded by CPT preference rules are exactly the edges of G_N. Thus, the edges of G_N and their transitive closure represent exactly the outcome preferences encoded by N. The preference graph is, thus, an equivalent representation of the CP-net itself. We use the CP-net, rather than dealing directly with preference graphs, as it is more compact and, thus, easier to elicit and reason with.

The preference graph induced by the CP-net in Example 1.2 is given in Figure 1.2. As this graph has 24 nodes, each of degree 5, it is reasonably complex to examine. We have therefore coloured all redundant edges in light blue so that the necessary relationships (given in black) can be seen more easily. That is, any preference given by a blue edge is also represented via a black path in the diagram. In this graph, an edge or path from outcome a to outcome b tells us that the user (whose preferences are represented by the CP-net in Figure 1.1) prefers outcome b to outcome a. These pairwise outcome preferences are exactly the set of preferences encoded by the CP-net in Figure 1.1.

Definition 1.5. Let N be a CP-net with induced preference graph G_N. Let o and o' be two outcomes associated with N. We say that N *entails* the preference 'o is preferred to o'', denoted $N \vDash o \succ o'$, if and only if there is a directed path $o' \rightsquigarrow o$ in G_N.

By the definition of G_N, a directed path $o' \rightsquigarrow o$ consists of a sequence of outcomes, $o' = o_1, o_2, ..., o_m = o$, such that o_i and o_{i+1} differ on exactly one variable and $N \vDash o_{i+1} \succ o_i$ for all i. We call such a sequence an *improving flipping sequence (IFS)* (Boutilier et al., 2004a).

If the CP-net structure is acyclic, then the preference graph must also be acyclic and, thus, the CP-net cannot entail any contradictions such as $N \vDash a \succ b$

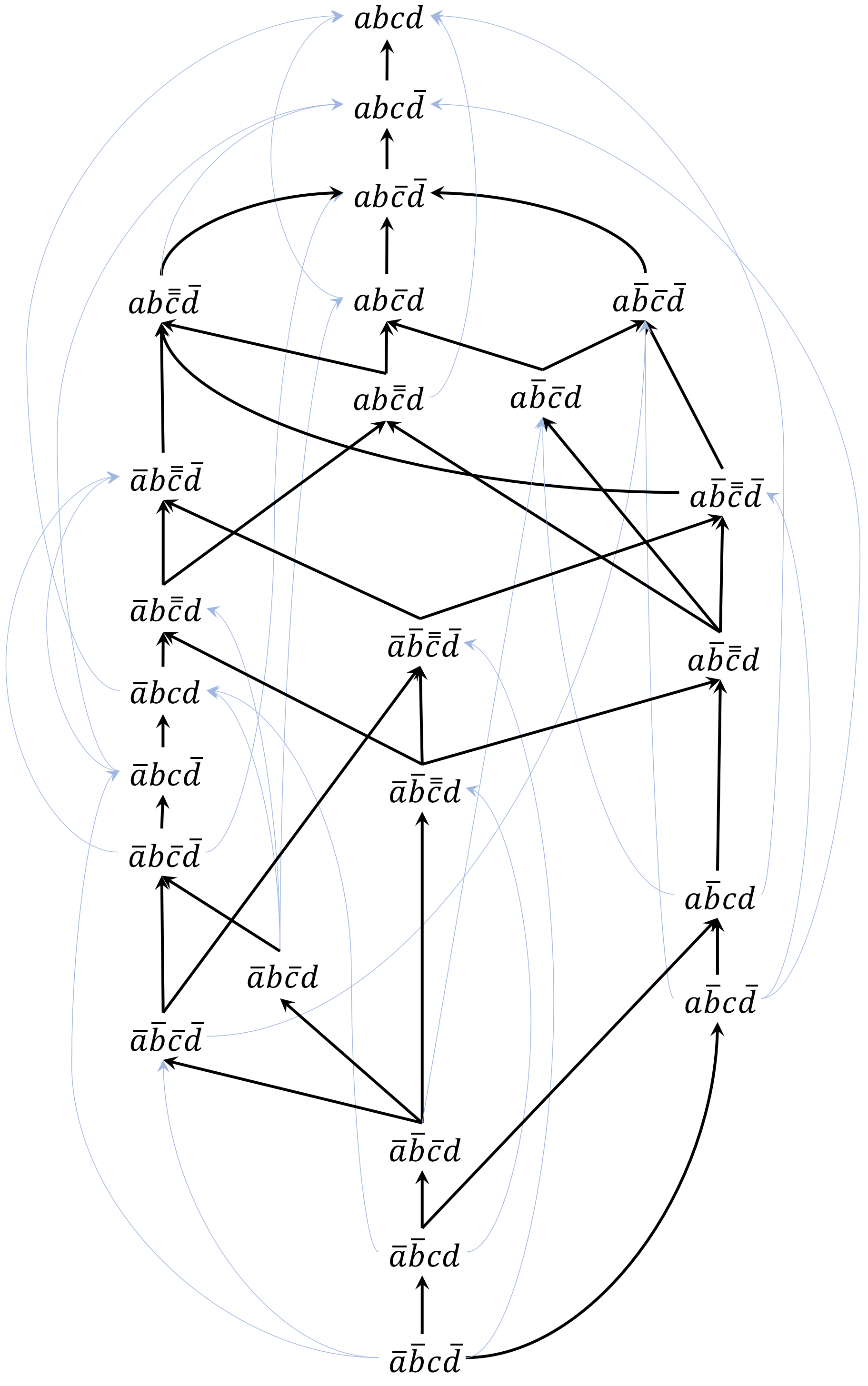


Figure 1.2: Example Preference Graph

and $N \models b \succ a$ (Boutilier et al., 2004a). Throughout this thesis, we assume all CP-nets to be acyclic in order to ensure consistency. Note that cyclic CP-nets can also be consistent, but it is not guaranteed and distinguishing the consistent cases from the inconsistent is a PSPACE-complete problem (Goldsmith et al., 2008).

Given a CP-net, N, a consistent ordering is a complete ordering over the outcomes that obeys all known (entailed) preference information about the user. We allow consistent orderings to contain indifference unless we specify that it is a strict ordering.

Definition 1.6. Let N be a CP-net over outcome set Ω. A *consistent ordering* of N is any total preorder (transitive and complete), $\succsim^C$, over Ω such that, for any $o, o' \in \Omega$, $N \models o \succ o' \implies o \succ^C o'$. Note that $o \succ^C o'$ means that $o \succsim^C o'$ but $o' \not\succsim^C o$.

Due to the connection between G_N and entailed preferences, consistent orderings can also be considered as the set of topological orderings of N_G. Consistent orderings are the set of true user preference orderings that can be accurately represented by N. As we assume CP-nets to be acyclic, there will always be at least one consistent ordering (Boutilier et al., 2004a).

For every pair of outcomes, o and o', we must have exactly one of the following cases: $N \models o \succ o'$, $N \models o' \succ o$, or $N \not\models o \succ o'$ and $N \not\models o' \succ o$. In the final case, we say that the two outcomes are *incomparable* and denote this $N \models o \bowtie o'$. This means that the user's preference between o and o' is unknown. Given a pair of outcomes, there are two types of query to evaluate the relative user preference between them. Ordering queries ask for an ordering of the outcome pair that is consistent with the known user preferences.

Definition 1.7. Let N be a CP-net and let o and o' be two associated outcomes. An *ordering query* requires us to prove at least one of $N \not\models o \succ o'$ or $N \not\models o' \succ o$. (Boutilier et al., 2004a)

If $N \not\models o \succ o'$, then $o' \succ o$ is a consistent ordering of the outcomes, as it does not contradict any known user preferences. However, as the outcomes may be incomparable according to N, consistent outcome orderings are not necessarily true preferences. In fact, they may be in contradiction to the user's true preference. Ordering queries are simple to answer, Boutilier et al. (2004a) give a method of answering these queries in $O(|V|)$ time. On the other hand, dominance queries seek to determine whether there is a genuine preference between the two outcomes.

Definition 1.8. Let N be a CP-net and let o and o' be two associated outcomes. A *dominance query* asks whether $N \models o \succ o'$ holds. (Boutilier et al., 2004a)

Dominance queries are stronger as they determine the entailed preference, not just a consistent ordering of the outcomes. However, they are consequently much more complex to answer. For general CP-nets, dominance queries are PSPACE-complete problems (Goldsmith et al., 2008). The query $N \models o \succ o'$ is true if and only if there is a directed path (IFS) $o' \rightsquigarrow o$ in N_G. Thus, we can consider dominance queries as a search for an IFS, which is how we approach them in Chapters 2 and 3.

Remark. The above definitions of entailment and consistent orderings have been modified from those given by Boutilier et al. (2004a) for simplicity, though they remain equivalent. Boutilier et al. (2004a) define a preference ordering that satisfies a CP-net (that is, a consistent ordering) to be any total preorder that obeys all *ceteris paribus* preference rules in the CPTs. As the preference graph is equivalent to the transitive closure of the CPT preference rules, obeying all CPT preferences is equivalent to being consistent with the preference graph structure. Thus, the two definitions of consistent ordering are equivalent. Boutilier et al. (2004a) then define an entailed relation to be any preference that occurs in all consistent orderings. They prove that this condition is equivalent to our definition of entailment. As both definitions are equivalent, all results from Boutilier et al. (2004a) continue to hold here.

1.4 Notation and Abbreviations

The following table summarises the common notation used throughout this thesis.

N, M	CP-nets
V	Set of all CP-net variables
n	Number of CP-net variables, $n = \|V\|$
X, Y, Z or X_1, X_2, X_3	CP-net variables
$x, \bar{x}$ or x_1, x_2, x_3	Possible values of variable X
$\mathbf{u}, \mathbf{w}$	Tuples of values assigned to a set of variables
$\mathrm{CPT}(X)$	Conditional preference table of variable X
$\mathrm{Dom}(X)$, $X \in V$	Domain of variable X
$\mathrm{Dom}(Y)$, $Y \subseteq V$	Cartesian product of the domains of the variables in Y, $\mathrm{Dom}(Y) = \bigtimes_{X \in Y} \mathrm{Dom}(X)$
n_X	Size of the domain of variable X, $n_X = \|\mathrm{Dom}(X)\|$
$\mathrm{Pa}(X)$	Set of parents of variable X in the CP-net structure
$\mathrm{Ch}(X)$	Set of children of variable X in the CP-net structure
$\mathrm{Anc}(X)$	Set of ancestors of variable X in the CP-net structure
$\mathrm{Dec}(X)$	Set of descendants of variable X in the CP-net structure
d_X	Number of descendent paths of variable X (directed paths originating at X) in the CP-net structure
$\mathbf{u} : x \succ \bar{x}$	The rule in $\mathrm{CPT}(X)$ corresponding to the assignment $\mathrm{Pa}(X) = \mathbf{u}$
G_N	Preference graph induced by CP-net N
Ω or Ω_N	Set of all outcomes associated with CP-net N
$\mathcal{O}$	Number of CP-net outcomes, $\mathcal{O} = \|\Omega\|$
o, o' or o_1, o_2, o_3	CP-net outcomes
$\succsim^C$	Arbitrary consistent ordering
$o \succ o'$	o is strictly preferred to o'
$o \sim o'$	o is equally preferred to o' (the user is indifferent)
$o \bowtie o'$	o and o' are incomparable
$N \vDash o \succ o'$	CP-net N entails the preference $o \succ o'$. Similarly for $o \sim o'$ and $o \bowtie o'$
$G(o')$	Search tree for the dominance query $N \vDash o \succ o'$

$o[X]$, $X \in V$	The value taken by X in outcome o
$o[Y]$, $Y \subseteq V$	The $\lvert Y \rvert$-tuple of values assigned to the variables in Y in outcome o
$e = a \rightarrow b$	A directed edge from a to b
$a \rightsquigarrow b$	A directed path from a to b
$T(N)$	Event tree representation of CP-net N
$W(N)$ or W	Weighted event tree representation of CP-net N
AF_X	Ancestral factor of variable X
$P_P(X = x \mid \text{Pa}(X) = \mathbf{u})$	The preference position of the choice $X = x$, given the assignment $\text{Pa}(X) = \mathbf{u}$
$r(o)$	Rank of outcome o
$r_G(o)$	Generalised rank of outcome o
$\succsim^R$	Rank induced outcome ordering, $\succ^R$ for the strict ordering
$\succsim^G$	Generalised rank induced outcome ordering
$L(X)$	Least rank improvement of variable X
$L_D(o_1, o_2)$	Least entailed rank difference between outcomes o_1 and o_2
$M_D(o_1, o_2)$	Minimum entailed rank difference between outcomes o_1 and o_2
$\text{HD}(o_1, o_2)$	Hamming distance between outcomes o_1 and o_2 considered as vectors
$d(o)$	Number of occurrences of outcome o in the choice data
p_i or p_{o_i}	The true probability of the user choosing outcome o_i
$S_r(\mathbf{u} : x \succ \bar{x})$	Rule score of the preference rule $\mathbf{u} : x \succ \bar{x}$
$S_t(\text{CPT}(X))$	CPT score of $\text{CPT}(X)$
$S_c(N)$	CP-net score of N
$S(G)$	Structure score of CP-net structure G
$\mathcal{T}(U, X)$	Set of all $\text{CPT}(X)$ possibilities, given that $\text{Pa}(X) = U$
$MaxS_t(X \mid U)$	Maximum $S_t(\text{CPT}(X))$ score over all $\text{CPT}(X) \in \mathcal{T}(X, U)$
$OptCPT(X \mid U)$	$\text{CPT}(X) \in \mathcal{T}(X, U)$ such that $S_t(\text{CPT}(X)) = MaxS_t(X \mid U)$
$A \oplus e$	Structure obtained by changing edge e in structure A
$\Delta(e)$	Multiplicative change in structure score caused by changing edge e

$C_{i,j}$	$(i,j)^{th}$ entry (row i column j) of cycles matrix C
α	Change threshold for learning
N_T	The user's true CP-net
N_L	CP-net learned from observed data

The following table lists commonly used abbreviations.

CP-Net	Conditional preference network
CPT	Conditional preference table
DFA	Data flip agreement
DOC	Data order consistency
IFS	Improving flipping sequence
PG	Preference graph
UVRS	Unimportant variable removal and separation (preprocessing)

Chapter 2

Outcome Rank Pruning for Efficient Dominance Testing

Most of the material presented in this chapter, as well as the associated proofs and appendices, has been previously published in our paper:

Laing, K., Thwaites, P. A., and Gosling, J. P. (2019). Rank pruning for dominance queries in CP-nets. *Journal of Artificial Intelligence Research,* 64:55–107.

2.1 Introduction

CP-nets represent conditional preferences over a given set of variables. These may be considered as local preferences over different features of a product. However, most questions of interest are about the user's preferences over the associated outcomes – the products the user is ultimately deciding between. In particular, the main reasoning tasks of interest are outcome optimisation, consistent orderings, and dominance testing (Allen et al., 2017a; Boutilier et al., 2004a,b; Brafman and Dimopoulos, 2004; Goldsmith et al., 2008; Santhanam et al., 2016).

Outcome optimisation aims to identify the outcome that is most preferred by the user (possibly under certain constraints). Answering such queries is important to applications such as automated decision making as it allows us to identify the best choice for the user. Boutilier et al. (2004a) show that optimality queries for acyclic CP-nets can be answered in linear time (in n, the number of variables). Their method can also obtain the optimal outcome when a partial variable assignment is specified.

In this chapter, we address the remaining two reasoning tasks. Consistent orderings are orderings of the outcomes that satisfy all of the known user preferences.

If the user prefers outcome a to outcome b, then a must come before b in every consistent ordering. We look at obtaining a consistent ordering over all outcomes as well as over any subset of the outcomes. Consistent orderings are important to recommendation systems; for example, in e-commerce, one would prefer the user to be presented with the products in order of the user's preferences. This would mean that the items of most interest to the user are presented first and, thus, the user is more likely to make a purchase.

Dominance testing asks, given a pair of outcomes, which does the user prefer? This is essential for decision support applications, as we must be able to determine the relative preference of potential outcomes. Boutilier et al. (2004b) show that dominance testing is necessary for constrained optimisation tasks (where the constraints are more complex than a partial variable assignment). However, despite being a natural query, dominance testing has been shown to be PSPACE-complete (see Appendix F for definition) for acyclic CP-nets (Goldsmith et al., 2008) and is, thus, difficult to perform efficiently, particularly for larger CP-nets. In order for CP-nets to be a practical choice of preference representation for applications, we must be able to perform these reasoning tasks efficiently.

In this chapter, we start by constructing a quantitative representation of user preference over outcomes, given an acyclic CP-net model of preference. These quantities are called outcome ranks. We show that these ranks can be used to construct a consistent ordering of (any subset of) the outcomes. This method is more efficient for large subsets than the method proposed by Boutilier et al. (2004a). Furthermore, we can use these ranks to make dominance testing more efficient via pruning the associated search tree. Our pruning technique can also be combined with any of the existing pruning methods to further improve dominance testing efficiency. We provide an experimental comparison between the performance of our rank pruning and the existing pruning methods. These experiments also evaluate the performance of all possible combinations of the different methods. This enables us to determine the optimal pruning schema for answering dominance queries efficiently. The results of these experiments show rank pruning to be more effective than the existing methods and an essential component for a successful combination of methods. Finally, we shall provide a generalisation of our outcome ranks that is defined for CP-nets that contain indifferences. This generalisation extends all of our prior results to CP-nets with indifference. Note, in particular, that our method of consistently ordering subsets of outcomes has the same complexity with and without indifference. Comparatively, the method by Boutilier et al. (2004a) has unknown complexity in the case of indifference, though they conjecture that it is hard.

The rest of the chapter is structured as follows; in §2.2, we review the existing literature on quantifying CP-net preference, consistent ordering methods, and improving dominance testing efficiency. In §2.3, we introduce our outcome ranks and demonstrate how they can be used to obtain consistent orderings. In §2.4, we explain how outcome ranks can be used to improve dominance testing efficiency and provide an experimental evaluation (and comparison) of the performance of this method. In §2.5, we generalise our outcome ranks to be defined for CP-nets with indifference and extend the previous results to this case. Finally, we provide a discussion of these results and related future work in §2.6.

2.2 Related Work

As discussed in §2.1, in this chapter, we introduce a novel quantification of user preference and address the problems of obtaining consistent orderings and efficient dominance testing (all assuming that a user's preferences are represented by a CP-net). We review the existing literature on these three topics in the following three subsections.

2.2.1 Quantitative Preference Representation

Boutilier et al. (2001) introduce an extension of CP-nets called UCP-nets (utility CP-nets). The motivation for these structures is that they would both represent a global utility function over the outcomes and a CP-net. The former is a quantitative preference and would make dominance queries trivial. The latter allows the identification of optimal outcomes in linear time. UCP-nets look identical to CP-nets except that each CPT row gives a numerical function defined over the variable domain, rather than a preference ordering. These functions can be considered as local utility functions. Under the assumption of general additive independence, the utility of an outcome is simply the sum of the relevant function values. Thus, such a model represents a complete utility function over the outcomes. However, in order to obtain this utility function, we must first elicit a UCP-net, which is more complex than a CP-net. The authors do not discuss how to directly elicit the local utility functions. Instead, they suggest using normalised UCP-nets, where each local utility function is in $[0, 1]$. This makes elicitation of the local utilities a fairly simple task. However, in this case, calculating the outcome utilities also requires tradeoff weights. The elicitation method by Boutilier et al. (2001) focuses on narrowing down the tradeoff weight options with the aim of choosing the optimal decision from a given set of options. They do not detail how to obtain or elicit the actual tradeoff weights. Thus, given a CP-net and elicited normalised local utilities, we still cannot construct the global outcome utilities, which is the aim of quantifying CP-net preference.

Domshlak et al. (2003) introduce two approximations of CP-net preferences in order to obtain a consistent outcome ordering. The first is a real valued penalty associated with each outcome. The penalty of an outcome is defined as follows (notation adapted for clarity):

$$\begin{gathered}\text{pen}(o) = \sum_{X \in V} w_X p(o, X), \\ w_X = \begin{cases} \sum_{Y \in \text{Ch}(X)} w_Y |\text{Dom}(Y)| & \text{if Ch}(X) \neq \varnothing, \\ 1 & \text{otherwise.} \end{cases}\end{gathered} \tag{2.1}$$

The set of variables that have X as a parent is denoted by $\text{Ch}(X)$ and we refer to this set as the *children* of X. The local penalties, $p(o, X)$, denote an integer penalty indicating to what degree the value of X in o is preferred. Suppose $\text{CPT}(X)$ contains the rule $o[\text{Pa}(X)] : x_1 \succ x_2 \succ \cdots \succ x_{|\text{Dom}(X)|}$. If $o[X] = x_1$, then $p(o, X) = 0$. If $o[X] = x_{|\text{Dom}(X)|}$, then $p(o, X) = |\text{Dom}(X)| - 1$. In general, if $o[X] = x_i$, then $p(o, X) = i - 1$. This penalty represents how much the user's preference has been violated by the choice of X in o (the worse the choice of X is, the higher the penalty). The weights, w_X, are present to ensure that any preference violation of a variable dominates all possible violations of its children's preferences. This is necessary as CP-net semantics dictate that ancestor variables are more important to the user than their descendants. We say that variable A is an *ancestor* of variable B (and, similarly, B is a *descendant* of A) if there is a directed path $A \rightsquigarrow B$ in the CP-net structure. Domshlak et al. (2003) prove that these penalties accurately represent the CP-net preferences. That is, $N \vDash o_1 \succ o_2 \implies \text{pen}(o_1) < \text{pen}(o_2)$. Thus, the outcome ordering induced by the penalty values is a consistent ordering. These penalties can be computed from the CP-net in polynomial time.

The second approximation associates each outcome with a vector of size $|V|$ via the following procedure. Let $X_1, ..., X_n$ be a topological ordering of the variables. That is, $\text{Pa}(X_i) \subseteq \{X_1, ..., X_{i-1}\}$. Let o be an associated outcome. They first compute a vector for each X_i. Let $m = D_{max} - 1$, where D_{max} is the size of the largest variable domain. The vector for X_i will have the entry m in every position except the i^{th} position. Suppose that $\text{CPT}(X_i)$ contains the rule $o[\text{Pa}(X_i)] : x_1^i \succ\ x_2^i \succ \cdots \succ x_k^i$ (where $k = |\text{Dom}(X_i)|$). If $o[X_i] = x_j^i$, then the i^{th} position of the X_i vector contains the value $m - j + 1$ – the more preferred $o[X_i]$ is, the larger the i^{th} position entry. The vector associated with o is then the minimum (lexicographically) of the vectors associated with the variables. Note that, this implies that the vector of o is the X_i vector where i is the minimum value such that $o[X_i]$ is not the most preferred value. If no such i exists, then the vector associated with o has m in every position. Outcomes are then ordered lexicographically according to their associated vectors. Domshlak et al. (2003) claim that this ordering is also consistent with the CP-net and, thus, this approximation also accurately reflects the CP-net preferences. However, this is contradicted by the following example.

Example 2.1. Consider a CP-net, N, with four variables, A, B, C, D, and no edges. Let the CPT of $X \in V$ be $x \succ \bar{x}$. Let $o_1 = a\bar{b}cd$ and $o_2 = a\bar{b}\bar{c}d$. As $c \succ \bar{c}$, we have $N \vDash o_1 \succ o_2$. As there are no edges, A, B, C, D is a topological ordering.

Let us compute the variable vectors for o_1. As all vectors are binary, $m = \; 2 - \; 1 = 1$. As A takes its preferred value, the first entry in the A vector is $m - 1 + 1 = 1$. The rest of the entries of the A vector are $m = 1$, so it is $(1, 1, 1, 1)$. Similarly, the C and D vectors are also $(1, 1, 1, 1)$. However, B takes the second most preferred value. Thus, the second entry of the B vector must be $m - 2 + 1 = 0$ (and the rest of the positions are $m = 1$). So the B vector is $(1, 0, 1, 1)$. Thus, the vector associated with o_1 is the lexicographic minimum of the following set:

$$\{(1, 1, 1, 1), (1, 0, 1, 1), (1, 1, 1, 1), (1, 1, 1, 1)\}.$$

Thus, the o_1 vector is $(1, 0, 1, 1)$. Note that this is the B vector and B is the first variable to not take its most preferred value.

Now let us compute the variable vectors for o_2. Again, A and D take their preferred values, so by the same argument they have vectors $(1, 1, 1, 1)$. B again takes the second most preferred value, so the B vector is $(1, 0, 1, 1)$ again. However, C now takes its second most preferred value. Thus, the third entry of the C vector must now be $m - 2 + 1 = 0$. So the C vector is $(1, 1, 0, 1)$. The vector associated with o_2 is the lexicographic minimum of the following set:

$$\{(1, 1, 1, 1), (1, 0, 1, 1), (1, 1, 0, 1), (1, 1, 1, 1)\}.$$

This is again $(1, 0, 1, 1)$. Note that this is the B vector and B is again the first variable to not take its most preferred value.

This contradicts the claim made by Domshlak et al. (2003) that the lexicographic ordering of the outcome vectors is a consistent ordering. If this were true, then $N \vDash o_1 \succ o_2$ would imply that the o_2 vector is lexicographically smaller than the o_1 vector. However, this is not the case as they are the same vector.

More generally, this outcome vector representation is contradicted by the following fact; if the (topologically) first variable that takes a not-preferred value in outcome o_1 is the same for outcome o_2 (and this variable takes the same value in both), then o_1 and o_2 will have the same associated vector. In particular, the associated vector will be the vector of the first variable that takes a not-preferred value. If the outcome vectors induce a consistent ordering, then the CP-net cannot entail a preference between o_1 and o_2. Any such preference would not be reflected in the consistent ordering and, thus, would be a contradiction as in the above example. However, there is no reason why such a pair of outcomes cannot have an entailed preference. In fact, a counterexample outcome pair can be constructed for any CP-net with two or more variables.

McGeachie and Doyle (2004) aim to construct a utility function that is consistent with any given set of *ceteris paribus* preference statements. That is, where every preference over the models (outcomes in the case of CP-nets) is reflected by the relative utility values. CP-nets represent a specific form of *ceteris paribus* preference statements and so, such a method can be used to obtain a utility function consistent with a given CP-net. However, as McGeachie and Doyle (2004) represent preferences as logical statements over Boolean features (variables), their methods are only appropriate for binary CP-nets. They present two approaches for obtaining a utility function consistent with a given set of preference rules, R. The first is to construct a 'model graph', $G(R)$. This is a directed graph with the set of models as nodes. A model assigns every feature exactly one truth value, in our context these are the CP-net outcomes. There is an edge $m_1 \rightarrow m_2$ in $G(R)$ if and only if $m_1 \succ m_2$ is a preference implied by the rules in R. For CP-nets, this graph is equivalent to the preference graph. They then define four possible utility functions over the models (outcomes) using the features of this graph:

- $u_M(m)$ = the number of unique nodes on the longest directed path in $G(R)$ that originates at m.
- $u_D(m)$ = the number of descendants of m in $G(R)$.
- $u_X(m)$ = the length of the longest path in $G(R)$ minus the number of unique nodes on the longest path terminating at m (and originating at a distinct node).
- $u_T(m)$ = the number of nodes in $G(R)$ minus the rank of m in a (given) topological sort of $G(R)$.

Each of these utility functions obeys all *ceteris paribus* preference rules in R. However, the calculation of these utilities has complexity exponential in the number of features (variables) in the worst case scenario. CP-nets fall into this worst case category as the preference graph has an exponential (in n) number of edges.

The second method of constructing a utility by McGeachie and Doyle (2004) is intended to improve efficiency. If S_1 and S_2 partition the feature set, then S_1 is utility independent of S_2 if the preferences over S_1 do not depend upon the values taken by S_2. That is, given any truth assignment to S_2, preference over the possible S_1 completions is fixed. Note that this is not a symmetric concept. The first task when constructing the utility is to find a partition of the features, $S'_1, ..., S'_Q$, such that the following condition holds for each S'_i; there exists some feature set T_i such that S'_i is dependent on T_i and independent of $V \backslash S'_i \cup T_i$. From this partition they form the sets $S_i = S'_i \cup T_i$. The utility function can then be assumed to be a weighted sum of sub-utilities, u_i, where u_i is a partial utility over S_i. For CP-nets, we can let S'_i be the variable X_i and $T_i = \text{Pa}(X_i)$. To determine the model (outcome) utility, it remains to find the sub-utilities and the sum weights.

In order to obtain u_i, a restricted version of the model graph, $G_i(R)$, is constructed over the models of S_i by considering only the rules in R that are relevant to S_i. From this graph, they want to construct u_i using one of the four methods used on $G(R)$ above. However, the process of restriction to S_i can cause $G_i(R)$ to contain cycles. Thus, it is necessary to select certain rules (out of those relevant) to not be included in $G_i(R)$. Selecting which rules are not included (that is, which rules disagree with the sub-utility functions) is formulated as a SAT problem that may be solved via a SAT-solver (though sometimes this can be done more simply). A solution to this SAT problem specifies the restricted model graphs and, thus, gives the sub-utilities. As the sub-utilities may disagree with certain rules, it is necessary to set the weights of the utility sum to ensure that the global utility is still consistent with all rules in R. The necessary conditions for this are a set of inequalities, which can be solved via linear programming to obtain the weights. If a solution is obtained, then the associated utility is consistent with all preference rules in R. However, this process will not always identify a utility function. In which case, one can either merge some of the S_i sets and attempt the process again, or use the original method. Under certain assumptions, this process has polynomial complexity. However, the worse case scenario complexity remains intractable.

Li et al. (2011a) introduce another penalty function over CP-net outcomes that is a slight variation on the penalty function by Domshlak et al. (2003). This is again a preference representation where smaller values indicate that an outcome is more preferred by the user. Their definition of outcome penalties is as follows (notation changed for consistency):

$$\begin{aligned} \text{pen}(o) &= \sum_{X \in V} w'_X p(o, X), \\ w'_X &= 1 + \sum_{Y \in \text{Ch}(X)} w'_Y(|\text{Dom}(Y)| - 1). \end{aligned} \tag{2.2}$$

This is identical to the penalty function of Domshlak et al. (2003) except for a slight variation in the weight definition. Li et al. (2011a) have proven that their penalty function is an accurate representation of the CP-net preferences. That is, $N \models o_1 \succ o_2 \implies \text{pen}(o_1) < \text{pen}(o_2)$. These penalty values can be computed in polynomial time. Li et al. (2013) go on to generalise this penalty function so that it is also defined for TCP-nets (CP-nets with additional conditional importance statements).

Our outcome ranks, as defined in §2.3.2, can be computed directly from a given CP-net, unlike the utilities described by Boutilier et al. (2001). The global utility by Boutilier et al. (2001) requires additional local utilities to be elicited from the user and, further, the authors do not specify how to obtain the weights required to combine these local utilities. Our outcome ranks can be calculated from the CP-net with time complexity $O(n^4)$. This is more efficient than the utility construction methods by McGeachie and Doyle (2004), which may be intractable. Further, outcome ranks are defined for both binary and multivalued CP-nets, whereas the utilities of McGeachie and Doyle (2004) are defined in the binary case only. The penalty functions by Domshlak et al. (2003) and Li et al. (2011a) can also be computed from the CP-net in polynomial time. Both formulations look fairly similar to our outcome rank formula. However, in Example 2.7 we show that our outcome ranks are meaningfully distinct from both penalty functions (that is, they are not simply transformations of one another). In particular, we demonstrate that our ranks may evaluate the relative preference of a given outcome pair differently to the penalty functions. Thus, they must be distinct preference representations.

Only Li et al. (2011a) use their qualitative preference representation to improve upon dominance testing efficiency (details in §2.2.2). We compare our pruning method to theirs in §2.4.2 and demonstrate that ours is more effective. In §2.4.1, we show that any consistent ordering can be used to improve dominance testing efficiency by pruning the search tree. Thus, although it is not addressed by the authors, the penalty function by Domshlak et al. (2003) and the utility functions by McGeachie and Doyle (2004) could also be used to improve dominance testing efficiency.

We also generalise our rank definition to be defined for CP-nets with indifference. Indifference is not permitted by any of the existing preference representation definitions.

2.2.2 Consistent Ordering

The outcome ordering induced by the CP-net approximation given by Domshlak et al. (2003) (discussed in §2.2.1) is a consistent ordering. As are the orderings induced by the utility functions created by McGeachie and Doyle (2004) (if the utility is constructed from a CP-net). Similarly, the penalty values defined by Li et al. (2011a) also induce a consistent ordering, though this is not mentioned by the authors.

Boutilier et al. (2004a) obtain a consistent ordering by ordering the outcomes lexicographically as follows. Let N be a CP-net over variables $\{X_1, ..., X_n\}$. Assume these variables are in a topological order, that is, each variable's parents come before the variable itself. Suppose we have two outcomes, o_1 and o_2, that have the same values for $X_1, ..., X_k$, but differ on the value of X_{k+1}, say $o_1[X_{k+1}] = x_{k+1}$ and $o_2[X_{k+1}] = \bar{x}_{k+1}$. If, given the assignment of values to $\mathrm{Pa}(X_{k+1})$ in both o_1 and o_2, $\mathrm{CPT}(X_{k+1})$ dictates that $x_{k+1} \succ \bar{x}_{k+1}$, then o_1 is ordered before o_2. The resulting order over the outcomes is a consistent ordering.

Boutilier et al. (2004a) also suggest a method for consistently ordering any subset of the outcomes. Given a pair of outcomes, o_1 and o_2, an ordering query determines an ordering of o_1 and o_2 that is consistent with the corresponding CP-net. Note that the ordering $o_1 \succ o_2$ is consistent as long as $N \nvDash o_2 \succ o_1$ (that is, as long as it is not contradicting an entailment), we do not need $N \vDash o_1 \succ o_2$. Boutilier et al. (2004a) demonstrate how ordering queries can be answered in linear time (in n). A consistent ordering of any subset is then obtained by repeatedly performing ordering queries on outcome pairs in this subset and using the results to order them. This method has complexity $O(nk^2)$ for a subset of size k.

Sun et al. (2017) obtain all of the consistent orderings of a given CP-net by successively composing variable preferences to form a single preference table. First, each variable in the CP-net is turned into a relation table. For variables without parents, each row corresponds to a domain value and they are ordered according to user preference (given by the CPT). If a variable, X, has a parent set, U, then the rows correspond to the possible value assignments to $U \cup \{X\}$. The rows that assign the same values to U are adjacent and are ordered according to the user's preference over X under this U assignment. Each of these tables has an associated preference relation over the rows, dictated by the original CPT. This relation is not necessarily complete, that is, it does not always imply a preference between all row pairs. In topological order (starting with the variables that have no parents), these tables are then successively composed by taking Cartesian products. As the tables are composed, the associated preference relations are also composed via an extended version of Pareto composition.

Once all of the tables are composed, the resulting table has $\geq 2^n$ rows (corresponding to the CP-net outcomes) and the associated relation implies a relation between every pair of outcomes (this may be a preference, incomparability, or uncertainty). Note that, in the binary case, there are $2^{n-1}(2^n - 1)$ unordered outcome pairs and in general there are even more. Taking the transitive closure of this relation results in every outcome pair either having a preference or being incomparable. These preferences are exactly the CP-net entailments. That is, the original CP-net entails $o_1 \succ o_2$ if and only if this relation dictates $o_1 \succ o_2$. This relation can be used to construct a simplified version of the CP-net preference graph. Specifically, if there is a path $o_1 \rightsquigarrow o_2$, then there cannot also be a directed edge $o_1 \rightarrow o_2$. This simplified version of the preference graph has the same topological outcome orderings as the original (recall that, for the original preference graph, these topological orderings constitute the consistent orderings of the CP-net). Thus, the set of all consistent orderings can be found by determining the topological orderings of this simplified preference graph. However, the authors do not explore how simplifying the preference graph affects the complexity of finding topological orderings. Further, it is not clear whether their process of obtaining a simplified preference graph is more or less efficient than simply constructing the original preference graph from scratch and omitting any redundant edges.

Note that, as we discuss in §2.3, no consistent ordering is 'better' or more likely than any other. Thus, all of the consistent orderings produced by the above methods are equally 'good'. We will use our outcome ranks to induce a consistent ordering over the outcomes in a similar manner to Domshlak et al. (2003), McGeachie and Doyle (2004), and Li et al. (2011a). This method can be directly used to obtain a consistent ordering of any subset of the outcomes also. We will show that, for larger subsets, this is more efficient than the method by Boutilier et al. (2004a) for consistently ordering outcome subsets. Domshlak et al. (2003), McGeachie and Doyle (2004), and Li et al. (2011a) could order outcome subsets in the same manner, using their respective quantitative preference representations, but this is not discussed by these papers.

Only Li et al. (2011a) use their consistent ordering (in particular the penalty values) to make dominance testing more efficient. We use our rank ordering similarly, and in §2.4.2 we show that this is more effective than the penalty pruning by Li et al. (2011a). In §2.4.1, we show that any consistent ordering can be used to improve dominance testing efficiency by pruning the search tree. Thus, although it is not addressed by the authors, any of the consistent orderings obtained by the methods in this section could be used to prune dominance query search trees.

In §2.5, we generalise outcome ranks to be defined in the case of CP-nets with indifference statements. This means that we can also obtain a consistent ordering of (any subset of) the outcomes when the CP-nets contain indifference. Further, the complexity of finding a consistent ordering, or consistently ordering any set of outcomes, is the same as for CP-nets without indifference. Out of the existing methods discussed in this section, only Boutilier et al. (2004a) claim that their methods extend to this case. However, the complexity of answering ordering queries in this case is unknown (though they conjecture that it is hard). Thus, the complexity of their method for consistently ordering subsets of outcomes has unknown complexity in this case.

2.2.3 Dominance Testing

Suppose we have a CP-net N, and two associated outcomes o and o'. We want to answer the dominance query 'Is o preferred to o'?' efficiently. The standard method of answering this query is to try and construct an improving flipping sequence (IFS) from o' to o (Boutilier et al., 2004a). This is can be visualised as building up a search tree, $G(o')$, from the root node o', that either eventually reaches o (and so the dominance query is true) or eventually can not expand any further (and so the dominance query is false). We discuss these notions in further detail in §2.4.1. There have been several attempts to improve the efficiency of this method by introducing procedures for pruning the branches of $G(o')$ as one constructs it. We will improve dominance testing efficiency similarly by using our new outcome ranks to prune the search tree.

In this section, we first review the existing methods for pruning the search tree. Then we look at the other existing methods for improving dominance testing efficiency. Note that reading the paragraphs on page 58 (§2.4.1) on dominance testing via the construction of $G(o')$ will make the following discussion of pruning methods clearer.

Search Tree Pruning Methods

Note that, except for Boutilier et al. (2004a), the authors in this section do not frame their methods as search tree pruning. Even Boutilier et al. (2004a) use a slightly different search tree definition to us. We have explained their methods in terms of pruning the search tree $G(o')$ so that it is clearer how they relate and compare to the pruning method we present in §2.4.1. However, this is not a significant alteration of their works. Each work attempts to build up an IFS by successively moving through the preference graph with certain additional conditions upon which flips they consider. We are simply visualising this process as the construction of a search tree, where not considering a direction means pruning the relevant edge from this tree.

Boutilier et al. (2004a) introduced three methods for improving search efficiency. The first, **suffix fixing**, prunes the search tree, $G(o')$, as it is constructed. Let N be a CP-net over variables V and suppose $\{X_1, X_2, ..., X_n\}$ is a topological ordering of V. The k^{th} *suffix* of any outcome o^* is $o^*[X_k, X_{k+1}, ..., X_n]$. Suppose we are constructing $G(o')$ and the leaf $\bar{o}$ has the same k^{th} suffix as o. Then, when adding the improving flips of $\bar{o}$, any improving flips that do not have the same k^{th} suffix as o and $\bar{o}$ are pruned. This pruning condition preserves search completeness (meaning that if there is a successful path, then one will always be found and so dominance queries are always answered correctly when using this pruning method) as Boutilier et al. (2004a) proved the following result; if o and o' have the same k^{th} suffix and $N \vDash o \succ o'$, then there exists an improving flipping sequence $o' = o_1, o_2, ..., o_m = o$, such that every o_i has the same k^{th} suffix as o and o'.

The second method is called **least-variable flipping**, which also prunes $G(o')$ as it is constructed. In this case, only least-variable flips are considered and any other improving flips are pruned. That is, when adding the improving flips of a leaf, $\bar{o}$, the flips that change least-improvable variables (with respect to o) are added and any other improving flips are pruned. Given $\bar{o}$, a least improvable variable with respect to o is any variable, X, that satisfies the following properties; in the row of CPT(X) corresponding to Pa$(X) = \bar{o}[\text{Pa}(X)]$, $\bar{o}[X]$ is not the most preferred value – it can be improved. Further, no descendent of X in the structure of N has this property. That is, X is (one of) the 'lowest improvable variable'. Finally, X must not be part of a matching suffix between o and $\bar{o}$. Boutilier et al. (2004a) proved the following result for the case where N is binary and directed-path singly connected (that is, there is at most one directed path between any pair of nodes); if there is an IFS $o' \rightsquigarrow o$, then there is an IFS $o'_f \rightsquigarrow o$, where o'_f is some outcome obtained from o' by flipping a least improvable variable with respect to o. This result implies that, in the case of binary, directed-path singly connected CP-nets, pruning all flips except the least-improvable variable flips does not affect search completeness. However, for CP-nets in general (which may be multiply connected or have non-binary variables), this pruning method does not preserve search completeness. In fact, Boutilier et al. (2004a) suggest that the set of binary and directed-path singly connected CP-nets may be the widest class for which least variable flipping preserves search completeness.

For general CP-nets, Boutilier et al. (2004a) suggest that the notion of least-variable flips may be used as a search heuristic to improve search efficiency. That is, rather than pruning flips that do not change least-improvable variables, those that do are prioritised (these directions of the tree are explored first). For non-binary CP-nets, they suggest an extension to this heuristic. If X is a multivalued, least-improvable variable, then there may be several possible improving flips of X. Boutilier et al. (2004a) suggest that these flips should be considered in order of increasing level of improvement. Consider a tertiary variable X with preference $x \succ x' \succ x''$. If $X = x''$, then there are two possible improving flips – X could change to $X = x$ or to $X = x'$. By the above heuristic, the flip to $X = x'$ is considered first, as it less of an improvement than flipping to $X = x$.

Their final method is **forward pruning**. This technique prunes the variable domains in order to reduce the search space, rather than pruning the search tree as it is constructed. Let $\{X_1, X_2, ..., X_n\}$ be a topological ordering of V. The idea is to sweep forward through the variables (X_1 to X_n) and remove domain values that cannot appear in an $o' \rightsquigarrow o$ IFS. For each X_i, they first build a domain transition graph, DTG(X_i). This graph has Dom(X_i) as its nodes. For $\mathbf{u} \in \text{Dom}(\text{Pa}(X_i))$ that contains only un-pruned values of Pa(X_i), suppose the CPT(X_i) entry corresponding to $\mathbf{u}$ is $x_1 \succ x_2 \succ \cdots \succ x_m$. Then there is an edge $x_i \rightarrow x_{i-1}$ in DTG(X_i) for every $1 < i \leq m$. Any X_i value (node) that is not on a directed path $o'[X_i] \rightsquigarrow o[X_i]$ in DTG(X_i) is pruned from Dom(X_i). If all of Dom(X_i) is pruned (that is, there is no directed path $o'[X_i] \rightsquigarrow o[X_i]$), then the dominance query is false. Thus, forward pruning can, in some cases, determine a query to be false without needing to search at all. If no domain is pruned entirely, the search tree is then constructed over the set of outcomes that take un-pruned values only. This will be a smaller search space and so the search will be more efficient. Further, this pruning process preserves search completeness. The complexity of forward pruning is $O(nrd^2)$, where r is the maximum number of conditional preference rules for a variable and d is the maximum domain size. Note that r is exponential in the size of parent sets, which may be up to $n-1$.

Wilson (2004b) suggests an extension to suffix fixing called **prefix fixing**. He proposes this as a method of making dominance testing more efficient for CP-theories, which are a strict generalisation of CP-nets. Suppose we wish to answer the dominance query $N \vDash o \succ o'$. If we have $o[X] = o'[X]$ for some $X \in V$ and $o[Y] = o'[Y]$ for all descendants of X ($\text{Dec}(X)$), then when using suffix fixing, we do not need to consider any X flips (nor any flips of its descendants). This is because, if there is an IFS $o' \rightsquigarrow o$, then there is an IFS that preserves any matching suffix. Prefix fixing means that we do not consider any flips of X (or its ancestors) if $o[X] = o'[X]$ and $o[Y] = o'[Y]$ for all ancestors of X ($\text{Anc}(X)$). In this case, Wilson (2004b) asserts that *all* IFS $o' \rightsquigarrow o$ preserve such matching prefixes.

The author suggests that both methods should be used to improve the efficiency of searching for an IFS. That is, for CP-nets, if X and $\text{Anc}(X)$ take the same values in both o and o', then any improving flip that changes variable X is pruned when constructing $G(o')$. Similarly, if X and $\text{Dec}(X)$ take the same values in both o and o', then all X improving flips are pruned when constructing $G(o')$. Boutilier et al. (2004a) prove that suffix fixing preserves search completeness in the case of CP-nets. However, Wilson (2004b) does not provide an explicit proof that this also holds for CP-theories. He also does not explicitly prove that prefix fixing preserves completeness in either the CP-net or CP-theory case (nor that every IFS preserves matching prefixes). In Chapter 3, we provide a proof that prefix fixing does preserve search completeness (and that all IFSs preserve matching prefixes) in the case of CP-nets, as we utilise this result in our preprocessing. Thus, as both prefix and suffix fixing preserve search completeness for CP-nets, the combination suggested by Wilson (2004b) must also preserve completeness in the CP-net case at least.

Li et al. (2011a) use their penalty function (defined in §2.2.1) to prune the search tree in an analogous method to how we use outcome ranks. We shall refer to this as **penalty pruning**. With respect to the dominance query 'Is $o \succ o'$?', they first define the following evaluation function:

$$f(o^*) = \text{pen}(o^*) - \text{pen}(o) - \text{HD}(o^*, o),$$

where HD is *Hamming distance*, $\text{HD}(o_1, o_2) = |\{X | o_1[X] \neq o_2[X]\}|$. Li et al. (2011a) have shown that, if there is an IFS $o' = o_1, o_2, ..., o_m = o$, then $f(o_i) \geq 0$ for all i. Thus, when constructing $G(o')$, any improving flips with $f < 0$ can be pruned. This preserves completeness of the search by the above result. Li et al. (2011a) present this method combined with suffix fixing by Boutilier et al. (2004a). That is, flips are pruned if they either have $f < 0$ or they violate a matching suffix. When constructing $G(o')$, they prioritise (that is, search first) leaves with smaller f values. Note that, like forward pruning, this procedure also has the capacity to determine the dominance query to be false without the need to perform a search; if $f(o') < 0$, then we know the dominance query to be false. Li et al. (2011a) experimentally compare penalty pruning combined with suffix fixing to suffix fixing alone and also least variable flipping. These experiments compare the size of the search performed by each method, but they do not compare the time taken by each method. The results suggest that penalty pruning with suffix fixing is significantly more effective than suffix fixing alone. There are some cases where least variable flipping is the most effective pruning method. However, this occurs for larger n values where least variable flipping is shown to have a high probability of answering queries incorrectly (due to its lack of completeness). Li et al. (2013) claim that the Li et al. (2011a) pruning method can be extended to dominance testing for TCP-nets, however this is not shown explicitly.

Allen et al. (2017a) perform an experimental evaluation of the length of minimal flipping sequences. That is, for entailed preferences, $N \models o_1 \succ o_2$, they evaluated the length of the shortest $o_2 \rightsquigarrow o_1$ IFS. If all entailed preferences can be proved by an IFS with length below a given bound, this would simplify the task of searching for an IFS and, hence, simplify dominance testing. From these experimental results, they conjecture that, in the case of binary CP-nets, the minimum IFS length (if one exists) has an upper bound of $\lfloor (n+1)^2/4 \rfloor$. This conjecture is backed up by their experimental results but is not proven to hold in general. Further, this bound explicitly does not hold in the case of CP-nets with multivalued variables.

Allen et al. (2017a) propose that dominance testing can be performed more efficiently by only searching $G(o')$ to a specified depth. We shall refer to this as **depth-bounded search**. In particular, they apply this depth bound to the dominance testing procedure proposed by Li et al. (2011a) (using penalty pruning and suffix fixing). In this case, an improving flip is pruned if either $f < 0$, or it violates a matching suffix, or it is beyond the specified depth in the search tree. Such a depth bound could similarly be applied to our rank pruning method for dominance testing. The authors also construct a formula that is true if and only if there exists an IFS between the relevant outcome pair that has length below the specified bound. The form of this formula means it is a SAT problem and can thus be solved using a SAT-solver. The modified Li et al. (2011a) procedure and the SAT-solver will return 'true' if and only if there is an $o' \rightsquigarrow o$ IFS of length less than the supplied bound, k. Allen et al. (2017a) propose repeatedly applying one of these methods with successively larger upper bounds up to some specified bound k. However, as we discussed above, their conjectured bound upon the minimal IFS length is not proven and does not hold in the non-binary case. Thus, there is not a known (non-trivial) bound that could be imposed upon the search depth that is guaranteed to preserve completeness in general. That is, that guarantees that their process will return 'true' if and only if the dominance query holds.

Our method for improving dominance testing efficiency prunes the search tree using outcome ranks, similar to penalty pruning by Li et al. (2011a). We show that our pruning method preserves search completeness in the case of both binary and multivalued CP-nets, unlike least variable flipping and depth-bounded search. Rank pruning can also be combined with any of the existing pruning methods to further improve efficiency. In our performance evaluation experiments, we compare rank pruning to both suffix fixing and penalty pruning. We treat penalty pruning separately (unlike Li et al., 2011a, where they present it in combination with suffix fixing), in order to see more clearly how effective each of the pruning methods are, both individually and in different combinations. Our experimental results show that rank pruning is more effective than both penalty pruning and suffix fixing, as well as their combination. We also find that, when considering pruning combinations, rank pruning is an essential component for effective pruning and efficient dominance testing. These experimental results also demonstrate that all three pruning methods are distinct – each method prunes branches of the search tree that are not pruned by the other two. Prefix fixing is not compared in these experiments (though, by symmetry, we may expect it to perform similarly to suffix fixing), however, we can see that it is distinct from rank pruning as rank pruning can prune improving flips of any variable (including variables that take different values in the outcome pair of interest)

Like penalty pruning, rank pruning can also show the dominance query to be false via a simple numerical check (performed prior to constructing the search tree). We compared the performance of these two numerical checks in the experiments discussed above (Li et al., 2011a did not look at the numerical check results in their experiments). These results show the rank numerical check to be a much stronger condition. That is, it identified a much higher proportion of the false dominance queries (thus, a higher proportion of cases were answered without needing to conduct a search).

Unlike the pruning methods compared by our experiments, forward pruning reduces the size of the dominance query by preprocessing the CP-net, rather than pruning the search tree. Thus, forward pruning reduces the size of the problem rather than providing a method for answering the query. Forward pruning results in a smaller dominance query, which then needs to be answered efficiently. Thus, we do not include it in these experimental comparisons. However, we will consider it in Chapter 3, where we look at CP-net preprocessing to improve dominance testing efficiency. Note that, if forward pruning was reframed as a pruning condition, it would have complexity $O(nrd^2)$ for every leaf of the search tree considered. In comparison, suffix fixing, penalty pruning, prefix fixing, and rank pruning are all linear or polynomial in n to check.

In our experiments, we use outcomes traversed to measure the complexity of the dominance testing procedure. This is similar to the metric used in the experiments by Li et al. (2011a). However, we also measure the time elapsed to illustrate the true efficiency of the different methods. These experiments also vary the leaf prioritisation method used, in order to see whether this choice has an effect on performance. Boutilier et al. (2004a) and Li et al. (2011a) have both suggested leaf prioritisation heuristics (and we shall introduce our own), but there has been no experimental investigation into their efficacy. Our experimental results suggest that one of our proposed prioritisation heuristics is the optimal choice for efficient dominance testing (though in general we do not find the choice of prioritisation method to make a significant difference to performance).

In §2.5, we generalise our outcome ranks so that they are defined for CP-nets with indifference statements within their CPTs. Further, we show that this generalisation allows us to use a similar rank pruning procedure to improve dominance testing efficiency in the case of indifference. Out of all of the existing work on improving dominance testing efficiency, only Boutilier et al. (2004a) claim that their methods extend to this case, though this is not shown explicitly.

Other Methods

Santhanam et al. (2010, 2016) introduced the idea of using model checking to answer dominance queries efficiently. Their method is applicable to more general preference structures than CP-nets, for example TCP-nets and CP-theories. Technically, their procedure may be applied to any preference structure where dominance is defined in terms of an outcome graph property (for CP-nets this is reachability within the preference graph). In order to use model checking, the CP-net (or other preference structure) must first be translated into a Kripke structure (see Appendix F for definition). Roughly, the outcomes (with some additional information about which values may change) translate into the states of the models and the flips (edges) of the preference graph become the transitions between states. This Kripke structure is proven to correspond to the original preference graph.

Once the Kripke structure is constructed, they define a CTL (computational tree logic) formula, ϕ, for a given dominance query, $o \succ o'$. This formula, ϕ, is satisfied by the Kripke structure if and only if, for every state corresponding to o, there is sequence of state transitions that terminates at a state corresponding to o' (this sequence corresponds to a worsening flipping sequence). Thus, if the Kripke structure satisfies ϕ, $o \succ o'$ is true. In order to evaluate whether this holds, the initial states of the Kripke structure are set to be the states corresponding to o and a model checker is used to evaluate whether the Kripke structure satisfies ϕ. If the model checker returns 'true', then the dominance query holds. The authors claim that any model checker that accepts Kripke structures can be used, though they use a model checker called NuSVM. This procedure can also be used to obtain a proof of dominance (a worsening flipping sequence). If ϕ holds, then using the model checker to query satisfiability of $\neg\phi$ will return 'false' with a counterexample. This counterexample will be a sequence of state transitions from an o state to an o' state – corresponding to a worsening flipping sequence $o \rightsquigarrow o'$. Santhanam et al. (2010) provide an experimental evaluation of the efficiency of this method for dominance testing on CP-nets and TCP-nets. These results showed the average dominance testing time to be less than 13 seconds for binary CP-nets with up to 30 variables (with restrictions on variable degrees and CPT sizes). However, no experimental analysis has been done in the case of CP-nets with multivalued variables. Thus, it is not clear how well this technique performs when there are multivalued variables.

Sun et al. (2017) also propose answering dominance queries by successively composing variable preferences to form a single preference table. In §2.2.2, we described their method of composing preference tables and relations to obtain a relation over Ω that specifies exactly the CP-net entailments. Thus, the resulting relation answers every possible dominance query. However, this procedure requires building a table of exponential size and applying Pareto composition to an exponential number of outcome pair relations. Further, their experimental evaluation of the efficiency and space requirements are only provided in the binary case for CP-nets with up to 10 variables. Thus, we do not know how efficient this method is when there are multivalued variables, or for larger n values.

Unlike both of these methods, our experimental comparisons look at both the binary and non-binary CP-net cases. Thus, we provide a more complete picture of how efficient our dominance testing procedure is. For practicality, our experimental comparisons are limited to dominance testing procedures that answer queries by constructing (and pruning) a search tree. Thus, these two methods are not included in our comparisons.

Ahmed and Mouhoub (2019) give an algorithm for answering dominance queries that applies itself recursively. Suppose we are interested in answering the dominance query $N \vDash o \succ o'$. The algorithm starts by identifying a variable, $X \in V$, such that all ancestors of X take the same values in o and o', but $o[X] \neq o'[X]$. It then evaluates some trivial conditions that may answer the query immediately. If $o'[X] \succ o[X]$, given the values taken by $\mathrm{Pa}(X)$ in o and o', then the query is false. If $o[X] \succ o'[X]$ and all other variables take the same value in both o and o', then the query is true. If these checks do not answer the query, then the algorithm constructs a series of reduced queries and answers them by calling itself recursively. Depending on the answers of these queries, the original query is determined true or false.

Let $W = V \backslash \mathrm{Anc}(X) \cup \{X\}$ and let $o[\mathrm{Anc}(X)] = o'[\mathrm{Anc}(X)] = \mathbf{u}$, $o[X] = x_1$, and $o'[X] = x_2$. First, the algorithm considers whether the dominance query holds when reduced to the CP-net N_1, obtained from N by fixing $\mathrm{Anc}(X) = \mathbf{u}$ and $X = x_1$. If this smaller query holds, the algorithm returns 'true' (the original query holds). If this query is false, then the algorithm next evaluates the original query reduced to the CP-net N_2, obtained from N by fixing $\mathrm{Anc}(X) = \mathbf{u}$ and $X = x_2$. Again, if this smaller query holds, the algorithm returns 'true'. If neither of these queries are true, then the algorithm evaluates the query reduced to each N_i in turn for $x_i \in \mathrm{Dom}(X)$ such that $x_1 \succ x_i \succ x_2$ (given $\mathrm{Anc}(X) = \mathbf{u}$). If any of these reduced queries are found true, then the algorithm returns 'true'.

If none of these reduced queries are found true, then the algorithm assess the two queries $N_1 \vDash o[W] \succ o^*$ and $N_2 \vDash o^* \succ o'[W]$ for each $o^* \in \mathrm{Dom}(W)$, by calling itself twice. If both queries hold for some o^*, then the algorithm returns 'true'. Otherwise, if this is not true for any o^*, the algorithm returns 'false' (the original query is false).

The idea of this algorithm is to improve dominance testing efficiency by breaking the problem down into smaller cases and checking trivial conditions that can answer the query immediately. However, if the final clause of the algorithm is called, it is possible that the algorithm will call itself an exponential number of times (as there are $\geq 2^{|W|}$ assignments to W). The authors confirm that the complexity of this algorithm is exponential in the worst case scenario. Furthermore, no experimental evaluation of the effect of this algorithm on dominance query efficiency is provided by Ahmed and Mouhoub (2019).

We believe their algorithm to be incorrect. In particular, that there are cases of true dominance queries where the algorithm returns 'false'. This is because the algorithm does not consider the possibility of an entailed preference, $N \vDash o \succ o'$, where all $o' \rightsquigarrow o$ IFSs utilise more than two X values non-trivially.

This paper was published during the late stages of writing up this thesis. As we only became aware of this work recently and believe the algorithm to contain significant errors, we have not included this work in any later comparisons.

2.3 Outcome Ranks

Given a CP-net representing the user's preferences, our aim is to quantify the user's preference for each outcome; we will call this value an outcome rank. These values should induce a consistent ordering over the outcomes as they must reflect all preferences entailed by the CP-net. In most cases, CP-nets do not fully specify the user's preferences over the outcomes. Rather, there are usually several outcome orderings that could be the user's true preference (consistent orderings). Furthermore, given a basic CP-net and no further information, we are unable to judge any consistent ordering to be more likely than another to be the user's true preference ordering. Thus, if we wish to order the outcomes according to user preference, then we can do no better than to find *any* consistent ordering.

We start this section by showing how CP-nets can be represented by event trees, this representation is necessary for the construction of our outcome ranks. We then define our outcome ranks and prove that these values accurately reflect user preference. We demonstrate how outcome ranks can be used to obtain a consistent ordering over (any subset of) the outcomes. This is also shown to work for CP-nets with additional plausibility constraints. Finally, we provide an algorithm that calculates our outcome ranks in $O(n^4)$ time.

2.3.1 Event Tree Representation of CP-Nets

Let N be a CP-net over variables V. We have mentioned previously that the induced preference graph, G_N, is an equivalent representation of this information. Another equivalent way of representing CP-nets is by an event tree (Edwards, 1983). We use this alternate representation to motivate and construct our quantification of user preferences in §2.3.2. The *event tree representation* of N, denoted $T(N)$, can be constructed in three steps.

First, put the variables in a topological order according to the CP-net structure, $V = \{X_1, ..., X_n\}$. That is, $\text{Pa}(X_i) \subseteq \{X_1, ..., X_{i-1}\}$. For the CP-net given in Example 1.2, there are two such orderings, $ABCD$ and $BACD$. We use $ABCD$ for simplicity.

Second, construct an event tree representing the successive events of X_1 taking a value, then X_2 taking a value, and so on up to X_n. The root node branches into $|\text{Dom}(X_1)|$ possibilities (each branch should be labelled with an associated element of $\text{Dom}(X_1)$). Then, each of these nodes branches into $|\text{Dom}(X_2)|$ possibilities (each labelled with an associated element of $\text{Dom}(X_2)$). And so on until each of $X_1, X_2, ..., X_n$ have all taken a value. The final tree has $|\Omega|$ root-to-leaf paths, corresponding to the outcomes. Figure 2.1 gives the event tree representation for the CP-net in Example 1.2 (ignore the branch weights for the moment).

Finally, the branches need to be labelled with the level of preference of the associated variable assignment. Suppose we are labelling the branch b, which represents that $X = x$ (for some $X \in V$). By inspecting the unique path from the root of the tree to the start of b, identify the values assigned to $\text{Pa}(X)$. From the appropriate row of $\text{CPT}(X)$, we can identify the position of preference of the choice $X = x$. If x is the best choice under this assignment to $\text{Pa}(X)$, then label b with '1^{st}', if it is the second best, then label it '2^{nd}', and so on. For the event tree of the CP-net in Example 1.2, at the first stage, we label the $A = a$ branch '1^{st}' and the $A = \bar{a}$ branch '2^{nd}' because of $\text{CPT}(A)$. Similarly, both $B = b$ branches have the label '1^{st}' and both $B = \bar{b}$ branches have the label '2^{nd}'. Now, consider the top-most instance of the tree branching into the options for C $(c, \bar{c}, \bar{\bar{c}})$. At this point, A and B have been assigned values a and b and so we are concerned with the corresponding (top) row of $\text{CPT}(C)$. From the CPT, we can see that, given the history of this path, c is preferred to $\bar{c}$ is preferred to $\bar{\bar{c}}$. Thus, we give the $C = c$ branch the label '1^{st}', the $C = \bar{c}$ branch the label '2^{nd}', and the $C = \bar{\bar{c}}$ branch the label '3^{rd}'. Labelling the rest of the C and D branches is a similar process. However, for the D branches we only need to look at the value previously taken by C to determine which $\text{CPT}(D)$ row to consult.

Proposition 2.2. *Let N be a CP-net and let $T(N)$ be the event tree representation of N. Then N and $T(N)$ are equivalent structures (they encode identical information). Recall that a CP-net consists of both the structure and the CPTs.*

Proof. See Appendix E.1.

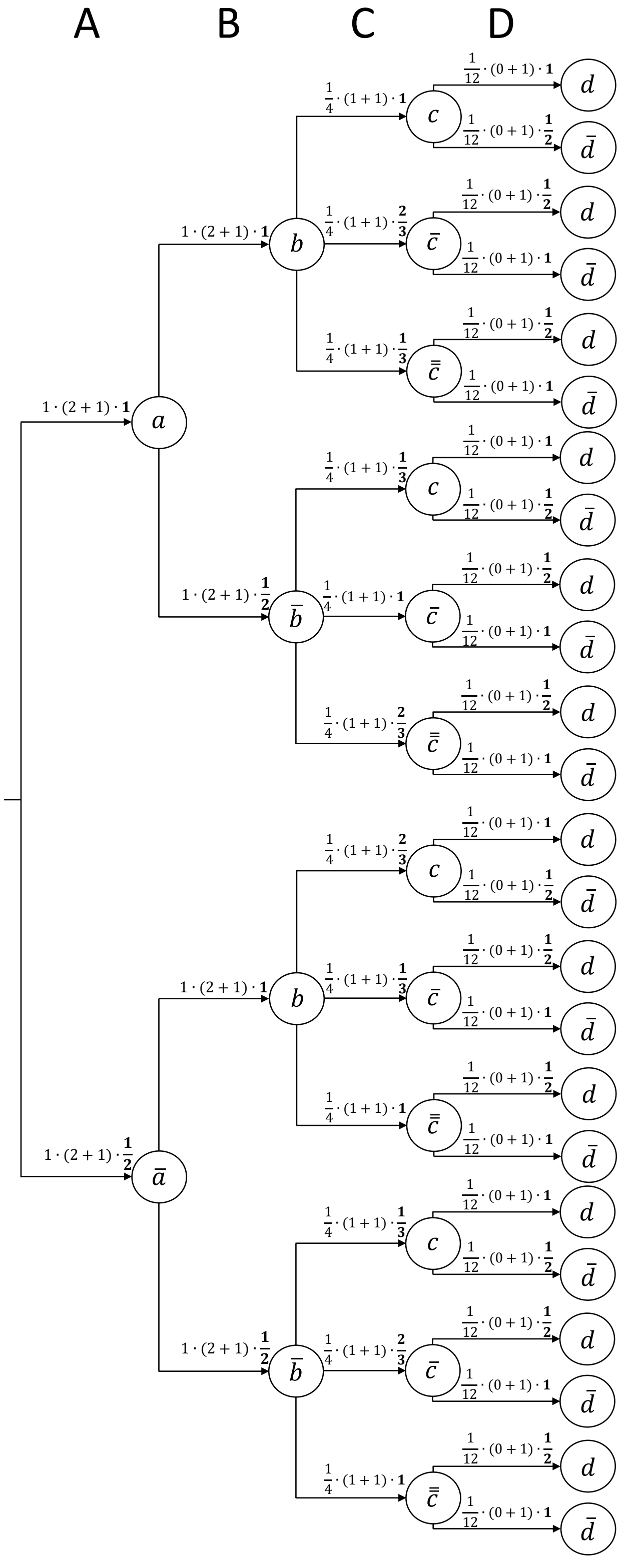

A
B
C
D
$1 \cdot (2+1) \cdot \mathbf{1}$
a
$1 \cdot (2+1) \cdot \mathbf{1}$
b
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{1}$
c
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
$\bar{d}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{2}{3}}$
$\bar{c}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
$\bar{d}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{1}{3}}$
$\bar{\bar{c}}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
$\bar{d}$
$1 \cdot (2+1) \cdot \mathbf{\frac{1}{2}}$
$\bar{b}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{1}{3}}$
c
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
$\bar{d}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{1}$
$\bar{c}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
$\bar{d}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{2}{3}}$
$\bar{\bar{c}}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
$\bar{d}$
$1 \cdot (2+1) \cdot \mathbf{\frac{1}{2}}$
$\bar{a}$
$1 \cdot (2+1) \cdot \mathbf{1}$
b
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{2}{3}}$
c
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
$\bar{d}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{1}{3}}$
$\bar{c}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
$\bar{d}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{1}$
$\bar{\bar{c}}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
$\bar{d}$
$1 \cdot (2+1) \cdot \mathbf{\frac{1}{2}}$
$\bar{b}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{1}{3}}$
c
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
$\bar{d}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{2}{3}}$
$\bar{c}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
$\bar{d}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{1}$
$\bar{\bar{c}}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
$\bar{d}$

Figure 2.1: Weighted Event Tree

From the example used above, it is clear that $T(N)$ can become very large even for relatively small CP-nets. This is because $T(N)$ will always branch into $|\Omega|$ distinct paths ending in $|\Omega|$ leaf nodes and $|\Omega| \geq 2^n$ (with equality only in the case of binary CP-nets). As mentioned previously, we use this event tree representation to aid the construction of our outcome ranks in §2.3.2. However, in §2.3.5, we demonstrate that constructing the event tree is not necessary for their calculation, so the exponential size of these trees is not a limitation.

Remark. While $T(N)$ quickly becomes an impractical representation due to its size, it has the advantage of flexibility over the usual CP-net representation. Event trees can be adapted to represent asymmetric information, whereas the compact CP-net notation does not have room to incorporate such information and so it must be reported separately. For example, in §2.3.4, we show how event trees are adapted in the case of additional plausibility constraints. A similar adaptation can be used in the case of missing information. Thus, if a CP-net is combined with a lot of additional information, it may become more practical to work with $T(N)$ rather than N. Note that, in general, event trees can be used to represent any qualitative preference structure where each variable is preferentially independent of its descendants, given its ancestors.

2.3.2 Outcome Rank Construction

In this section, we define our outcome ranks (which successfully induce a consistent ordering over the outcomes). These ranks are obtained using the event tree representation discussed in §2.3.1. Specifically, we first weight the edges of the event tree representation and then read off the rank of an outcome from this weighted tree. These outcome ranks reflect user preference, so more preferred outcomes receive higher scores.

To motivate our weighting convention for the edges of $T(N)$, we must look at what determines the user's level of preference for a given outcome, o. The position of preference of the values taken by the individual variables, according to the CPTs, needs to be taken into account. However, according to the semantics of CP-nets, ancestor variables in the CP-net structure are more important to the user than their descendants (Boutilier et al., 2004a). Thus, if variable A is an ancestor of variable B, then when quantifying user preference over outcomes, we must have a larger penalty for violating the user's preference over A than for violating their preference over B. Therefore, the position of variables in the CP-net structure will also need to be taken into account when determining the user's level of preference for an outcome.

As we allow our CP-net variables to be multivalued, we must also take into account how domain size affects user preference. By the semantics of CP-nets, domain size should not affect the importance of a variable flip. That is, the importance of changing variable X from the i^{th} most preferred value to the $(i-1)^{th}$ should be independent of the size of $\mathrm{Dom}(X)$. Suppose we have variables X and Y, such that Y is a descendant of X in the CP-net structure. Then any decrease of preference in X should dominate any decrease of preference in Y, regardless of their domain sizes. Thus, our quantification of preference must also have this property.

Motivated by these restrictions imposed by the CP-net semantics, we have created the following weighting formula for the branches of the event tree representation of a CP-net.

Definition 2.3. Let N be a CP-net over variables $V = \{X_1, ..., X_n\}$ and assume that the variables are in a topological ordering with respect to the structure of N. Now, consider the event tree representation of N, $T(N)$. Let e be the edge of $T(N)$ that indicates variable X_i takes value x_i, given that $X_1, ..., X_{i-1}$ take values $x_1, ..., x_{i-1}$, respectively. Use p to denote the directed path from the root of $T(N)$ to the start of e, which dictates in turn that $X_1 = x_1$, $X_2 = x_2, ..., X_{i-1} = x_{i-1}$. Let $\mathbf{u} \in \mathrm{Dom}(\mathrm{Pa}(X_i))$ be the assignment of values to the parents of X_i dictated by p. We define the *edge weight* of e as follows:

$$\left(\prod_{Y \in \mathrm{Anc}(X_i)} \frac{1}{n_Y}\right)(d_{X_i}+1)\frac{n_{X_i}-k+1}{n_{X_i}}, \tag{2.3}$$

using the following notation:

- $n_{X_i} = |\mathrm{Dom}(X_i)|$,
- d_{X_i} is the number of distinct directed paths of any (positive) length in the structure of N that originate at X_i (the number of *descendent paths* of X_i),
- k is the position of preference of the choice of $X_i = x_i$ given $\mathrm{Pa}(X_i) = \mathbf{u}$. So, if $X_i = x_i$ is the best choice for the user, then $k = 1$, if it is the second best choice, then $k = 2$, and so on. If it is the worst possible choice for X_i, then $k = |\mathrm{Dom}(X_i)|$.

We refer to the leftmost product term in Equation 2.3 as the *ancestral factor* of X_i, AF_{X_i}. This factor scales the weight down by the size of X_i's ancestors' domains. The purpose of this is so that any violation of preference for an ancestor will dominate a violation of preference for X_i, regardless of the size of the ancestor's domain relative to $|\mathrm{Dom}(X_i)|$.

Consider the central term of Equation 2.3, $(d_{X_i} + 1)$. If X is an ancestor of Y, then $d_X > d_Y$. An ancestor variable is more important to the user than its descendent variables, this term allocates these more important variables more weight. In particular, this term ensures that reductions in preference of an ancestor variable have larger penalties than reductions in preference of a descendant.

We refer to the rightmost product term in Equation 2.3 as the *preference position* of the choice $X_i = x_i$ given $\mathrm{Pa}(X_i) = \mathbf{u}$, denoted $P_P(X_i = x_i \mid \mathrm{Pa}(X_i) = \mathbf{u})$. This is a value in $\{1/n_{X_i}, 2/n_{X_i}, ..., (n_{X_i} - 1)/n_{X_i}, 1\}$. This is simply a factor on the (0,1] scale indicating to what degree the user prefers this choice of value for X_i. This naturally impacts the user's preference for the overall outcome. This factor gets larger for more preferred values, with the best value assigned a preference position of 1.

Remark. Notice that the preference position factor decreases in equal increments. Due to a lack of information provided by the CP-net, we cannot justify a more complex increment when quantitatively representing the user's preferences over $\mathrm{Dom}(X_i)$. Consider a variable A with $\mathrm{Dom}(A) = \{a_1, a_2, a_3\}$ and CPT $a_1 \succ a_2 \succ a_3$. This could mean that, to the user, a_2 is *slightly* worse than a_1, but a_3 is *much* worse than a_2. Alternatively, it could be that a_2 is *much* worse than a_1, but a_3 is only *slightly* worse than a_2. We cannot determine which of these is the case as CP-nets provide only qualitative (relative) preferences, and so we assume that preference decreases in equal increments each time. In this situation, our preference positions would be 1, $\frac{2}{3}$, and $\frac{1}{3}$ for a_1, a_2, and a_3 respectively.

However, note that the validity of outcome ranks does not rely upon the user's true preferences increasing in these equal increments. Our interest in outcome ranks lies in their relative sizes, rather than the actual rank values, and all CP-net preferences are accurately reflected by the relative outcome ranks (as we shall prove). This holds because our equal increment assumption is a plausible model for user preference, given the CP-net. Furthermore, the rank induced ordering is also not reliant upon the equal increment assumption to be valid; the user's true preference order can be our rank ordering even if our equal increment assumption is not true.

Let $W(N)$ be the event tree representation of N with the weights from Definition 2.3 attached. We refer to $W(N)$ as the *weighted event tree representation* of N.

Example 2.4. We now return to the CP-net, N, from Example 1.2 and the corresponding event tree, $T(N)$, given in §2.3.1. Simple examination of the CP-net structure and CPTs gives us the following results:

$$\text{Anc}(A) = \varnothing,\ \ \text{Anc}(B) = \varnothing,\ \ \text{Anc}(C) = \{A, B\},\ \ \text{Anc}(D) = \{A, B, C\},$$
$$n_A = 2,\ \ n_B = 2,\ \ n_C = 3,\ \ n_D = 2,$$
$$d_A = 2,\ \ d_B = 2,\ \ d_C = 1,\ \ d_D = 0.$$

From the n_X values and the ancestor sets, we can calculate the ancestral factor of each variable.

$$AF_A = 1,\ \ AF_B = 1,$$
$$AF_C = \frac{1}{2} \times \frac{1}{2} = \frac{1}{4},$$
$$AF_D = \frac{1}{2} \times \frac{1}{2} \times \frac{1}{3} = \frac{1}{12}.$$

We can now use these values and the CPTs to directly calculate the edge weights and, thus, construct the weighted event tree representation of N. $W(N)$ is given in Figure 2.1, with the preference position in each edge weight given in boldface.

Take, for example, the top-most edge in $T(N)$ that assigns a value to C. This edge assigns $C = c$, given that $A = a$ and $B = b$ have been assigned previously. By Definition 2.3, the associated edge weight is

$$AF_C\ (d_C + 1)\ P_P(C = c|\text{Pa}(C) = ab) = AF_C\ (d_C + 1)\ \frac{n_C - k + 1}{n_C},$$

where k is the position of preference of the choice $C = c$, given $AB = ab$. If we consult CPT(C) in Example 1.2, we see that it contains the preference rule $ab : c \succ \bar{c} \succ \bar{\bar{c}}$. Thus, $C = c$ is the best choice in this case and so has preference position 1. If we had $C = \bar{c}$, then we would have $k = 2$ and for $C = \bar{\bar{c}}$, we would have $k = 3$. Using this and the above results, the edge weight is

$$\frac{1}{4} \cdot (1 + 1) \cdot \frac{3 - 1 + 1}{3} = \frac{1}{4} \cdot (1 + 1) \cdot 1.$$

The other edge weights in Figure 2.1 are calculated similarly.

By examining the weighted event tree for this example, it can be seen that, for any two edges indicating the value taken by the same variable, the attached weights differ only on the preference position (the boldface number). Consider the set of edges leaving any node in the tree. By the definition of preference position, those edges indicating that the next variable takes a more preferred value will have larger weights. Thus, we can recover $T(N)$ given $W(N)$ (by recovering the edge labels by ordering them by weight). As $T(N)$ is an equivalent representation to N (Proposition 2.2), this shows that we can recover N from $W(N)$. Recall that N

is both the CP-net structure and the CPTs. By definition, $W(N)$ is constructed from N (after first constructing $T(N)$ from N). Thus, $W(N)$ is also an equivalent representation to N (and to $T(N)$).

For ease of notation we shall, from this point on, simplify the notation for the weighted event tree representation of N from $W(N)$ to W without ambiguity.

Now that we can construct the weighted event tree representation of any given CP-net, we use this structure to define our quantitative measure of preference for any outcome. The rank of an outcome will be the sum of the preference weights of the associated variable assignments.

Definition 2.5. Given a CP-net, N, and an associated outcome, o, we define the *rank* of o, $r(o)$, to be the sum of the weights on the edges of the root-to-leaf path of W that corresponds to o. This gives the following formula:

$$r(o) = \sum_{X \in V} AF_X(d_X + 1)P_P(X = o[X]|\mathrm{Pa}(X) = o[\mathrm{Pa}(X)]). \tag{2.4}$$

Example 2.6. Continuing on from Example 2.4, we calculate the ranks of several outcomes directly from W:

$$\begin{aligned} r(\bar{a}b\bar{\bar{c}}\bar{d}) =& \left[1 \cdot (2+1) \cdot \frac{1}{2}\right] + \left[1 \cdot (2+1) \cdot 1\right] + \left[\frac{1}{4} \cdot (1+1) \cdot 1\right] + \left[\frac{1}{12} \cdot (0+1) \cdot 1\right] \\ =& \frac{61}{12}, \end{aligned}$$

$$\begin{aligned} r(ab\bar{c}\bar{d}) =& \left[1 \cdot (2+1) \cdot 1\right] + \left[1 \cdot (2+1) \cdot 1\right] + \left[\frac{1}{4} \cdot (1+1) \cdot \frac{2}{3}\right] + \left[\frac{1}{12} \cdot (0+1) \cdot 1\right] \\ =& \frac{77}{12}, \end{aligned}$$

$$\begin{aligned} r(\bar{a}\bar{b}cd) =& \left[1 \cdot (2+1) \cdot \frac{1}{2}\right] + \left[1 \cdot (2+1) \cdot \frac{1}{2}\right] + \left[\frac{1}{4} \cdot (1+1) \cdot \frac{1}{3}\right] + \left[\frac{1}{12} \cdot (0+1) \cdot 1\right] \\ =& \frac{39}{12}. \end{aligned}$$

Recall that our aim was to assign higher values to the more preferred outcomes. Thus, the relative sizes of these ranks are as we would expect, as we can derive the following preference sequences directly from the CPTs of N:

$$ab\bar{c}\bar{d} \succ ab\bar{\bar{c}}\bar{d} \succ \bar{a}b\bar{\bar{c}}\bar{d},$$
$$\bar{a}b\bar{\bar{c}}\bar{d} \succ \bar{a}b\bar{\bar{c}}d \succ \bar{a}bcd \succ \bar{a}\bar{b}cd.$$

Thus, we have $N \vDash ab\bar{c}\bar{d} \succ \bar{a}b\bar{\bar{c}}\bar{d} \succ \bar{a}\bar{b}cd$ and $r(ab\bar{c}\bar{d}) > r(\bar{a}b\bar{\bar{c}}\bar{d}) > r(\bar{a}\bar{b}cd)$. We prove that this property holds in general in §2.3.3.

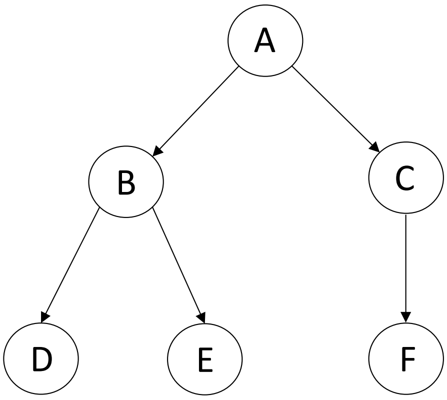


Figure 2.2: CP-Net Structure

Note that, although we have used W to calculate outcome ranks here, we show in §2.3.5 that ranks can be calculated efficiently without constructing W (in $O(n^4)$ time). This is reassuring as W is an exponentially large structure (in n).

The above formula for outcome ranks appears similar to the penalty functions defined by Domshlak et al. (2003) and Li et al. (2011a) (full details given in §2.2.1, Equations 2.1 and 2.2). However, our outcome ranks are meaningfully distinct from these penalty functions; they are not simply a transformation of one another. We prove, via the following example, that our outcome ranks give a distinct preference representation. In order to do so, we demonstrate that the consistent orderings induced by the penalty functions can be different to those induced by our outcome ranks. Thus, given a pair of outcomes, our ranks may assess their relative preference differently to the penalty functions.

Example 2.7. We now give an example of a CP-net where the penalty functions by Domshlak et al. (2003) and Li et al. (2011a) give (consistent) orderings over the outcomes that are distinct from our rank ordering. For this example, we refer to the penalty functions as pen_1 and pen_2, respectively.

Let N be a CP-net with the structure given in Figure 2.2. Let variable B have a domain of size five and let every other variable be binary. Let o_1 be the outcome associated with N where every variable takes its most preferred value except B, which takes the second most preferred value (out of five). Such an outcome can be constructed by assigning variables their most preferred (or second most preferred in the case of B) value in topological order. Further, o_1 is uniquely specified by this definition. Similarly, let o_2 be the outcome where every variable takes its most preferred value except C, which takes the second most preferred value (out of two).

We know $o_1 \neq o_2$ as they both assign the same value to A but C is its preferred position in o_1 and its not-preferred position in o_2 (thus, $o_1[C] \neq o_2[C]$).

By the definitions of pen_1 and pen_2, if a variable takes its most preferred value, then it has a local penalty of zero, $p(o, X) = 0$. If it is in the second most preferred position, it has a local penalty of 1. Thus, we can simplify the penalty definitions here:

$$\begin{aligned}
\text{pen}_1(o_1) &= \sum_{X \in V} w_X p(o_1, X) = \sum_{X \neq B} w_X \times 0 + w_B \times 1 = w_B, \\
\text{pen}_1(o_2) &= \sum_{X \in V} w_X p(o_2, X) = \sum_{X \neq C} w_X \times 0 + w_C \times 1 = w_C, \\
\text{pen}_2(o_1) &= \sum_{X \in V} w'_X p(o_1, X) = w'_B, \\
\text{pen}_2(o_2) &= \sum_{X \in V} w'_X p(o_2, X) = w'_C.
\end{aligned}$$

We therefore only need to calculate w_B, w_C, w'_B, and w'_C. By the recursive form of their definitions, this can be done by calculating w_X and w'_X from the leaves of the structure upwards. First, $w_D = w_E = w_F = 1$ and $w'_D = w'_E = w'_F = 1$ by definition as D, E, and F have no children. From these values, and the fact that D, E, and F are all binary, we can calculate w_B, w_C, w'_B, and w'_C:

$$\begin{aligned}
w_B =& \sum_{Y \in \text{Ch}(B)} w_Y |\text{Dom}(Y)| = w_D |\text{Dom}(D)| + w_E |\text{Dom}(E)| = 2 + 2 = 4, \\
w_C =& \sum_{Y \in \text{Ch}(C)} w_Y |\text{Dom}(Y)| = w_F |\text{Dom}(F)| = 2, \\
w'_B =& 1 + \sum_{Y \in \text{Ch}(B)} w'_Y (|\text{Dom}(Y)| - 1) \\
=& 1 + w'_D (|\text{Dom}(D)| - 1) + w'_E (|\text{Dom}(E)| - 1) = 1 + 1 + 1 = 3, \\
w'_C =& 1 + \sum_{Y \in \text{Ch}(C)} w'_Y (|\text{Dom}(Y)| - 1) = 1 + w'_F (|\text{Dom}(F)| - 1) = 1 + 1 = 2.
\end{aligned}$$

As $w_C < w_B$ and $w'_C < w'_B$, we have that $\text{pen}_1(o_2) < \text{pen}_1(o_1)$ and $\text{pen}_2(o_2) < \text{pen}_2(o_1)$. Thus, both penalty functions imply that $o_2 \succ o_1$. That is, both penalty orderings order o_2 above o_1.

If we calculate our outcome ranks using the formula from Definition 2.5, similarly to our previous rank calculations, we get $r(o_1) = 8.95$ and $r(o_2) = 8.75$. This means that outcome ranks order o_1 above o_2, the opposite of both penalty orders. Thus, ranks produce distinct consistent orderings to both penalty functions and, therefore, must be a distinct preference representation; it cannot be obtained by a simple transformation of either penalty function.

2.3.3 Consistently Ordering with Outcome Ranks

In this section, we demonstrate how our outcome ranks can be used to obtain consistent orderings. This can be applied to the whole outcome set, in order to get a complete consistent ordering for the CP-net, or to any subset of the outcomes.

As we constructed our outcome ranks to reflect user preference, they obey all entailed relations, as we wanted. Thus, our ranks induce a complete consistent ordering over the outcomes, $\succsim^R$. This $\succsim^R$ is obtained simply by ordering the outcomes according to their rank values, with with higher ranked outcomes considered to be more preferred. Proofs of these claims are given below.

Theorem 2.8. *Given a CP-net, N, for any outcomes o and o', we have that $N \models o \succ o' \implies r(o) > r(o')$.*

Proof. See Appendix E.2.

This result shows that, if the CP-net dictates that the user prefers o to o', then $r(o) > r(o')$, that is, $o \succ^R o'$. Thus, our outcome ranks have been shown to accurately reflect the user preferences encoded by the CP-net. In fact, we can say more than $r(o) > r(o')$; we can give a tight lower bound for the rank difference, $r(o) - r(o')$. Details of this lower bound are given in §2.4.1.

Corollary 2.9. *Given a CP-net, N, and two distinct associated outcomes, o and o', $r(o) = r(o') \implies N \models o \bowtie o'$. That is, o and o' are incomparable, $N \not\models o \succ o'$ and $N \not\models o' \succ o$.*

Proof. Theorem 2.8 shows that for any two outcomes, o_1 and o_2, $N \models o_1 \succ o_2 \implies r(o_1) > r(o_2)$, or equivalently $r(o_1) \leq r(o_2) \implies N \not\models o_1 \succ o_2$. Using this equivalent result gives us the following:

$$\begin{aligned} & r(o) = r(o') \\ \implies & (r(o) \leq r(o')) \wedge (r(o') \leq r(o)) \\ \implies & (N \not\models o \succ o') \wedge (N \not\models o' \succ o) \\ \implies & N \models o \bowtie o'. \end{aligned}$$

□

Corollary 2.10. *Let N be a CP-net. Let $\succsim^R$ be the complete ordering over the outcomes of N induced by the outcome ranks. Then $\succsim^R$ is a consistent ordering of the outcomes with respect to N.*

Proof. In order to show that $\succsim^R$ is a consistent ordering, we need to show that, for any two outcomes o_1 and o_2, $N \vDash o_1 \succ o_2 \implies o_1 \succ^R o_2$. Theorem 2.8 shows that $N \vDash o_1 \succ o_2 \implies r(o_1) > r(o_2)$. By definition of $\succsim^R$, $r(o_1) > r(o_2) \implies o_1 \succ^R o_2$. Thus, we have $N \vDash o_1 \succ o_2 \implies o_1 \succ^R o_2$ and so we can conclude that $\succsim^R$ is a consistent ordering of the outcomes. □

We cannot guarantee that $\succsim^R$ is a strict order. There is a possibility that two distinct outcomes, o and o', could be assigned equal rank. However, Corollary 2.9 shows that this can only occur when we do not know which outcome the user prefers. If we want a strict ordering of the outcomes, then it is enough to force any outcomes with equal ranks into an arbitrary order. Any strict ordering of the outcomes obtained from $\succsim^R$ in this manner is a consistent ordering of the outcomes as we have only altered the order of incomparable outcomes.

We have now introduced a novel method of quantifying user preference and obtaining a consistent outcome ordering given any (possibly multivalued) acyclic CP-net. Further, we can ensure that this is a strict ordering of the outcomes. From now on, when we refer to the outcome ordering induced by ranks, we are referring to a strict ordering, $\succ^R$.

Example 2.11. For the CP-net given in Example 1.2, the ordering of the outcomes induced by ranks is as follows:

$$abcd \succ^R abc\bar{d} \succ^R ab\bar{c}\bar{d} \succ^R ab\bar{c}d \succ^R ab\bar{\bar{c}}\bar{d} \succ^R ab\bar{\bar{c}}d \succ^R a\bar{b}\bar{c}\bar{d} \sim^R \bar{a}b\bar{\bar{c}}\bar{d} \succ^R$$
$$a\bar{b}\bar{c}d \sim^R \bar{a}b\bar{\bar{c}}d \succ^R a\bar{b}\bar{\bar{c}}\bar{d} \sim^R \bar{a}bcd \succ^R a\bar{b}\bar{\bar{c}}d \sim^R \bar{a}bc\bar{d} \succ^R a\bar{b}cd \sim^R \bar{a}b\bar{c}\bar{d} \succ^R$$
$$a\bar{b}c\bar{d} \sim^R \bar{a}b\bar{c}d \succ^R \bar{a}\bar{b}\bar{\bar{c}}\bar{d} \succ^R \bar{a}\bar{b}\bar{\bar{c}}d \succ^R \bar{a}\bar{b}\bar{c}\bar{d} \succ^R \bar{a}\bar{b}\bar{c}d \succ^R \bar{a}\bar{b}cd \succ^R \bar{a}\bar{b}c\bar{d}.$$

We can obtain a strict ordering of the outcomes simply by replacing each $\sim^R$ with a $\succ^R$.

Remark. Going from a CP-net to a consistent ordering gives the impression of losing a great deal of information, especially as there are likely to be many consistent orderings and we have constructed one that is no better than any other. Moreover, the process of forcing our ordering to be strict arbitrarily discards several possible orderings. However, we have found that, given this consistent ordering (or the outcome rank values), we can answer ordering and dominance queries directly, without needing to consult the CP-net. Further, we can use outcome ranks to improve the efficiency of answering dominance queries. We can therefore determine whether $o \succ^R o'$ is entailed by the CP-net ($N \vDash o \succ o'$) or constructed ($N \vDash o \bowtie o'$) from $\succ^R$ directly (note that $N \vDash o' \succ o$ is not possible as $\succ^R$ is consistent, thus, answering the dominance query '$N \vDash o \succ o'$?' is sufficient). These results are all discussed in §2.4.1. It is also possible to update $\succ^R$ given new (consistent) preference information, without consulting the CP-net, this is shown in Appendix A. In fact, despite constructing a consistent ordering somewhat arbitrarily, we have not lost any information at all, as we shall prove below. These results are not specific to $\succ^R$, rather we show that they hold for any consistent ordering.

Theorem 2.12. *Let N be a CP-net and $\succsim^C$ be any consistent ordering over the outcomes. Then no information is lost by reducing N to $\succsim^C$. That is, $\succsim^C$ encodes all of the preference information given by N.*

Proof. See Appendix E.3.

Corollary 2.13. *Let N be a CP-net. Reducing N to outcome ranks, or the associated induced ordering, $\succ^R$, loses no information.*

Proof. By Theorem 2.12, reducing N to a consistent ordering loses no information. The ordering induced by outcome ranks, $\succ^R$, is consistent by Corollary 2.10. Thus, reducing N to this ordering loses no information. The outcome ranks induce this ordering. Thus, $\succ^R$ can be obtained from the outcome ranks and so, using Theorem 2.12, N can be recovered from the outcome ranks alone. Hence, reducing N to outcome ranks loses no information. □

Our method of obtaining a consistent ordering using outcome ranks has the advantage of how easily it can be adapted to find a consistent ordering of any subset of the outcomes. Let N be a CP-net over variables, V, and let S be some subset of the outcomes, $S \subseteq \Omega$. Suppose we wish to put these outcomes, S, in an order that agrees with everything the CP-net tells us about the user's preference. That is, we wish to find a strict order over S, $\succ^S$, such that for any two outcomes $o_1, o_2 \in S$, we have that $N \vDash o_1 \succ o_2 \implies o_1 \succ^S o_2$. To motivate the consistent ordering of subsets, consider an online shopping website displaying its products and suppose the seller wishes to promote a certain range of items; the seller wants exactly these items to appear on the first page. However, they still want this range of outcomes to appear in an order such that those items of more interest to the client are higher up. Thus, a consistent ordering of this specified range of products is required.

A consistent ordering of $S \subseteq \Omega$ can be obtained in exactly the same way that we obtained a consistent ordering for N. For each $o \in S$, calculate the rank of o, $r(o)$, and then order S according to rank value. To get a strict consistent ordering of S, force outcomes of equal rank into an arbitrary order. We call this strict ordering of S, $\succ^S$. We can see that $\succ^S$ is a consistent ordering of S by using exactly the same reasoning we used to show that $\succ^R$ is a consistent ordering. In principle, we could instead obtain $\succ^S$ by constructing $\succ^R$ and then restricting the ordering to S (as can be done for any consistent ordering, not just $\succ^R$); however, this is unnecessary in practice, as the above method is more efficient.

In §2.3.5, we present an algorithm that can calculate $r(o)$ for any outcome in time $O(n^4)$. Thus, a consistent ordering for a subset of size k can be obtained, as described above, in $O(n^4k + k^2)$ time. As we discussed in §2.2.2, Boutilier et al. (2004a) also proposed a solution to the problem of obtaining a consistent ordering for any subset of the outcomes. They proposed finding a consistent ordering of S by repeatedly answering ordering queries. Using this method, a consistent ordering for a subset of size k can be obtained in $O(nk^2)$ time. Thus, for larger subsets of the outcomes, our method becomes more efficient. This is because every ordering query has complexity $O(n)$, whereas, in our method, once the ranks are calculated the problem is reduced to a simple sorting task. Note that the total number of outcomes is at least 2^n (with equality only in the case of binary CP-nets), so subsets of the outcomes can be very large even for relatively small CP-nets.

We have now introduced a novel quantification of user preference, given a CP-net representation of preference. We have shown that these ranks successfully reflect all entailed relations and how they can be used to obtain a consistent ordering of the outcomes. Further, we have shown that this method can be directly applied to obtain a consistent ordering of any subset of the outcomes.

2.3.4 Consistently Ordering Under Plausibility Constraints

A particularly interesting application of consistently ordering subsets of the outcomes is finding a consistent ordering for CP-nets that have additional plausibility constraints. That is, a CP-net where only a specified proper subset of the outcomes, say $P \subsetneq \Omega$, are possible and the remainder are considered impossible. In real world problems, this kind of asymmetry in a CP-net system is commonplace. Consider, for example, an airline where there are no flights between specified dates and destinations with available business class seats, such tickets would then be impossible outcomes.

Proposition 2.14. *Given a CP-net, N, and the further constraint that the only outcomes that are possible are those contained in $P \subsetneq \Omega$, let N_C denote the CP-net with these additional constraints. Let $\succsim^P$ be any total preorder over P such that, for all $o, o' \in P$, we have $N \models o \succ o' \implies o \succ^P o'$. Then $\succsim^P$ is a consistent ordering for N_C.*

Proof. In order to show that $\succsim^P$ is a consistent ordering for N_C, it is enough to show that $N_C \vDash o \succ o' \implies o \succ^P o'$. We know that $N \vDash o \succ o' \implies o \succ^P o'$ holds, so it is sufficient to prove that $N_C \vDash o \succ o' \implies N \vDash o \succ o'$. Recall that a CP-net entails the relation $o \succ o'$ if and only if there is a path $o' \rightsquigarrow o$ in the preference graph. Let G_N be the preference graph for N and let G_{N_C} be the preference graph for N_C. Then G_{N_C} is the induced subgraph of G_N on outcomes (nodes) P. Thus, if there exists a $o' \rightsquigarrow o$ path in G_{N_C}, then this will be a path (improving flipping sequence) in G_N that exclusively uses outcomes in P. Therefore, there is a path $o' \rightsquigarrow o$ in G_N and so we have that $N_C \vDash o \succ o' \implies N \vDash o \succ o'$. □

By Proposition 2.14, every consistent ordering (with respect to N) of the subset $P \subsetneq \Omega$ is a consistent ordering for N_C. Thus, being able to obtain a consistent ordering of any subset of outcomes for a CP-net, N, means that we can also obtain a consistent ordering for any constrained CP-net, N_C.

In the case of CP-nets with additional plausibility constraints, any consistent ordering restricted to P will be a consistent ordering for N_C. To obtain a consistent ordering of P using outcome ranks you do not have to construct the full consistent ordering. In fact, you only need to calculate the edge weights of W for edges that are on root-to-leaf paths corresponding to some $o \in P$. Depending on the severity of the plausibility constraints, this could cut down calculations significantly. For larger possibility sets, P, this would also be more efficient than using the method by Boutilier et al. (2004a) for ordering outcome subsets (by the same reasoning as §2.3.3).

Example 2.15. Consider the CP-net given in Example 1.2 with the following constraints.

$$C = \{\neg\bar{a}, \neg(b \wedge c), \neg(\bar{b} \wedge \bar{c}), \neg(\bar{b} \wedge \bar{\bar{c}} \wedge \bar{d})\}.$$

In order to construct a consistent ordering for N_C, we only need to consider the restricted W seen in Figure 2.3 (edge weights are calculated exactly the same way as in Figure 2.1).

From this much smaller tree, we calculate ranks as usual and order the possible outcomes (P) by their rank:

$$ab\bar{c}\bar{d} \succ^P ab\bar{c}d \succ^P ab\bar{\bar{c}}\bar{d} \succ^P ab\bar{\bar{c}}d \succ^P a\bar{b}\bar{\bar{c}}d \succ^P a\bar{b}cd \succ^P a\bar{b}c\bar{d}.$$

This is a consistent ordering of P for N_C. It can be seen by comparing $\succ^P$ to $\succ^R$ (given in Example 2.11), that $\succ^P$ is the restriction of $\succ^R$ to P.

Remark. Constrained CP-nets can be considered from two perspectives, which determine the preference structure they encode. Suppose we have a CP-net, N

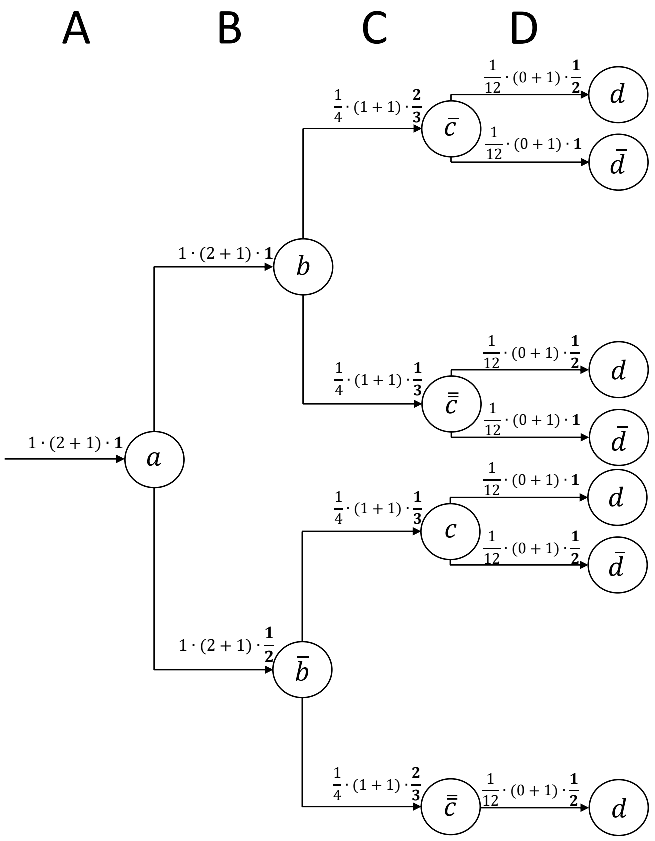

A
B
C
D
$1 \cdot (2+1) \cdot \mathbf{1}$
a
$1 \cdot (2+1) \cdot \mathbf{1}$
b
$1 \cdot (2+1) \cdot \mathbf{\frac{1}{2}}$
$\bar{b}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{2}{3}}$
$\bar{c}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{1}{3}}$
$\bar{\bar{c}}$
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{1}{3}}$
c
$\frac{1}{4} \cdot (1+1) \cdot \mathbf{\frac{2}{3}}$
$\bar{\bar{c}}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
$\bar{d}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
$\bar{d}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{1}$
d
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
$\bar{d}$
$\frac{1}{12} \cdot (0+1) \cdot \mathbf{\frac{1}{2}}$
d

Figure 2.3: Weighted Event Tree for a Constrained CP-Net

and some associated plausibility constraints, C, which result in a constrained CP-net, N_C. Let P denote the plausible outcomes specified by C. Suppose the user knew about these plausibility constraints, C, when expressing their preferences. In this case, we cannot assume that these preferences apply to outcomes not in P. Returning to our flight seat example, suppose we told the user that Wi-Fi is not available on short flights unless they are seated in first class. This may then affect their specified preferences over variable C (the class of their seat). Suppose, in this case, the specified preferences (encoded by N) imply an IFS $o_1 \rightsquigarrow o_2 \rightsquigarrow o_3$ where $o_1, o_3 \in P$ and $o_2 \notin P$. The user did not consider o_2 when specifying their preferences and so we cannot consider this to be a valid proof of $o_3 \succ o_1$. Thus, $N_C \vDash o \succ o'$ (for $o, o' \in P$) only if there is an IFS in N that consists only of outcomes in P. This makes the preference graph of N_C the induced subgraph of G_N on nodes P, which is what we used in the proof of Proposition 2.14.

Alternatively, if the user was unaware of the constraints, then all IFS for N constitute valid preferences. Thus, $N_C \vDash o \succ o'$ $(o, o' \in P)$ if and only if $N \vDash \ o \succ \ o'$. Note that this alternative interpretation does not affect our results as we still have $N_C \vDash o \succ o' \implies N \vDash o \succ o'$, as required to prove Proposition 2.14.

If the user knew some but not all of the constraints, then the preference graph of N_C would be somewhere between the above two options. The former is the most conservative in its assumptions regarding the preferences implied under constraints. Thus, assuming that the user is aware of any constraints could be considered the 'safer' option if it is unclear from context.

2.3.5 Rank Calculation Algorithm

The outcome ranks defined in §2.3.2 (Definition 2.5) are time consuming to calculate by hand even for fairly small CP-net examples. In this section, we present an algorithm for calculating the rank of any outcome. In §2.3.2, we used the event tree representation of CP-nets in both constructing our rank definition and in calculating example ranks. However, in this section, we show that ranks can be calculated directly from a CP-net input. Further, we can calculate the rank of any outcome in $O(n^4)$ time.

Algorithm 1 takes a CP-net and an outcome as inputs and outputs the rank of the given outcome. Recall, the rank of an outcome, o, is the sum of the weights on the root-to-leaf path of W corresponding to o. Algorithm 1 calls two other algorithms. Algorithm 6 takes a variable, X, and outputs the set of its ancestors in the CP-net structure, $\text{Anc}(X) = \{Y \mid \exists \text{ a directed path } Y \rightsquigarrow X \text{ in } N\}$. Algorithm 7 takes a variable, X, and calculates the number of descendent paths of X in the CP-net structure, d_X. Algorithms 6 and 7 are given in Appendix B.3.

For the remainder of this section, suppose we have a CP-net, N, over a set of variables, $V = \{X_1, ..., X_n\}$, that are in a topological order with respect to the structure of N. We assume that N is input to Algorithm 1 as a pair, $N = (A, CPT)$, where A is the *adjacency matrix* for the structure of N. That is, A is an $n \times n$ matrix such that $A_{i,j} = 1$ if there is an edge $X_i \rightarrow X_j$ in the structure of N, and $A_{i,j} = 0$ otherwise. The second entry in the CP-net pair, CPT, is the set of CPTs associated with N. We assume this to be input in a particular format, which is given in Appendix B.1 with an illustrative example. From this CPT input, we can extract $|\text{Dom}(X_i)|$ for any $1 \leq i \leq n$. To keep Algorithm 1 as readable as possible, we assume that, given i, we can obtain $|\text{Dom}(X_i)|$, rather than writing the details of how this is achieved (these details are given in Appendix B.1). We also leave the details of the format for input outcomes to Appendix B.1.

Algorithm 1 takes the CP-net, N, and some outcome, o, and outputs the rank of this outcome, $r(o)$. It calculates $r(o)$ by setting the value of $r(o)$ to 0 (step **1**) and

Algorithm 1: Outcome Rank Calculation

```
Input  : $N = (A, CPT)$ – CP-net
         $o$ – Outcome
Output: $r(o)$ – Outcome rank
// Variables and array elements are indexed from 1 (rather
   than 0) in the following pseudocode
$r(o) = 0$;
for $X_i \in V = \{X_1, X_2, ..., X_n\}$ do
    $Anc = \mathrm{Anc}(X_i)$;                    // Calculated using Algorithm 6
    $AF = \prod_{Y \in Anc} \frac{1}{|\mathrm{Dom}(Y)|}$;
    $d = d_{X_i}$;                               // Calculated using Algorithm 7
    $Pa = \{j \mid A_{j,i} = 1\}$;                          // Parent set of $X_i$
    $\mathbf{u} = o[Pa]$;                   // Values taken by $Pa$ in outcome $o$
    $\mathbf{order} = CPT[i][\mathbf{u}]$;  // Preference order over $X_i$, given $Pa = \mathrm{u}$
    $k = \mathbf{order}[o[i]]$;                                  // $o[i] = o[X_i]$
    // $k$ - position of preference of $o[i]$ in the previous order
    $P_P = \frac{|\mathrm{Dom}(X_i)| - k + 1}{|\mathrm{Dom}(X_i)|}$;
    $r(o) = r(o) + AF \cdot (d + 1) \cdot P_P$;
end
return $r(o)$;
```

successively adding the edge weights of the root-to-leaf path in W that corresponds to o (steps **2-12**). A more detailed explanation of how Algorithm 1 works and why it is correct can be found in Appendix B.2.

We have used W here and in Appendix B.2 to help explain what Algorithm 1 is doing and to show why it is correct. However, notice that the algorithm itself does not utilise W at any point and instead works directly with the CP-net to obtain the rank. This shows that, whilst the event tree representation was useful in motivating and explaining our ranking system, constructing the tree is not a necessary step in calculating the rankings. This is reassuring as W has exponential size (in n) and so quickly becomes large even for relatively small CP-nets.

For a CP-net, N, with n variables, Algorithms 6 and 7 both have complexity $O(n^3)$ and Algorithm 1 has complexity $O(n^4)$. Thus, for any associated outcome, o, we can compute $r(o)$ in $O(n^4)$ time; that is, finding the rank of an outcome is tractable. Similarly, the penalty values by Domshlak et al. (2003) and Li et al. (2011a) can be calculated in time polynomial in n. However, the utilities presented by McGeachie and Doyle (2004) can be intractable to calculate.

Remark. We could use Algorithm 1 to produce a consistent ordering, given a CP-net, N, as shown by Corollary 2.10. This is done by using Algorithm 1 to calculate the rank of each outcome, and then sorting these outcomes into rank-order. However, to obtain a consistent ordering in this manner, we are applying Algorithm 1 $|\Omega|$ many times, making the time complexity in terms of $|\Omega|$. As $|\Omega| \geq 2^n$ (with equality only in the case of binary CP-nets), this is not a tractable method. This is unsurprising as putting $|\Omega|$ objects into an order will always have time complexity in terms of $|\Omega|$ (intractable). Our primary aim is to use these ranks (algorithms) to improve the efficiency of dominance testing, which, as shown in §2.4.1, does not require a consistent ordering. Thus, we are not concerned by this lack of tractability.

2.4 Efficient Dominance Testing

In this section, we show how our outcome ranks can be used to improve dominance testing efficiency and compare this to the existing methods for improving dominance testing efficiency. In §2.4.1, we explain how rank pruning for efficient dominance testing works. In §2.4.2, we provide an experimental comparison to the existing methods for improving dominance testing efficiency via search tree pruning. These results show that rank pruning significantly outperforms the existing methods. We also evaluate all possible combinations of methods and find that rank pruning is a critical component for a successful combination.

2.4.1 Rank Pruning for Dominance Testing

As we explained in §2.1, answering dominance queries is an important task (see §1.3 for formal dominance query definition). We must be able to answer these queries efficiently for CP-nets to be a practical representation of user preference. However, dominance queries are complex to answer. If $N \models o \succ o'$, then the user prefers o to o' and so o comes before o' in all consistent orderings. Thus, dominance queries require us to consider all consistent orderings, whereas previously we have been concerned only with finding an arbitrary consistent ordering. To answer the query '$N \models o \succ o'$?', we must prove either that o comes before o' in every consistent ordering or, alternatively, prove that there exists a consistent ordering where o' comes before o. Unless one is lucky enough to construct a consistent ordering where o' comes before o, this cannot be answered by considering a single arbitrary consistent ordering.

Dominance queries have been proven to be complex tasks to answer in general. Dominance testing (answering dominance queries) for binary CP-nets has the following complexities:

- $O(n)$ when the CP-net has a tree structure (Bigot et al., 2013).
- Polynomial in n when the CP-net structure is a polytree (has no cycles when orientation is removed) (Boutilier et al., 2004a).
- NP-complete when any two variables are connected by at most one directed path in the CP-net structure (Boutilier et al., 2004a).
- NP-complete when any two variables are connected by at most δ directed paths in the CP-net structure and δ is polynomially bounded (Boutilier et al., 2004a).
- PSPACE-complete for CP-nets in general (and if we restrict to consistent CP-nets) (Goldsmith et al., 2008).

As multivalued CP-nets are a strict generalisation of binary CP-nets, dominance testing for multivalued CP-nets must have complexities at least as hard as the above. As we discussed in §2.2.3, there are several existing methods for making dominance testing more efficient. In this section, we introduce a novel method of improving dominance testing efficiency (for acyclic, multivalued CP-nets) using our outcome ranks.

The standard method of answering dominance queries is by searching for an improving flipping sequence, which can be visualised as constructing a search tree.

Before we explain how outcome ranks can improve dominance testing efficiency, we must formalise these notions.

Suppose we have the dominance query '$N \models o \succ o'$?'. This is true if and only if there is a path $o' \rightsquigarrow o$ in the preference graph of N, G_N (Boutilier et al., 2004a). This directed path corresponds to an improving flipping sequence (IFS) of outcomes – $o' = o_1, o_2, ..., o_m = o$, such that o_i and o_{i+1} differ on the value of exactly one variable and $N \models o_{i+1} \succ o_i$. Therefore, a dominance query can be reframed as a search for an IFS in G_N. Thus, the query becomes 'is o reachable from o' in G_N?'. This question can be answered by searching through G_N from node o', to see which outcomes are reachable. We formalise this as the process of constructing a search tree, $G(o')$. Note that, by performing the search in the following manner, we only construct the necessary section of G_N.

Given the dominance query $N \models o \succ o'$, we want to determine whether o is reachable from o' in G_N. We do this by constructing the *dominance query search tree*, $G(o')$, until either o is reached (and so the dominance query is true) or it cannot be constructed further (and so the dominance query is false). This search tree is constructed as follows. Start with o' at the root of the tree and then repeat the following procedure. Select some leaf $o_\ell \in G(o')$ that has not previously been considered and, for every improving flip, o_ℓ^*, of o_ℓ that is not already in $G(o')$, add the edge $o_\ell \rightarrow o_\ell^*$ to the tree. We now say that o_ℓ has been considered. This step is repeated until either o is reached (the dominance query is true) or all leaves in $G(o')$ have been considered previously (the dominance query is false). Boutilier et al. (2004a) first demonstrated how dominance queries can be answered via the construction of a search tree. However, they allowed duplicate outcomes in their search trees and so, in general, their proposed search trees are larger than ours.

The above method correctly answers the dominance query because, when $G(o')$ is fully constructed, the outcomes contained in $G(o')$ are exactly those outcomes reachable from o' in G_N. This is because, for any outcomes o_1 and o_2, G_N contains the edge $o_1 \rightarrow o_2$ if and only if o_2 is an improving flip of o_1 (by the definition of a preference graph). Thus, constructing $G(o')$ is equivalent to exploring all paths in G_N originating at o' until either o is found or no new outcomes can be found. Note that the fact we do not include outcome duplicates does not affect these results. There are a fixed set of outcomes in G_N reachable from any given outcome, o^*. If we reach o^* from o' in two different ways, there is no reason to include both instances of o^* as they will both lead to the same set of outcomes. Thus, as we are only in interested in whether $o \in G(o')$ or not (not in finding all possible paths from o' to o), there is no need to include duplicates and it is more efficient to exclude them.

Note that the improving flips of an outcome can be easily found by consulting the CPTs of N. Every improving flip of outcome o is obtained by improving exactly one variable. Suppose we have $X \in V$. We can find the possible improving X flips of o by consulting the $o[\mathrm{Pa}(X)]$ row of $\mathrm{CPT}(X)$ to find which X values would be an improvement on $o[X]$ (if any).

As we discussed in §2.2.3, several existing works improve dominance testing efficiency by pruning the search tree, $G(o')$ (Allen et al., 2017a; Boutilier et al., 2004a; Li et al., 2011a; Wilson, 2004b). In this section, we show how our outcome ranks can be used to prune $G(o')$ in a new way, in order to improve dominance testing efficiency.

In §2.3.2, we constructed an outcome rank that reflects all entailed relations, that is, $N \vDash o_1 \succ o_2 \implies r(o_1) > r(o_2)$. However, this is not all we can say about the difference between the ranks of o_1 and o_2; we can also identify a tight lower bound on the rank difference, $r(o_1) - r(o_2)$, as we show below.

Definition 2.16. Let N be a CP-net over variables V. For any $X \in V$, we define the *least rank improvement* of X, denoted $L(X)$, as

$$L(X) = AF_X(d_X + 1)\frac{1}{n_X} - \sum_{Y \in \mathrm{Ch}(X)} AF_Y(d_Y + 1)\frac{n_Y - 1}{n_Y},$$

where, for any $X \in V$, $n_X = |\mathrm{Dom}(X)|$ and $\mathrm{Ch}(X) = \{Y \in V \mid X \in \mathrm{Pa}(Y)\}$. We call $\mathrm{Ch}(X)$ the children of X.

This value, $L(X)$, is interpreted as a lower bound on the increase in rank resulting from flipping X to a more preferred value. That is, $L(X)$ corresponds to the rank increase of the improving X flip $\alpha \to \beta$, $L(X) = r(\beta) - r(\alpha)$, where X only improves by one preference position and every $Y \in \mathrm{Ch}(X)$ goes from being the most preferred value to the least preferred value[1]. Note that for all other variables, Z, the values taken by Z and $\mathrm{Pa}(Z)$ must be identical in α and β. Therefore, the preference position of Z must be identical in α and β. As β must be preferred to α (as it is an improving flip), we would expect $L(X)$ to be a strictly positive value. This is proven by the following lemma.

Lemma 2.17. *Let N be a CP-net over variables V. For any $X \in V$, $L(X) > 0$.*

Proof. See Appendix E.4.

[1] Note that such outcomes, α and β, may not always exist. We constructed $L(X)$ based on the 'worst' possible improving X flip, this does not occur in every CP-net for every variable

Least rank improvement terms can be used to define a tight lower bound on the difference in rank implied by entailment. That is, given $N \vDash o_1 \succ o_2$, Theorem 2.8 tells us that $r(o_1) > r(o_2)$, but we can use $L(X)$ terms to define a tight, positive lower bound for $r(o_1) - r(o_2)$.

Corollary 2.18. *Let N be a CP-net over variables V. Let o_1 and o_2 be associated outcomes and $D = \{X \in V \mid o_1[X] \neq o_2[X]\}$. Then,*

$$N \vDash o_1 \succ o_2 \implies r(o_1) - r(o_2) \geq \sum_{X \in D} L(X) > 0.$$

This is a tight lower bound on the rank difference implied by entailment.

Proof. See Appendix E.5.

Definition 2.19. Let N be a CP-net over variables V, and let o_1 and o_2 be associated outcomes. Let $D = \{X \in V \mid o_1[X] \neq o_2[X]\}$. The *least (entailed) rank difference* between o_1 and o_2, denoted $L_D(o_1, o_2)$, is defined as follows:

$$L_D(o_1, o_2) = \sum_{X \in D} L(X).$$

We now illustrate how Corollary 2.18 can be used to improve dominance testing efficiency. Suppose we have a CP-net N, and we wish to answer the dominance query '$N \vDash o \succ o'$?'. There are three possibilities, either $N \vDash o \succ o'$, $N \vDash o' \succ o$, or $N \nvDash o \succ o' \wedge N \nvDash o' \succ o$. In the latter case, we say o and o' are incomparable and denote this by $N \vDash o \bowtie o'$. We can get at least halfway to answering our dominance query by calculating the ranks of o and o' and their least rank difference. As shown in Corollary 2.18, if $r(o') + L_D(o, o') > r(o)$, then $N \nvDash o \succ o'$ and the dominance query is false. If $r(o) \geq r(o') + L_D(o, o')$, then, by Theorem 2.8 and Lemma 2.17, $N \nvDash o' \succ o$ and so it remains to determine whether $N \vDash o \succ o'$ or $N \vDash o \bowtie o'$. To answer this we would then construct the search tree, $G(o')$.

Remark. Note that getting 'halfway' to answering our dominance query is equivalent to answering an ordering query. An ordering query asks for a consistent ordering of a given pair outcomes, o and o' (we assume $o \neq o'$). We can answer ordering queries directly from outcome ranks or, as above, using L_D terms also. If $r(o) > r(o')$ then we know, by Theorem 2.8, that $N \nvDash o' \succ o$. Thus, $o \succ o'$ is a consistent ordering. If $r(o) = r(o')$, then, by Corollary 2.9, $N \vDash o \bowtie o'$. Thus, we have answered both associated dominance queries and we know that both $o \succ o'$ and $o' \succ o$ (and $o' \sim o$) are consistent orderings. As we can calculate rank values in $O(n^4)$ time (see §2.3.5), this means we can also answer ordering queries in $O(n^4)$ time. This test can also be performed directly from the rank induced consistent ordering (see §2.3.3), $\succsim^R$, rather than from rank values.

Alternatively, we can also use the L_D terms as we have above. If $r(o) \geq r(o') + L_D(o,o')$, then $r(o) > r(o')$ by Lemma 2.17 and so, as above, $o \succ o'$ is consistent. If $r(o) < r(o') + L_D(o,o')$, then $N \not\models o \succ o'$ by Corollary 2.18. Thus, $o' \succ o$ is a consistent ordering. As we show later in this section, calculating L_D also takes $O(n^4)$ time. Thus, this method of answering ordering queries also takes $O(n^4)$ time. It is worth performing this test in both direction as it can yield more information. Suppose $r(o) < r(o') + L_D(o,o')$ and we conclude that $o' \succ o$ is a consistent ordering. If we then find $r(o') < r(o) + L_D(o,o')$, we can conclude similarly that $o \succ o'$ is also a consistent ordering. Thus, $N \models o \bowtie o'$ and we have now answered both of the associated dominance queries. Note that performing this test in both directions again has complexity $O(n^4)$. In Appendix C.2, we discuss how often this test is sufficient for answering dominance queries. Further, we show that this test can be used to predict dominance queries with reasonable accuracy.

In general, more information can also be gained by using more than one test to answer ordering queries. If one method results in the ordering $o \succ o'$ and another in $o' \succ o$, then $N \models o \bowtie o'$. Thus, by using both tests, we have gained more information than either could supply individually.

Boutilier et al. (2004a) give a method of answering ordering queries in linear time, $O(n)$. Thus, the above tests are not the most efficient methods unless the rank values (and L_D terms) are already known – in this case they become constant time methods. However, as we discussed above, it can be more informative to answer ordering queries in multiple distinct ways. Thus, these additional tests may still be of use despite not being the most efficient choices.

As we mentioned above, the first rank test can be performed using $\succsim^R$ rather than rank values. In fact, given any consistent ordering, $\succsim^C$, the same test can be used to answer ordering queries directly in constant time. Given any pair of outcomes, we must have either $o \succ^C o'$ or $o \sim^C o'$. If $o \succ^C o'$, then clearly $o \succ o'$ is a consistent order. If $o \sim^C o'$, then we must have $N \models o \bowtie o'$, as $N \models o_1 \succ o_2$ implies $o_1 \succ^C o_2$ (as $\succsim^C$ consistent). Thus, both associated dominance queries are answered and both $o \succ o'$ and $o' \succ o$ (and $o' \sim o$) are consistent orderings. As CP-nets usually have multiple consistent orderings, this general method provides several distinct tests for answering ordering queries. As we mentioned above, these can be combined with each other, or with other methods of answering ordering queries, in order to yield more information.

We propose that our outcome ranks can be used to improve dominance testing efficiency by imposing an upper bound on the rank values of outcomes in $G(o')$. This will allow us to prune the tree as it is constructed and, thus, improve the efficiency of constructing $G(o')$ and answering the query. Ideally, our pruning technique would be implemented alongside other existing methods of improving search efficiency. In §2.4.2, we provide an experimental evaluation of the performance of our rank pruning in comparison to existing methods as well as all possible combinations of methods. However, here we illustrate how our pruning works with the basic search method only.

Returning to the dominance query, $N \vDash o \succ o'$?, suppose we have already confirmed that $r(o) \geq r(o') + L_D(o, o')$. We can answer this dominance query by determining whether or not there exists an IFS from o' to o. Note that if $o' = o_1, o_2, ..., o_m = o$ is such an IFS, then we must have $N \vDash o_{i+1} \succ o_i$ for all i. Thus, by transitive closure, $N \vDash o \succ o_i$ for all $1 \leq i < m$. Corollary 2.18 dictates that, if $N \vDash o \succ o_i$, then the rank difference between o and o_i must be at least $L_D(o, o_i)$. Therefore, o_i must satisfy $r(o) \geq r(o_i) + L_D(o, o_i)$ for all $1 \leq i < m$; this enforces an upper bound on the rank values of $G(o')$ – we only need to consider outcomes o^* with rank at most $r(o) - L_D(o, o^*)$. We determine whether such an IFS exists by constructing (and pruning) $G(o')$ as follows:

For any outcome o^*, define $F(o^*) = \{o \in \Omega \mid o^* \to o \text{ is an improving flip}\}$. That is, $F(o^*)$ is the set of outcomes, o, that differ from o^* on exactly one variable and such that $N \vDash o \succ o^*$. This set can be evaluated by inspecting the CPTs of N, as discussed above. We start constructing $G(o')$ by setting o' as the root node. As o' is the only leaf node, we then add an edge from o' to all improving flips of o', $F(o')$. If $o \in F(o')$, then clearly there is an $o' \rightsquigarrow o$ IFS (of length one) and the answer to the dominance query is yes, $N \vDash o \succ o'$. If $o \notin F(o')$, then we cannot reach o from o' in one improving flip. The next step is to add the improving flips of the leaf nodes in order to determine whether o can be reached from o' in two improving flips. However, before looking at all outcomes that can be reached from $F(o')$ by improving flips, there may be some search directions that can already be dismissed using our upper bound on rank values for $G(o')$. For each $o^* \in F(o')$, evaluate $r(o^*) + L_D(o, o^*)$. Any outcome, o^*, such that $r(o^*) + L_D(o, o^*) > r(o)$ is not on an $o' \rightsquigarrow o$ IFS by the above argument. Therefore, it is unnecessary to evaluate which outcomes can be reached by improving flips from o^*. That is, we know that any paths originating at o^* will not contain o and so we do not need to explore in this direction. Any nodes (outcomes), $o^* \in F(o')$, such that $r(o^*) + L_D(o, o^*) > r(o)$ are pruned from $G(o')$.

We repeat this process at each new leaf node that we consider in the construction of $G(o')$. Let o_ℓ be any (not pruned) leaf node of $G(o')$ that has not been considered previously. We first evaluate $F(o_\ell)$. If $o \in F(o_\ell)$, then our dominance query is true. As $G(o')$ is constructed by starting at o' and adding improving flips, $o_\ell \in G(o')$ implies that there is an IFS $o' \rightsquigarrow o_\ell$. Thus, if $o \in F(o_\ell)$ (that is, o

is an improving flip of o_ℓ), there is an IFS $o' \rightsquigarrow o$ and so our dominance query is true ($N \vDash o \succ o'$). In particular, if o_ℓ is at depth i in the tree, there is an IFS of length $i+1$ from $o' \rightsquigarrow o$.

If o is not in $F(o_\ell)$, then we add the improving flips that are not already in $G(o')$. Let $F(o_\ell)\backslash G(o')$ denote the outcomes in $F(o_\ell)$ that are not in $G(o')$ already. Thus, we add the edge $o_\ell \to o_\ell^*$ to $G(o')$ for every $o_\ell^* \in F(o_\ell)\backslash G(o')$. However, as before, if $o_\ell^* \in F(o_\ell)\backslash G(o')$ has outcome rank such that $r(o_\ell^*) + L_D(o, o_\ell^*) > r(o)$, then o_ℓ^* cannot lie on an IFS to o. Thus, any paths in $G(o')$ originating at o_ℓ^* will not reach o and so we do not need to consider these directions. Hence, any such improving flips of o_ℓ are pruned from the tree.

We continue to construct $G(o')$ like this, pruning any nodes with outcome rank above our bound. This continues until either o is reached or all leaf nodes of $G(o')$ have been considered. If we obtain $o \in G(o')$, then, as we argued above, this proves that there is an IFS $o' \rightsquigarrow o$ and so the dominance query is true ($N \vDash o \succ o'$). As we proved in our above arguments, any node that is pruned cannot lead to o. Thus, in constructing $G(o')$, we explored every path in G_N originating at o' that could plausibly lead to o. If all leaf nodes have been considered then no additional outcomes can be reached by following these plausible paths. Thus, if o is not present at this point, it cannot be reached from o' in G_N and so the dominance query is false ($N \nvDash o \succ o'$).

The upper bound on ranks means that we can stop considering an improving flipping sequence as soon we reach an outcome o^*, such that $r(o^*) + L_D(o, o^*)$ exceeds $r(o)$, rather than pursuing unsuccessful paths until they reach the optimal outcome (where all IFS terminate). This makes our search more efficient. Consider the strict, rank induced consistent ordering, $\succ^R$, that we introduced in §2.3.3. Visualise this ordering as a list of outcomes with the optimal outcome at the top and the worst at the bottom. We know that o is above o' as $r(o) \geq r(o') + L_D(o, o')$, so $r(o) > r(o')$ and, thus, $o \succ^R o'$. All IFS invariably move up this list. Thus, when searching for an $o' \rightsquigarrow o$ IFS, we are searching for a sequence that starts at o' and moves up the list to o. By applying our upper bound, we restrict the search area to the $o \to o'$ segment of this list, as searching stops as soon as you reach any outcome above o, as such outcomes have rank greater than or equal to $r(o)$. In fact, this upper bound will usually restrict the search further, as there are likely to be outcomes between o and o' that also violate the upper bound due to the L_D term. The maximum possible number of outcomes we can consider in this search process is equal to the length of the $o \to o'$ list segment, though it will generally be less. Thus, it will always terminate in finite time (it will also generally be quicker for outcomes o and o' that are closer in $\succ^R$). This can also be seen by the fact that the search tree cannot contain duplicates and so, as there are finitely many outcomes, the tree must be of finite size. Thus, there are only finitely many edges to add and leaves to consider. Therefore, there can only be a finite number of steps to our search process.

Note that we have not yet addressed how we select which leaf node to consider next. We discuss the possible methods in §2.4.2, but for the following example we simply prioritise by depth. That is, leaf nodes higher up in the tree are considered first. However, in practice, we will find that there is no choice of nodes in Example 2.20.

Example 2.20. We now use the CP-net given in Example 1.2 to illustrate our method of answering dominance queries with rank pruning.

Does $N \models \bar{a}b\bar{\bar{c}}d \succ \bar{a}b\bar{c}\bar{d}$ hold? First, we evaluate the ranks of these two outcomes, which can be done by consulting W, given in Figure 2.1, or using Algorithm 1:

$$r(\bar{a}b\bar{\bar{c}}d) = \frac{121}{24}, \quad r(\bar{a}b\bar{c}\bar{d}) = \frac{114}{24}.$$

We must also calculate $L_D(\bar{a}b\bar{\bar{c}}d, \bar{a}b\bar{c}\bar{d})$. We calculate $L(X)$ for all $X \in V$:

$$L(A) = \frac{7}{6}, \quad L(B) = \frac{7}{6}, \quad L(C) = \frac{1}{8}, \quad L(D) = \frac{1}{24}.$$

Then, we calculate $L_D(\bar{a}b\bar{\bar{c}}d, \bar{a}b\bar{c}\bar{d})$:

$$L_D(\bar{a}b\bar{\bar{c}}d, \bar{a}b\bar{c}\bar{d}) = \sum_{X \in \{C,D\}} L(X) = \frac{1}{6}.$$

As $r(\bar{a}b\bar{\bar{c}}d) > r(\bar{a}b\bar{c}\bar{d}) + L_D(\bar{a}b\bar{\bar{c}}d, \bar{a}b\bar{c}\bar{d})$, to answer the dominance query we will need to determine whether there exists an IFS from $\bar{a}b\bar{c}\bar{d}$ to $\bar{a}b\bar{\bar{c}}d$ by constructing $G(\bar{a}b\bar{c}\bar{d})$.

First we make $\bar{a}b\bar{c}\bar{d}$ the root of the tree. As it is the only leaf node, we start by adding the improving flips, $F(\bar{a}b\bar{c}\bar{d})$, to the tree. From the CPTs, we can see that only A and C can be changed into a more preferred position from $\bar{a}b\bar{c}\bar{d}$. So we have $F(\bar{a}b\bar{c}\bar{d}) = \{ab\bar{c}\bar{d}, \bar{a}bc\bar{d}, \bar{a}b\bar{\bar{c}}\bar{d}\}$. As $\bar{a}b\bar{\bar{c}}d \notin F(\bar{a}bcd)$, we cannot reach $\bar{a}b\bar{\bar{c}}d$ from $\bar{a}b\bar{c}\bar{d}$ in one improving flip and so we must continue to construct $G(\bar{a}b\bar{c}\bar{d})$. Thus, we add an edge $\bar{a}b\bar{c}\bar{d} \rightarrow o$ for each $o \in F(\bar{a}b\bar{c}\bar{d})$. We now calculate $r(o) + L_D(\bar{a}b\bar{\bar{c}}d, o)$ for each $o \in F(\bar{a}b\bar{c}\bar{d})$. Again, we use W or Algorithm 1 to calculate the ranks and we can use the $L(X)$ values from above to calculate the L_D terms.

$$\begin{aligned}
r(ab\bar{c}\bar{d}) + L_D(\bar{a}b\bar{\bar{c}}d, ab\bar{c}\bar{d}) =& \frac{154}{24} + \left(\frac{7}{6} + \frac{1}{8} + \frac{1}{24}\right) = \frac{186}{24}, \\
r(\bar{a}bc\bar{d}) + L_D(\bar{a}b\bar{\bar{c}}d, \bar{a}bc\bar{d}) =& \frac{117}{24} + \left(\frac{1}{8} + \frac{1}{24}\right) = \frac{121}{24}, \\
r(\bar{a}b\bar{\bar{c}}\bar{d}) + L_D(\bar{a}b\bar{\bar{c}}d, \bar{a}b\bar{\bar{c}}\bar{d}) =& \frac{122}{24} + \left(\frac{1}{24}\right) = \frac{123}{24}.
\end{aligned}$$

As $ab\bar{c}\bar{d}$ and $\bar{a}b\bar{\bar{c}}\bar{d}$ both satisfy $r(o) + L_D(\bar{a}b\bar{\bar{c}}d, o) > r(\bar{a}b\bar{\bar{c}}d)$, we do not need to pursue these search directions further (as they will not lie on an IFS from $\bar{a}b\bar{c}\bar{d}$ to $\bar{a}b\bar{\bar{c}}d$). Thus, these nodes are pruned from $G(\bar{a}b\bar{c}\bar{d})$.

We again have only one leaf node in the tree (as the others have been pruned), $\bar{a}bc\bar{d}$. Thus, we next add the improving flips of $\bar{a}bc\bar{d}$. This will show us whether $\bar{a}b\bar{\bar{c}}d$ can be reached from $\bar{a}b\bar{c}\bar{d}$ in two improving flips. By inspecting the CPTs, we find $F(\bar{a}bc\bar{d}) = \{abc\bar{d}, \bar{a}b\bar{\bar{c}}\bar{d}, \bar{a}bcd\}$. As $\bar{a}b\bar{\bar{c}}d$ is not one of these flips, we have not yet reached $\bar{a}b\bar{\bar{c}}d$ and so we must continue to construct $G(\bar{a}b\bar{c}\bar{d})$. As $\bar{a}b\bar{\bar{c}}\bar{d}$ is already present in the tree, we do not add it again. We add the edges $\bar{a}bc\bar{d} \rightarrow abc\bar{d}$ and $\bar{a}bc\bar{d} \rightarrow \bar{a}bcd$ only. Evaluate the ranks and L_D terms of the new nodes:

$$\begin{aligned}
r(abc\bar{d}) + L_D(\bar{a}b\bar{\bar{c}}d, abc\bar{d}) =& \frac{157}{24} + \left(\frac{7}{6} + \frac{1}{8} + \frac{1}{24}\right) = \frac{189}{24}, \\
r(\bar{a}bcd) + L_D(\bar{a}b\bar{\bar{c}}d, \bar{a}bcd) =& \frac{118}{24} + \left(\frac{1}{8}\right) = \frac{121}{24}.
\end{aligned}$$

As $r(abc\bar{d}) + L_D(\bar{a}b\bar{\bar{c}}d, abc\bar{d}) > r(\bar{a}b\bar{\bar{c}}d)$, we do not need to continue searching from $abc\bar{d}$ and can prune the node from the tree.

Therefore, there is again only one possible node to consider, $\bar{a}bcd$. We start by evaluating the improving flips, $F(\bar{a}bcd) = \{abcd, \bar{a}b\bar{\bar{c}}d\}$.

We have $\bar{a}b\bar{\bar{c}}d \in F(\bar{a}bcd)$, thus $\bar{a}b\bar{\bar{c}}d$ is in our search tree and is, therefore, reachable from $\bar{a}b\bar{c}\bar{d}$ by improving flips. In particular, there is an IFS from $\bar{a}b\bar{c}\bar{d}$ to $\bar{a}b\bar{\bar{c}}d$ of length three (as $\bar{a}bcd$ is at depth two). Thus, our dominance query is true, $N \models \bar{a}b\bar{\bar{c}}d \succ \bar{a}b\bar{c}\bar{d}$ holds. The search tree we have constructed is given in Figure 2.4.

For efficiency, we calculate all $L(X)$ terms first, then calculate L_D terms as necessary from these values. We start by calculating the AF_Y and d_Y terms for every $Y \in V$. Algorithm 6 can calculate any ancestor set in $O(n^3)$ time, from which AF_Y can be calculated in time linear in the size of the ancestor set (see Algorithm 1). Thus, AF_Y can be calculated from $\mathrm{Anc}(Y)$ in less than $O(n)$ time. Algorithm 7 can calculate any d_Y term in $O(n^3)$ time. Thus, we can calculate all AF_Y and d_Y terms for each $Y \in V$ in $O(n(n^3 + n + n^3)) = O(n^4)$ time. To calculate $L(X)$ requires determining the $\mathrm{Ch}(X)$ set and then $L(X)$ can be calculated

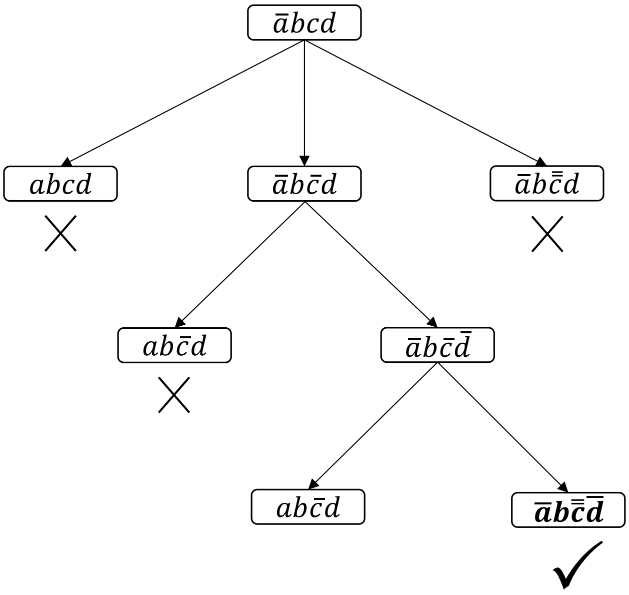


Figure 2.4: Pruned Dominance Query Search Tree

in linear time using the AF_Y and d_Y terms and domain sizes. The latter requires linear time, $O(n)$, as X must have $\leq n-1$ children so $|\{X\} \cup \mathrm{Ch}(X)| \leq n$. The children of X can also be determined in linear time, $O(n)$, from the structure of N (or the equivalent adjacency matrix). Thus, given the AF_Y and d_Y terms, it takes $O(n+n) = O(n)$ time to calculate $L(X)$. Thus, to calculate the AF_Y and d_Y terms and then every $L(X)$ term, it takes $O(n^4 + n(n)) = O(n^4)$ time. Given o and o', we can determine $\mathrm{HD}(o, o')$ in $O(n)$ time. Then, given we have all the $L(X)$ terms already, we can calculate $L_D(o, o')$ in time linear in $\mathrm{HD}(o, o') \leq\ n$. Thus, we can find each $L_D(o, o')$ in $O(n)$ time given all $L(X)$ terms, or in $O(n^4 +\ n) =\ O(n^4)$ time if we had not calculated the $L(X)$ terms previously. As ranks are calculated in $O(n^4)$ time also, this means we can check our pruning condition, 'is $r(o) \geq\ r(o') + L_D(o, o')$?', in $O(n^4 + n^4 + n^4 + 1) = O(n^4)$ time.

We have now provided a new method for pruning the search tree in order to improve dominance testing efficiency. This method can be applied to binary or multivalued CP-nets (and CP-nets with indifference, as we discuss in §2.5) and preserves search completeness in both cases, unlike least variable flipping by Boutilier et al. (2004a) or depth-bounded search by Allen et al. (2017a).

Remark. In this section, we have shown how outcome ranks can be used to improve dominance testing efficiency by pruning the associated search tree. Given any consistent ordering, $\succsim^C$, we can define an analogous method for prune dominance query search trees.

Let $\succsim^C$ be a consistent ordering for N and let o and o' be two outcomes. Suppose we want to answer the dominance query '$N \vDash o \succ o'$?'. If there is an IFS $o' \rightsquigarrow o$, say $o' = o_1, o_2, ..., o_m = o$, then $N \vDash o_{i+1} \succ o_i$ for all i. Thus, as $\succsim^C$ is a consistent ordering, $o_{i+1} \succ^C o_i$ for all i. That is $o_m \succ^C o_{m-1} \succ^C \cdots \succ^C o_1$. Recall that $o_m = o$ and consider $o^* \neq o$. If o^* lies on an IFS terminating at o, then $o \succ^C o^*$. Thus, by the same reasoning as we used previously, when constructing the search tree, $G(o')$, we can prune any nodes that do not come after o in $\succsim^C$ (that is, any o^* such that $o^* \succ^C o$ or $o^* \sim^C o$). Using this pruning method for $\succsim^R$ (the rank induced ordering) would be equivalent to pruning based on relative rank values (no L_D terms) – using the Theorem 2.8 result.

We can also implement a stronger pruning condition that is analogous to using L_D terms (rather than just relative rank values). Suppose $o^* \neq o$ and there is an IFS $o^* \rightsquigarrow o$, $o^* = o_1, o_2, ..., o_m = o$. Suppose that o^* and o differ on the value of k variables, $\text{HD}(o^*, o) = k$. As this IFS changes o^* to o, each of these k variables must change value at least once. Thus, there must be at least k flips in this IFS and so $m \geq k + 1$. As we discussed above, $o_{i+1} \succ^C o_i$, so o_{i+1} is on a higher level of $\succsim^C$ than o_i (see Appendix A, page 243, for the definition of 'level' here). Thus, as $m - 1 \geq k$, o must be at least k levels above o^*. If o^* is in $G(o')$, we only want to pursue this direction if o can be reached from o^* by improving flips, that is, there is an IFS $o^* \rightsquigarrow o$. Thus, by the same arguments as above, we can prune any nodes, o^*, in $G(o')$ where o is not at least $\text{HD}(o^*, o)$ levels above o^* in $\succsim^C$. This is stronger than the above condition as we previously pruned any nodes that were not at least 1 level below o and $\text{HD}(o, o^*) \geq 1$. This pruning is analogous to pruning any nodes that do not have rank at least L_D less than $r(o)$. Note, however, that using rank pruning as detailed in this section (with ranks and L_D terms) is not equivalent to using this method with $\succ^R$.

This pruning condition can be checked in linear time given $\succsim^C$. Further, as most CP-nets have multiple consistent orderings, this general method gives several distinct pruning conditions. These methods can be combined with each other, rank pruning, or any of the existing pruning conditions (see §2.4.2 for an explanation of how pruning methods are combined) to create a more effective pruning schema.

The only point at which the above process consults the CP-net, N, is to determine the improving flips when constructing $G(o')$. However, this is not necessary. An improving flip of o is an outcome that can be obtained from o by improving the value of exactly one variable. If $\text{HD}(o, o') = 1$, then we must have either $N \vDash o \succ o'$ or $N \vDash o' \succ o$, as o and o' must be connected by an edge in G_N (by the definition of preference graph). Respectively, we have either $o \succ^C o'$ or $o' \succ^C o$, as $\succsim^C$ is consistent. The improving flips of o are those outcomes, o'.

such that $\text{HD}(o, o')$ and $N \vDash o' \succ o$ (otherwise they are worsening flips). Thus, $F(o) = \{o' \in \Omega | \text{HD}(o, o') = 1 \wedge o' \succ^C o\}$. Hence, we can find the improving flips directly from $\succsim^C$ and we can answer dominance queries from $\succsim^C$ alone. This is not surprising as all CP-net information is encoded by consistent orderings, as shown by Theorem 2.12.

Similarly, we can find improving flips using rank values alone. If $\text{HD}(o, o') = 1$, then either $N \vDash o \succ o'$ or $N \vDash o' \succ o$ and so $r(o) > r(o')$ or $r(o') > r(o)$, respectively (by Theorem 2.8). Thus, $F(o) = \{o' \in \Omega | \text{HD}(o, o') = 1 \wedge r(o') > r(o)\}$. Thus, as the improving flips can be found directly from rank values, then we can also answer dominance queries (by constructing and pruning $G(o')$) directly from rank and $L(X)$ values. This is also not surprising as ranks contain all CP-net preference information by Corollary 2.13.

The latter, stronger $\succsim^C$ pruning condition also provides another method for answer ordering queries directly from $\succsim^C$ in constant time. Let $\text{HD}(o, o') = k$ and $o \neq o'$ (so $k > 0$). If o is less than k levels above o' in $\succsim^C$, then we know there is no $o' \rightsquigarrow o$ IFS and so $N \nvDash o \succ o'$. Thus, $o' \succ o$ is a consistent ordering. If o is at least k levels above o', then, as $k > 0$, o is on a higher level than o' in $\succsim^C$ and so $o \succ^C o'$. Thus, $o \succ o'$ is a consistent order (as $\succsim^C$ is a consistent ordering). This method can also provide more information if considered in both directions (as we did with a previous rank test for ordering queries). Suppose that o is less than k levels above o' in $\succsim^C$ and we conclude that $o' \succ o$ is consistent. If o' is also not at least k levels above o, then by the same argument, $o \succ o'$ is also consistent. In this case, we now know that $N \vDash o \bowtie o'$ and, thus, both associated dominance queries have been answered. Hence, we can gain more information by considering both directions. This test is analogous to the rank test for ordering queries that utilises L_D terms, discussed previously. This test can, in some cases, be sufficient to answer a given dominance query. Thus, we would check this condition (or the previous $\succsim^C$ based method for ordering queries) before commencing the above search (and pruning) procedures. This is analogous to checking the relative rank (and L_D) values prior to commencing the search tree construction when utilising rank pruning.

Previously, we saw that combining distinct methods of answering ordering queries can yield more information. This general method provides a different test for each consistent ordering, $\succsim^C$, which can be combined with each other or with any other method of answering ordering queries.

2.4.2 Experimental Evaluation and Comparison of Rank Pruning

In §2.4.1, we showed how our outcome ranks can be used to make dominance testing more efficient by pruning the search tree. In this section, we evaluate the performance of our rank pruning in comparison with the existing pruning methods. We also examine the performance of all possible combinations of these methods, in order to determine the most effective pruning schema for efficient dominance testing. We first give the details of our experiments, then analyse the performance results of the different dominance testing methods and combinations. These results show our rank pruning to be the best of the individual methods, and the most important to include when considering combinations of techniques.

Experiment

There are many existing methods to improve dominance testing efficiency, as we reviewed in §2.2.3. We have chosen to compare rank pruning only to other methods for pruning the search tree that preserve search completeness (otherwise, dominance queries may be answered incorrectly). In particular, we are comparing our rank pruning to penalty pruning by Li et al. (2011a) and suffix fixing by Boutilier et al. (2004a), but not prefix fixing by Wilson (2004b) (due to its symmetry with suffix fixing, we may expect prefix fixing to perform similarly). We have excluded from our comparisons least variable flipping by Boutilier et al. (2004a) and the depth bound on flipping sequences proposed by Allen et al. (2017a), as they do not preserve search completeness. We also do not compare the model checking method introduced by Santhanam et al. (2010), the composition of preference tables introduced by Sun et al. (2017), or the CP-net preprocessing method, forward pruning, by Boutilier et al. (2004a). Forward pruning reduces the original problem, rather than providing an efficient search technique. The resulting, smaller, dominance query remains to be answered by some other technique. Forward pruning and prefix fixing (combined with suffix fixing) are considered in Chapter 3, where we introduce a new method of reducing the dominance query size via CP-net preprocessing.

In §2.4.1, we showed that our rank pruning condition can be checked in $O(n^4)$ time. Comparatively, the suffix fixing pruning condition can be checked in $O(n)$ time and the penalty pruning condition can be checked in polynomial (in n) time. The efficiency of the dominance testing process is determined both by these complexities and the efficacy of the pruning methods.

It is simple to combine any of the three pruning measures we are considering. Suppose we wish to answer the dominance query $N \vDash o \succ o'$, utilising the combination of a set of pruning measures, Γ. We build $G(o')$ as usual. When considering the outcome, o^*, let $F(o^*)$ denote the set of all improving flips of o^*, as in §2.4.1. As usual, we prune any elements of $F(o^*)$ that are already present in $G(o')$. Then, for each pruning measure, $\gamma \in \Gamma$, in turn, we prune all elements remaining in $F(o^*)$ that satisfy the pruning condition of γ. Any improving flips that have not been pruned from $F(o^*)$ are added to $G(o')$ in the normal manner. We continue until o is reached, that is, the dominance query is true, or the pruned $G(o')$ is complete (that is, all not-pruned leaves have been considered) and, thus, the dominance query is false.

In our experiment, we evaluated the performance of each pruning measure individually, all pairwise combinations, and all three methods combined. Thus, we compare the performance of seven different pruning schemas. However, in order for these search methods to be fully defined, we must declare how we select the next leaf for consideration when constructing $G(o')$. Different methods of leaf prioritisation have been suggested previously by Boutilier et al. (2004a) (based upon least variable flipping) and Li et al. (2011a) (based on the evaluation function value). One can similarly propose prioritisation heuristics based upon rank values. One could either prioritise the leaf, o^*, with maximal $r(o^*)$ or $r(o^*) + L_D(o^*, o)$ value. We will refer to these as *rank prioritisation* and *rank + diff. prioritisation*, respectively. The reasoning behind both heuristics is that such directions in $G(o')$ will quickly either reach o or terminate (that is, when all nodes that o^* leads to are pruned or have been previously considered). Thus, either the query is answered efficiently or the direction can be efficiently ruled out.

No analysis has been done previously on the effect of the leaf prioritisation choice. Thus, we have varied the heuristics used in our experiments. However, for the sake of efficiency, we have only allowed heuristics that do not require further calculations. Both rank prioritisations require the rank (and L_D) values of the leaves, so they are only used by pruning schemas that include rank pruning. The prioritisation heuristic by Li et al. (2011a) (*penalty prioritisation*) requires the evaluation function value of each leaf and is, therefore, only used by schemas that include penalty pruning. We do not consider pruning based on least variable flipping in our experiments (as it does not preserve completeness), so we do not include the associated prioritisation heuristic by Boutilier et al. (2004a) either. When using suffix fixing only (the only pruning schema that contains neither rank nor penalty pruning), we use the trivial minimal depth prioritisation of leaves, as we did in Example 2.20.

We measured the performance of the dominance testing functions in two ways. First, we looked at *outcomes traversed*, this is the number of outcomes added to the search tree before an answer to the dominance query can be determined (this count does not include improving flips that are pruned). This is similar to the measure used by Li et al. (2011a) in their pruning method comparisons (where they compared penalty pruning combined with suffix fixing to suffix fixing and least variable flipping). Outcomes traversed provides us with a theoretical measure of how effective the different methods are at pruning the search tree. It reflects the number of steps the different algorithms have to go through before the queries can be answered, thus showing how efficient the different methods are in a theoretical sense. This measure has the advantage of being independent of the specific code used and the order in which pruning conditions in combinations are considered.

Note that it is possible for the number of outcomes traversed to be zero, that is, the dominance query may be answered without starting to construct a search tree. This can happen in three different ways for the dominance query $N \vDash o \succ o$; first, if $o = o'$, then this is trivially false. Second, if (one of) the pruning measure(s) used is penalty pruning, then, if $f(o') < 0$, we can determine the dominance query to be false (Li et al., 2011a). Finally, if (one of) the pruning measure(s) used is rank pruning, then, if $r(o) - r(o') < L_D(o, o')$, we can determine the dominance query to be false, by Corollary 2.18. As these conditions are all assessed before starting to construct the search tree, they result in zero outcomes traversed. In Appendix C.2, we look at the proportion of queries that the different conditions can immediately determine to be false (that is, those queries that have zero outcomes traversed). By evaluating this proportion in comparison to the total proportion of false queries, we can see how accurately these initial conditions predict the dominance query outcome.

Our second measure of performance is the time elapsed (in seconds) while the dominance testing algorithm answers the query (this was not measured in the performance experiments by Li et al., 2011a). Whilst this measure is dependent upon the exact code used, we have tried to keep the code for the different functions as uniform as possible, so that differences in performance are due to the methods rather than the code. From time elapsed, we can identify which method will be the most efficient in practice. By looking at both performance measures, we can see the tradeoff between how effective a pruning method is theoretically and the time cost due to the complexity of implementation. Ultimately, we will see if the theoretical benefit is worth the cost in complexity by looking at the time elapsed results.

The experiments we ran to evaluate performance were as follows. For given n (number of CP-net variables) and d_M (maximum domain size of the variables) values, 100 CP-nets were randomly generated. Each of these CP-nets has an acyclic structure over n variables, each variable has a domain size of at most d_M (and at least two), and all parent-child relations are valid (that is, if there is an edge $X \rightarrow Y$ in the CP-net structure, it is possible to change the preference over Y by altering the value of X only). Full details of the CP-net random generation process is given in Appendix C.1. For each CP-net, 10 dominance queries were randomly generated by randomly selecting a pair of outcomes. Each of these 1000 dominance queries were answered by all seven dominance testing functions (with all possible leaf prioritisation heuristics) and the outcomes traversed and time elapsed were recorded. The average of these results over the 1000 queries are the values plotted for each (n, d_M) pair in the following graphs.

This experiment was run in the binary case, $d_M = 2$, for $n = 3, 4, ..., 19$. For the multivalued variable case, we allowed domain size to be up to five. We ran the experiments in this case ($d_M = 5$) for $n = 3, 4, ..., 10$.

Results

Note that the results presented in this section (and Appendices C.2 and C.3) differ from those presented in our journal paper, Laing et al. (2019). Firstly, the original functions have been translated from R to C++. Secondly, larger values of n are tested in these experiments (made possible by the translation to C++).

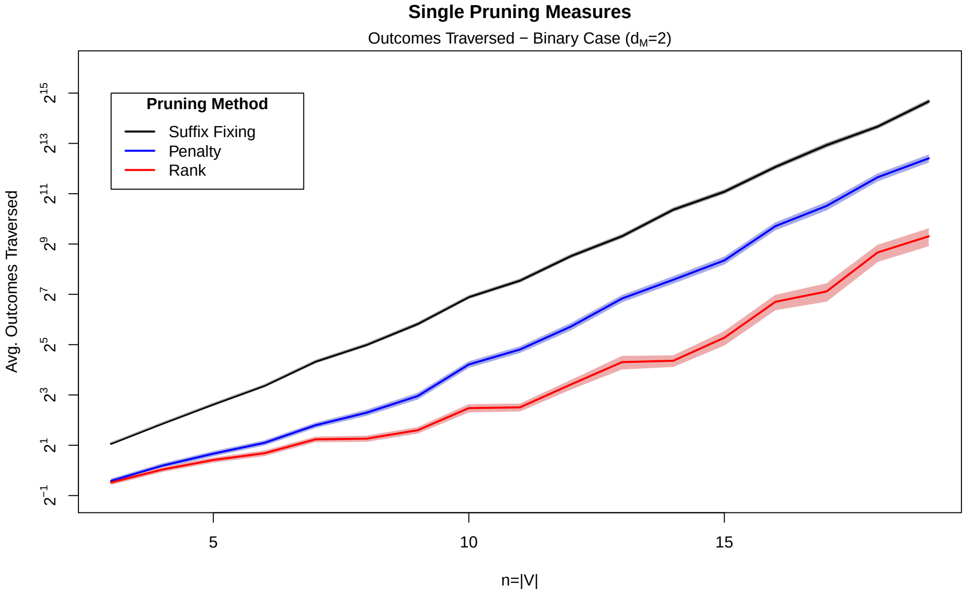
Each of the seven functions were tested with all possible leaf prioritisation heuristics (as described in the previous experiment section). The full performance results are given in Appendix C.3. The graphs in this section show only the functions using their respective optimal leaf prioritisation heuristics. For suffix fixing, penalty pruning, and their combination, the optimal prioritisation methods are minimal depth prioritisation, penalty prioritisation, and penalty prioritisation, respectively (note that there is only one choice of prioritisation heuristic in these cases). For all other pruning schemas (that is, all pruning schemas that include rank pruning), the optimal prioritisation heuristic is rank prioritisation. The results in Appendix C.3 show that rank prioritisation performs best in almost all cases, though, in general, changing the leaf prioritisation method does not have a significant effect on performance. However, the effect of the prioritisation method can be sufficient, in some cases, to affect which pruning method performs better on average.

For each of the seven dominance testing methods (using their optimal leaf prioritisation method), we have four sets of data – outcomes traversed and time elapsed data for both the binary and multivalued CP-net cases. These four data sets are given in Figures 2.5 – 2.8. In each of Figures 2.5 – 2.8, Figure (a) shows the performance of the three pruning measures when used individually. To keep this plot legible, a logarithmic scale is used. Figure (b) shows the performance of all seven possible combinations of the three pruning measures.

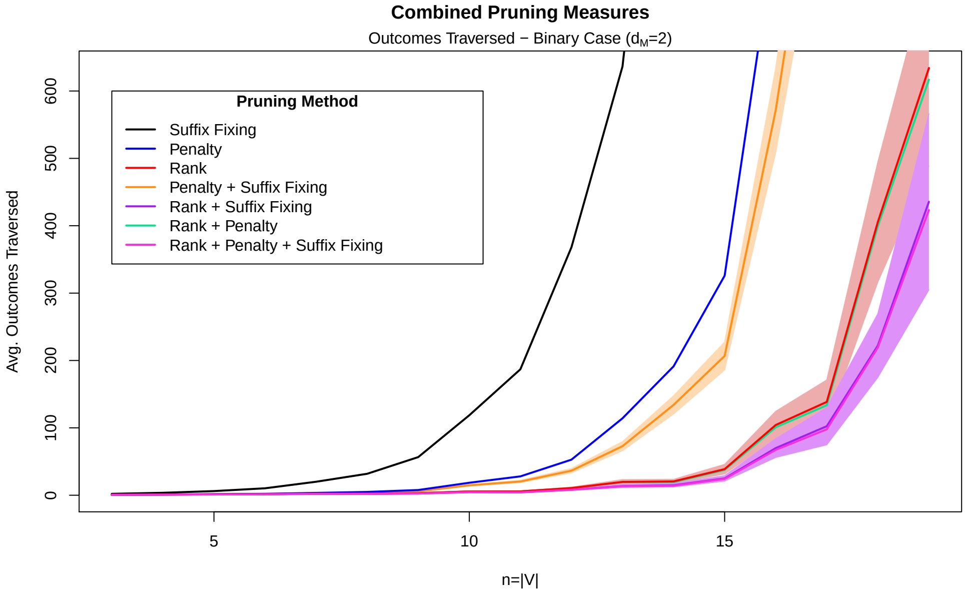
In each figure, several ±SE (standard error) intervals are illustrated by a shaded region in the corresponding function's colour. The standard error interval depicts where we expect the true mean performance of the function to lie. The uncertainty represented by this interval comes from the fact that the complexity of a dominance query, regardless of the pruning technique used, is dependent upon both the CP-net and the outcomes of interest; CP-nets with denser structures, or more convoluted preference graphs, are more likely to produce dominance queries that take longer to answer. Once a CP-net has been chosen, the position of the outcomes of interest within the preference graph further impacts how difficult the dominance query is to answer. The efficacy of the different pruning methods are also likely to vary between queries. As our CP-nets and queries were randomly generated, it is unsurprising that each function shows variation in performance. However, as all functions were tested on the same set of dominance queries, our results should accurately portray their relative performance on average.

In the multivalued case, the domain sizes were allowed to vary between two and five. Larger domain sizes will produce harder dominance queries in general, so we would expect this extra uncertainty to result in further variation within the results. Moreover, CP-nets with larger domain sizes have larger preference graphs, so there will also be more variation in dominance queries of the same CP-net. Hence, in the multivalued case, we expect more uncertainty in the average performance of the functions and this is reflected by the wider error intervals in the multivalued case plots (Figures 2.7 and 2.8).

From Figures 2.5(b) and 2.7(b), we can see that adding extra pruning conditions always improves the theoretical performance of a method (that is, it results in fewer outcomes traversed on average). This holds both when optimal prioritisation methods are used and whenever the prioritisation method is kept fixed and additional pruning measures are added. This shows that all three pruning measures are distinct, and that no pruning measure is subsumed by any other. Further, this shows us that each technique prunes branches that are not affected by either of the other two methods. It is not obvious from the way in which they are formulated that the three pruning measures are distinct in this manner. Moreover, this

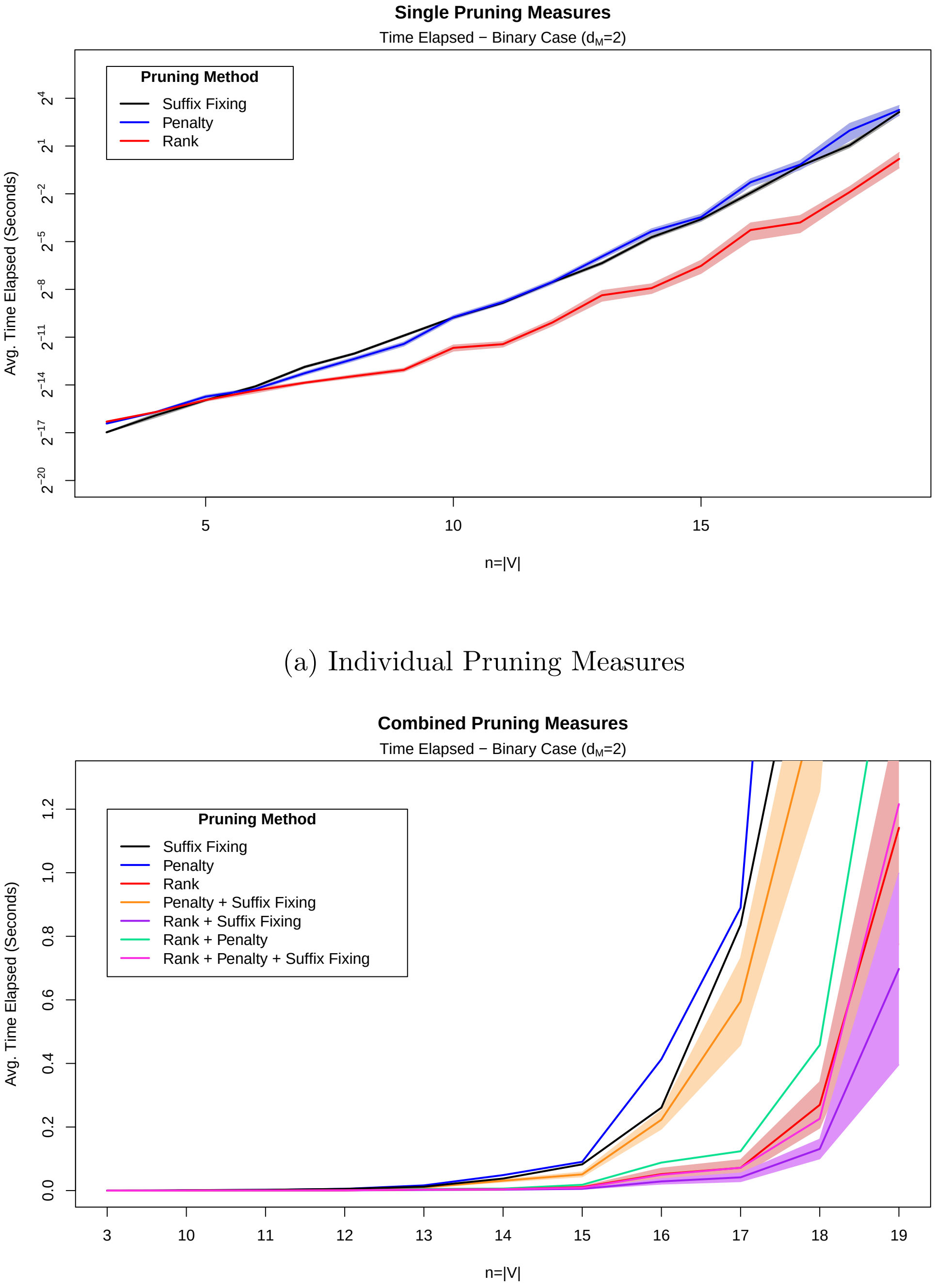


(a) Individual Pruning Measures



(b) All Pruning Measure Combinations

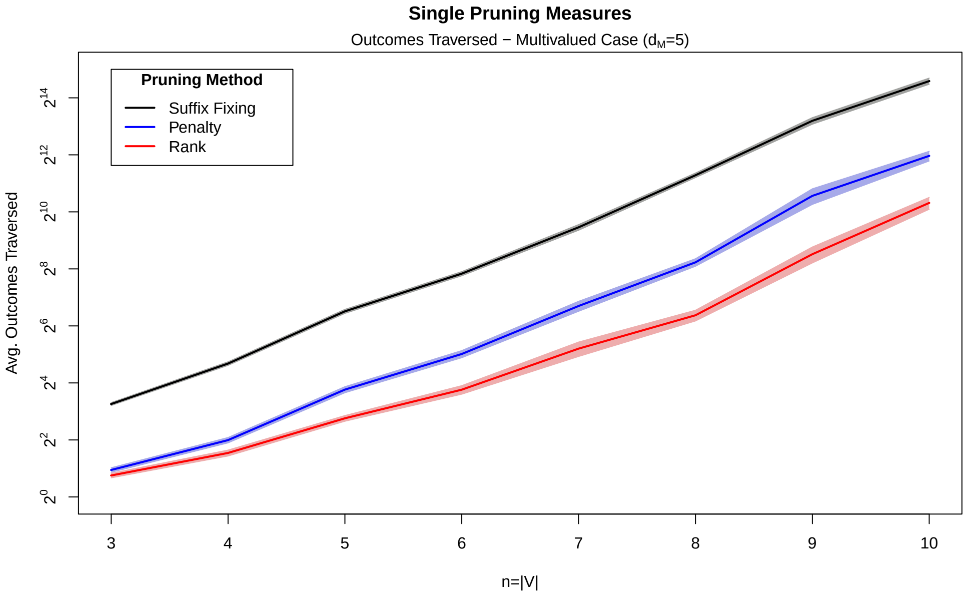
Figure 2.5: Pruning Comparison Results – Binary CP-Nets, Outcomes Traversed



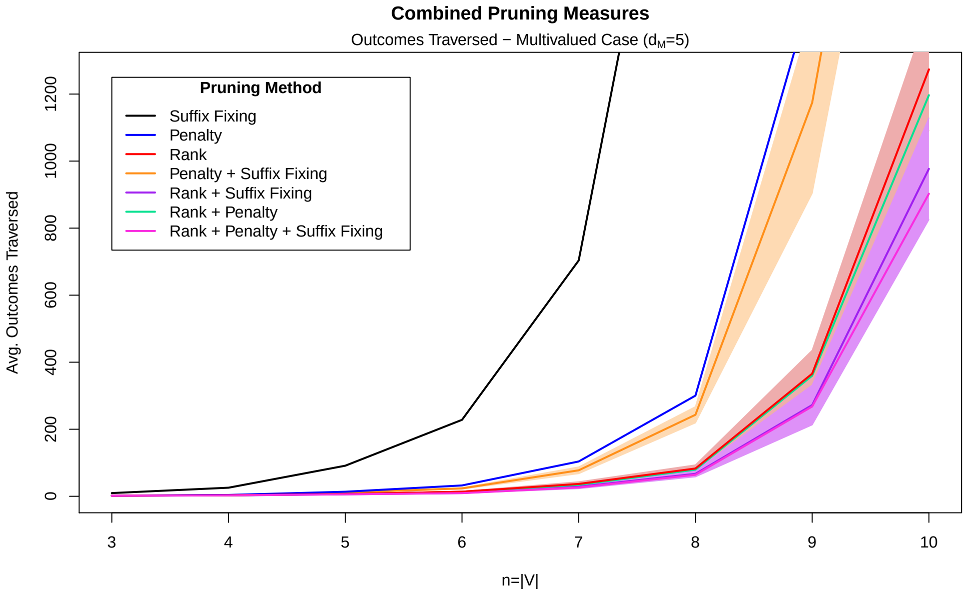
(a) Individual Pruning Measures

(b) All Pruning Measure Combinations
Note: n values between 3 and 10 are compressed in order to improve plot clarity for larger n values

Figure 2.6: Pruning Comparison Results – Binary CP-Nets, Time Elapsed

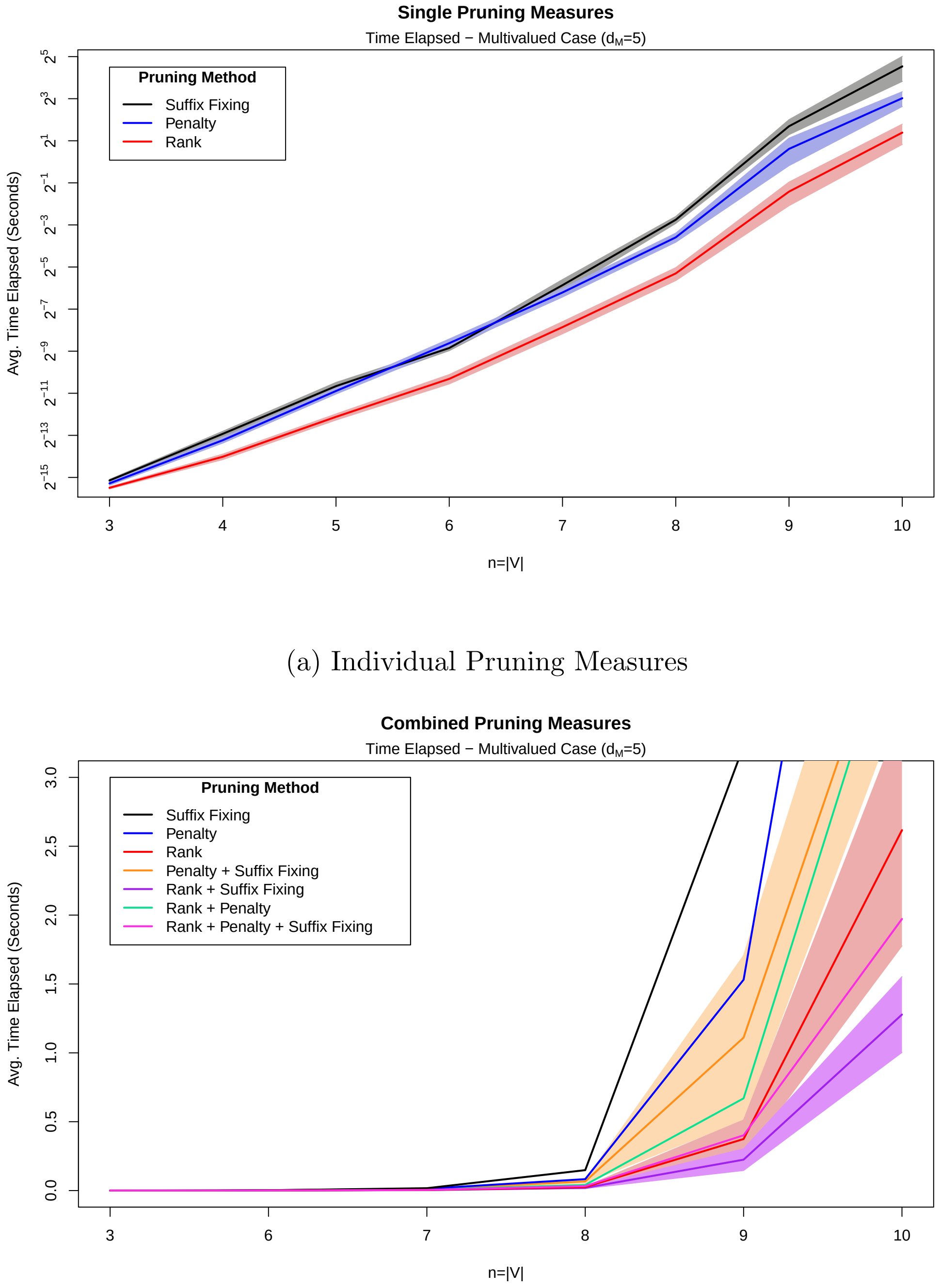


(a) Individual Pruning Measures



(b) All Pruning Measure Combinations

Figure 2.7: Pruning Comparison Results – Multivalued CP-Nets, Outcomes Traversed



(a) Individual Pruning Measures

(b) All Pruning Measure Combinations

Note: n values between 3 and 6 are compressed in order to improve plot clarity for larger n values

Figure 2.8: Pruning Comparison Results – Multivalued CP-Nets, Time Elapsed

has not been previously confirmed by existing literature. From this result, it is unsurprising that the single best performing function, in the theoretical sense, is that which uses all three pruning measures.

However, finding the best pruning schema is not as simple as applying as many pruning conditions as possible. As we are aware that additional pruning methods come at the cost of additional complexity, we would naturally question whether these improvements are large enough to warrant the additional cost. Looking at the time elapsed results (Figures 2.6(b) and 2.8(b)), we can see that some of these 'improvements' actually increase the average time taken, so the theoretical benefit is not worth the complexity cost. In particular, we find that a pairwise combination is actually the most efficient dominance testing method – faster than using all three pruning methods.

Consider the Figure (a) plots, which show the performance of the three pruning methods when used individually. It is clear in all four cases that rank pruning is the most effective and most efficient method of the three by a large margin. Further, the performance results of rank pruning (outcomes traversed or time elapsed) show a slower rate of growth than the others as the number of variables (n) increases. Thus, if we wanted to use a single pruning method, rank pruning is the best choice by a large margin.

Now consider the Figure (b) plots, these show the performance of all possible combinations of the different pruning methods. In all four of these figures, the black, blue, and orange lines (suffix fixing, penalty pruning, and their combination), perform distinctly worse than the rest as n increases. The remaining functions perform notably better and show a slower rate of growth with n. These more effective and efficient pruning methods are exactly those combinations that include rank pruning. Hence, we can see a clear distinction in performance between those functions that do and do not apply rank pruning. From this, we may conclude that rank pruning is a necessary ingredient for a good pruning schema.

In Figure (b), the red shaded area shows the standard error interval for rank pruning, the best performing of the individual pruning measures. Thus, only functions that lie below this area may be considered significantly better than using rank pruning alone. In both Figures 2.5(b) and 2.7(b), adding penalty pruning to rank pruning makes little improvement to the average number of outcomes traversed. This suggests that there are few branches pruned by penalty pruning that are not already pruned by rank pruning. This would account for why adding penalty pruning to rank pruning actually increases the time elapsed. This is because the additional complexity of checking the penalty condition outweighs the minor theoretical benefit.

The combination of rank pruning with suffix fixing and the combination of all three measures both perform significantly better than rank pruning alone, in terms of outcomes traversed (Figures 2.5(b) and 2.7(b)). This is probably due to less overlap in the branches pruned by rank pruning and suffix fixing. The two combinations show very similar performances in terms of outcomes traversed (both in the binary and multivalued cases), though the function using all three methods does slightly better in this theoretical case, as expected. In terms of time elapsed (Figures 2.6(b) and 2.8(b)), the combination of all three performs similarly to rank pruning alone, whereas rank pruning and suffix fixing is notably faster. As using all three techniques takes a similar amount of time to rank pruning alone, this shows that the associated cost of implementing the additional pruning measures is not worth the theoretical benefit. The fact that the rank pruning and suffix fixing combination is notably faster shows, again, that the slight theoretical improvement provided by penalty pruning is not worth the associated complexity cost. However, the improvement of using suffix is worth the associated cost – this is perhaps unsurprising as we can see from the outcomes traversed that suffix fixing provides a large theoretical improvement and it is a simple (linear) pruning condition to check.

From the above results, we have seen that our rank pruning is the most effective and efficient of the individual methods considered. Further, from the clear distinction between functions that do and do not utilise rank pruning, we can see that rank pruning constitutes a valuable contribution to the existing methods when we allow combinations. Considering all possible combinations of the pruning methods, the above results suggest that the most efficient combination for dominance testing is rank pruning and suffix fixing.

2.5 Outcome Ranks for CP-nets with Indifference

In this section, we provide a more general form of our outcome rank formula that allows for indifference statements within the CP-net's CPTs. These generalised ranks again reflect all entailed relations and, therefore, allow all of our previous methods and results to be applied to CP-nets that express indifference.

We do not assume in this section that the CPT(X) preference ordering over Dom(X), given the values taken by Pa(X), is a strict ordering (Boutilier et al., 2004a). Consider the CP-net given in Example 1.2. We would now permit CPT(C) to express that, if it is a short flight in term time, then the user prefers to fly economy, but is indifferent between first and business class. This would make the CPT(C) entry that corresponds to $AB = ab$ be $c \succ \bar{c} \sim \bar{\bar{c}}$. This kind of *ceteris paribus* indifference statement is natural and likely to be commonplace when looking at real world systems (Allen, 2013). In particular, for problems with a large number of outcomes, there are likely to be instances of user indifference between certain outcome pairs. Thus, being able to deal with such indifference expands the applications of our results. Furthermore, if one were comfortable modelling unknown preferences as indifference, our results could also be applied to partially specified CP-nets.

Boutilier et al. (2004a) show that the presence of such indifference allows CP-nets with acyclic structures to be inconsistent (that is, they have no consistent ordering as they entail both $o_1 \succ o_2$ and $o_2 \succ o_1$ for some outcome pair). However, this can be avoided if one assumes the following condition (Boutilier et al., 2004a). Suppose we have two variables, $X, Y \in V$, such that X is a parent of Y. Let $P_X = \text{Pa}(X)$ and $P'_Y = \text{Pa}(Y) \backslash \{X\}$. Let $\mathbf{u}_1 \in \text{Dom}(P_X)$ and suppose that the user is indifferent between x and x' given $P_X = \mathbf{u}_1$ (where $x, x' \in \text{Dom}(X)$). Let $W = P_X \cap P'_Y$ and let $\mathbf{u}_2 \in \text{Dom}(P'_Y)$ be such that $\mathbf{u}_1[W] = \mathbf{u}_2[W]$. The user's preference over Y must be the same under both $\text{Pa}(Y) = \mathbf{u}_2 x$ and $\text{Pa}(Y) = \mathbf{u}_2 x'$. More simply, changing between indifferent parental assignments cannot affect the preference over the child. To ensure consistency, we assume here that all CP-nets with indifference statements obey this condition.

Recall from §2.3.2, that the rank of an outcome, o, is the sum of the weights attached to each variable assignment in o. These weights were constructed to approximate the utility of each variable choice in o. If $o[X] = x$ and $o[\text{Pa}(X)] = \mathbf{u}$, then the weight attached to the assignment $X = x$ is:

$$AF_X(d_X + 1)P_P(X = x \mid \text{Pa}(X) = \mathbf{u}).$$

The justification for the presence of each of these factors remains valid for CP-nets with indifference statements. Thus, we do not need to create a new weighting convention, we simply need to generalize this formula so that it is defined in the case of indifference. The AF_X and d_X terms depend only on the CP-net structure, not on the CPTs, and thus can remain as they were defined previously. The $P_P(X = x \mid \text{Pa}(X) = \mathbf{u})$ factor, as defined in §2.3.2, needs to be generalised to permit indifference statements.

Recall that $P_P(X = x \mid \text{Pa}(X) = \mathbf{u})$ is a factor on the (0,1] scale, indicating to what degree the user prefers this choice of value for X (given $\text{Pa}(X) = \mathbf{u}$). We redefine $P_P(X = x \mid \text{Pa}(X) = \mathbf{u})$ more generally, while retaining this interpretation, as follows.

Definition 2.21. Let N be a CP-net over variables V, which may have indifferences in its CPTs. Let $X \in V$ and $\mathbf{u} \in \mathrm{Dom}(\mathrm{Pa}(X))$. Suppose the row of $\mathrm{CPT}(X)$ that corresponds to $\mathrm{Pa}(X) = \mathbf{u}$ has ℓ indifferences. We define the *generalised preference position* of the assignment $X = x$ (given $\mathrm{Pa}(X) = \mathbf{u}$) as follows:

$$P_P(X = x \mid \mathrm{Pa}(X) = \mathbf{u}) = \frac{(n_X - \ell) - k + 1}{n_X - \ell},$$

where k is the position of preference of the choice of $X = x$ given $\mathrm{Pa}(X) = \mathbf{u}$. Note that we consider all values of X to which the user is pairwise indifferent to be in the same preference position. That is, there are $n_X - \ell$ possible positions of preference $(1, 2, ..., n_X - \ell)$. Here, $k = 1$ if x is (one of) the most preferred value(s) X can take, $k = 2$ if x is (one of) the value(s) of X in the 2^{nd} most preferred position, and so on.

Example 2.22. Let N be a CP-net over variables V. Let $X \in V$ be some variable with the following row in its CPT:

$\mathrm{Pa}(X) = \mathbf{u}$	$x_1 \succ x_2 \sim x_3 \sim x_4 \succ x_5 \succ x_6 \sim x_7 \succ x_8$

Then, using the generalised preference position definition above, we have the following P_P values:

$$\begin{aligned} P_P(X = x_1 \mid \mathrm{Pa}(X) = \mathbf{u}) &= \frac{(8-3) - 1 + 1}{8-3} = \frac{5}{5}, \\ P_P(X = x_2 \mid \mathrm{Pa}(X) = \mathbf{u}) &= \frac{(8-3) - 2 + 1}{8-3} = \frac{4}{5}, \end{aligned}$$

$$\begin{aligned} P_P(X = x_3 \mid \mathrm{Pa}(X) = \mathbf{u}) &= \frac{4}{5}, \quad & P_P(X = x_4 \mid \mathrm{Pa}(X) = \mathbf{u}) &= \frac{4}{5}, \\ P_P(X = x_5 \mid \mathrm{Pa}(X) = \mathbf{u}) &= \frac{3}{5}, \quad & P_P(X = x_6 \mid \mathrm{Pa}(X) = \mathbf{u}) &= \frac{2}{5}, \\ P_P(X = x_7 \mid \mathrm{Pa}(X) = \mathbf{u}) &= \frac{2}{5}, \quad & P_P(X = x_8 \mid \mathrm{Pa}(X) = \mathbf{u}) &= \frac{1}{5}. \end{aligned}$$

Notice that this generalised definition of $P_P(X = x \mid \mathrm{Pa}(X) = \mathbf{u})$ is a value in $\{1/(n_X - \ell), 2/(n_X - \ell), ..., (n_X - \ell - 1)/(n_X - \ell), 1\}$. Further, this can still be interpreted as a factor on the (0,1] scale indicating to what degree the user prefers this choice of value for X (given $\mathrm{Pa}(X) = \mathbf{u}$).

Now that all of the terms in our previous weight formula are defined in the case of N having indifference statements, we can define outcome ranks for CP-nets with indifference.

Definition 2.23. Let N be a CP-net over variables V, which may have indifference statements in its CPTs. Let o be an associated outcome. Then, the *(generalised)* *rank* of o, $r_G(o)$, is defined as

$$r_G(o) = \sum_{X \in V} AF_X(d_X + 1)P_P(X = o[X] \mid \mathrm{Pa}(X) = o[\mathrm{Pa}(X)]),$$

where the P_P terms are generalised preference positions.

In the special case where there are zero indifference statements, the generalised P_P terms clearly simplify to the original definition, given in §2.3.2. Thus, in this special case, the generalised outcome ranks simplify to the original outcome ranks given by Definition 2.5 (for multivalued CP-nets in which we assumed no indifference statements).

Remark. We have could have used an event tree representation to define generalised outcome ranks (as we did for outcome ranks in §2.3) by generalising the notion of event tree representation to include indifference. This generalised $T(N)$ would have the same structure as in §2.3.1, but use the k values from Definition 2.21 in order to label the branches. For example, let N be the CP-net in Example 2.22. At the point where $T(N)$ branches into the possible values of X, if $\mathrm{Pa}(X)$ were previously assigned the values in $\mathbf{u}$, then the X branches would be labelled as follows. The branch corresponding to x_1 would be labelled '1^{st}'. The x_2, x_3, and x_4 branches would all be labelled '2^{nd}'. The x_5 branch would be labelled '3^{rd}', and so on. We again have that N and $T(N)$ are equivalent by an argument almost identical to the proof of Proposition 2.2. The weighted event tree, $W(N)$, would be defined in the same way as in §2.3.2, now using the generalised definition of P_P. The generalised outcome ranks would be defined analogously to the original outcome ranks (Definition 2.5), as the sum of path weights in $W(N)$. Further, $W(N)$ would be equivalent to both $T(N)$ and N, by almost identical reasoning to that given in §2.3.2.

All of our applications of outcome ranks in §2.3.3 and §2.3.4 rely solely on the fact that the ranks reflect all entailed preferences (Theorem 2.8). Naturally, we also want this property to hold for our generalised outcome ranks and the following theorem shows that it does.

Theorem 2.24. *Let N be a CP-net over a set of variables V, which may have indifference statements in its CPTs. Let o and o' be associated outcomes. Then,*

$$\begin{aligned} N \models o \succ o' &\implies r_G(o) > r_G(o') \\ \textit{and } N \models o \sim o' &\implies r_G(o) = r_G(o'). \end{aligned}$$

Proof. See Appendix E.6.

By this result, the (not necessarily strict) ordering of the outcomes, $\succsim^G$, induced by the generalised ranks, r_G, is again a consistent ordering. That is, $N \models o \succ o' \implies o \succ^G o'$ and $N \models o \sim o' \implies o \sim^G o'$. Thus, using the generalised outcome ranks, we can obtain a (not necessarily strict) consistent ordering for any N, which may have indifference statements, using exactly the same method as given in §2.3.3. Similarly, we can obtain a (not necessarily strict) consistent ordering for any subset of the outcomes or for a CP-net with additional plausibility constraints using the methods given in §2.3.3 and §2.3.4 (ignoring any instruction to arbitrarily order outcomes with equal ranks), now using the generalised ranks, r_G, given by Definition 2.23. These orderings can be proven to be consistent in the same way as the corresponding orderings in Section §2.3.3 and §2.3.4.

Boutilier et al. (2004a) claim that their methods for obtaining a consistent ordering of (any subset of) the outcomes also apply to CP-nets with indifference. However, the complexity of ordering queries in this case is unknown (though they conjecture that it is hard) and, therefore, so is the complexity of their method for consistently ordering a subset of the outcomes. In contrast, if one uses our method, the complexity of consistently ordering any subset of the outcomes of size k, in the case of indifference, is still $O(n^4k + k^2)$. This is a result of the fact that we can compute $r_G(o)$ in the same time complexity as $r(o)$, as we show below.

Remark. Ordering queries ask, given an outcome pair, o and o', to find a consistent ordering. In the case of no indifference, this is equivalent to proving at least one of $N \not\models o \succ o'$ or $N \not\models o' \succ o$. In the case of indifference, one must prove at least two of $N \not\models o \succ o'$, $N \not\models o' \succ o$, and $N \not\models o \sim o'$. Boutilier et al. (2004a) conjecture that ordering queries are hard in the case of indifference. If a CP-net has indifference statements, then certain outcome pairs are indifferent and neither $o \succ o'$ nor $o' \succ o$ is a consistent ordering. Thus, 'of equal preference' must be an acceptable result of ordering queries in this case. In which case, we can answer ordering queries in $O(n^4)$ time. Given o and o', if $r_G(o) > r_G(o')$, then we know $N \not\models o' \succ o$ and $N \not\models o \sim o'$ by Theorem 2.24. Thus, either they are incomparable or the user prefers o and so $o \succ o'$ is a consistent ordering (as it does not contradict known preferences). If $r_G(o) = r_G(o')$, then we know $N \not\models o' \succ o$ and $N \not\models o \succ o'$ by Theorem 2.24. Thus, they are either indifferent or incomparable outcomes. In which case, asserting that they are of equal preference ($o \sim o'$) does not contradict any known preference and is, thus, consistent. As we show below, generalised ranks can be calculated in $O(n^4)$ time and, thus, this method for answering ordering queries has complexity $O(n^4)$. We have therefore provided a tractable method for answering ordering queries in the case of indifference, if it is considered acceptable to order some incomparable outcomes as equally preferred.

The above method can be performed using any consistent ordering, $\succsim^C$, or any values that induce such a consistent ordering (it is not specific to ranks). If $o \succ^C o'$, then we know $N \not\models o' \succ o$ and $N \not\models o' \sim o$ (otherwise we would have $o' \succ^C o$ or $o \sim^C o'$, respectively, as $\succsim^C$ is consistent), so $o \succ o'$ is a consistent ordering by the same reasoning as above. If $o \sim^C o'$, then $N \not\models o' \succ o$ and $N \not\models o \succ o'$, so $o \sim o'$ is a consistent ordering by the same reasoning as above. Thus, we can answer ordering queries in the case of indifference in constant time from any consistent ordering. This general method gives multiple distinct methods for answering ordering queries as CP-nets usually have multiple consistent orderings. This is useful, as answering ordering queries in multiple ways can yield more information in the same way as in the case of CP-nets without indifference.

In all of the above applications of r_G, we have obtained a consistent ordering (of N, some subset of the outcomes, or some constrained CP-net, N_C), $\succsim^G$, which is not necessarily strict. That is, for any entailed relation $o \succ o'$ (or $o \sim o'$) we have $o \succ^G o'$ (or $o \sim^G o'$). The presence of indifference might mean that we do not mind a non-strict ordering; however $o \sim^G o' \not\Rightarrow o$ and o' are indifferent, they could also be incomparable. In order to obtain an ordering where only indifferent outcomes are ranked equally, one would need to perform an indifference query on every pair $o \sim^G o'$, to determine whether $N \models o \sim o'$ holds or not. If this does not hold, then $N \models o \bowtie o'$, and so the outcomes can be ordered arbitrarily as before. However, all $N \models o \sim o'$ pairs must be kept as $o \sim^G o'$ for consistency. Indifference queries can be answered in $O(n)$ time, as we show below, so the efficiency of this process depends on the number of $\sim^G$ instances in the ordering. Alternatively, if a strict consistent ordering is required (so we are not interested in preserving indifference), then we can obtain a strict ordering, $\succ^G$, from $\succsim^G$ simply by forcing outcomes of equal rank into an arbitrary order. This strict ordering retains the property that, for any entailed preference, $o \succ o'$, we have $o \succ^G o'$ (by Theorem 2.24). Thus, we can obtain a strict ordering that is consistent with all entailed preferences (but not indifferences).

Consistent orderings are equivalent to CP-nets in the case of indifference also. This can be proven using a similar proof to Theorem 2.12, only now one must allow preference orders to include indifference. Thus, reducing a CP-net to generalised rank values (and thus the associated consistent ordering) does not lose any information.

The process of updating a consistent ordering given new information, as described in Appendix A, can also be adapted to work in the case of CP-nets with indifference. In this case, outcomes on the same level of the ordering can either be indifferent or incomparable. For a CP-net, if o and o' are on the same level of a consistent ordering and $\mathrm{HD}(o,o') = 1$, then they are indifferent outcomes. Taking the transitive closure of this relation identifies the set(s) of outcomes on a given level that are pairwise indifferent. This set(s) can be found in $O(ndk)$ time, where d is the maximum domain size and k is the number of outcomes on the given level. For a generic preference structure with indifferences, G, we replace the $\mathrm{HD}(o,o') = 1$ condition with 'there is an edge between o and o' in G'. The update procedure when a new (consistent) preference is learned can be adapted to the case of indifference by insisting that pairwise indifference sets can only move level as a whole. We can use a similar update procedure when new (consistent) indifference statement is learned.

Algorithms 1, 6, and 7 can be used to calculate $r_G(o)$ exactly as described for $r(o)$ in §2.3.5 (with the same time complexity) if we make two small adjustments. First, line **10** of Algorithm 1 should use $|\mathrm{Dom}(X_i)|-\ell$ in place of $|\mathrm{Dom}(X_i)|$, where $\ell = \#$ indifferences in the $\mathrm{Pa}(X_i) = o[\mathrm{Pa}(X_i)]$ entry of $\mathrm{CPT}(X_i)$. Second, in the case of indifference statements, the preference positions in the input CPTs must be as defined in Definition 2.21 (these are the k terms). Thus, we can compute $r_G(o)$ in the same time as $r(o)$, that is, $O(n^4)$ time. Thus, all previous complexity results transfer directly to the case of CP-nets with indifferences in their CPTs.

Suppose N is a CP-net, which may have indifferences in its CPTs, and let o and o' be associated outcomes. The dominance query $N \vDash o \succ o'$ can be answered using a method very similar to the one described in §2.4.1. First, note that $N \vDash \; o \succ o'$ if and only if there is an improving flipping sequence $o' \rightsquigarrow o$ (Boutilier et al., 2004a). As there may be indifferences, we must clarify what we mean by an IFS here. An IFS is a sequence of outcomes, $o' = o_1, o_2, ..., o_m = o$, such that, for all i, o_i and o_{i+1} differ on the value taken by exactly one variable and either $N \vDash o_{i+1} \succ o_i$ or $N \vDash o_{i+1} \sim o_i$ holds; further, for at least one j, we must have $N \vDash o_{j+1} \succ o_j$. Returning to our dominance query, if $r_G(o') \geq r_G(o)$, then the dominance query is false by Theorem 2.24. Otherwise, starting from o', we build up the search tree as in §2.4.1, only now each outcome branches into all improving flips *and* all indifferent flips. Only outcomes that are not already in the tree may be added. An outcome, o^*, is pruned (not explored further) if $r_G(o^*) > r_G(o)$ – if o^* is on an $o' \rightsquigarrow o$ IFS, we must have either $N \vDash o \succ o^*$ or $N \vDash o \sim o^*$ and so, by Theorem 2.24, $r_G(o^*) \leq r_G(o)$. As in §2.4.1, this pruning will improve the efficiency of answering dominance queries and, in finitely many steps (as there are only a finite number of outcomes), we will either reach o (dominance query is true) or there will be no more valid leaves to consider (dominance query is false).

Once an outcome, o^*, is reached such that $r_G(o^*) = r_G(o)$, the rank cannot be increased any further and, thus, only indifferent flips can be considered from this point. Thus, we are determining whether o can be reached from o^* via indifference flips, that is, whether $N \vDash o \sim o^*$ holds. We show below that such indifference queries can be answered in $O(n)$ time. Thus, we can improve efficiency further by determining whether $N \vDash o \sim o^*$ holds directly, rather than continuing to search in this direction. If $N \vDash o \sim o^*$ holds, then the dominance query is true. Otherwise, we do not need to consider the o^* direction and it can be considered as pruned.

Boutilier et al. (2004a) claim that their pruning methods for dominance queries also transfer to CP-nets with indifference, though this is not shown explicitly. Additionally, Allen (2013) looked at answering dominance queries for CP-nets with indifference by utilising a SAT solver. In contrast to our work, he considers 'weak dominance', that is, asking whether $N \vDash o \succsim o'$ holds, and does not utilise our assumption to ensure consistency under indifference

Indifference queries such as '$N \vDash o \sim o'$?' (if $r_G(o) = r_G(o')$) hold if and only if o' can be reached from o by a sequence of indifferent flips (analogous to an IFS). As all outcomes considered in searching for such a sequence will have the same rank as o and o' (by Theorem 2.24), we cannot utilise rank pruning here (beyond checking that $r_G(o) = r_G(o')$). However, indifference queries are simple, and can be answered in $O(n)$ time. By our assumption regarding consistency under indifference, changing a variable X between indifferent values cannot affect the preference order over any child of X. Thus, any sequence of indifferent flips cannot change the variable preference orders; suppose we start at o_1 and perform k indifference flips to reach o_2, then the preference order over $X \in V$ must be the same under both $\mathrm{Pa}(X) = o_1[\mathrm{Pa}(X)]$ and $\mathrm{Pa}(X) = o_2[\mathrm{Pa}(X)]$. Thus, at any point in a sequence of indifferent flips starting at o, the possible indifferent flips of $X \in V$ will always be the set of values in $\mathrm{Dom}(X)$ that are indifferent to $o[X]$ under $\mathrm{Pa}(X) = o[\mathrm{Pa}(X)]$ (as $\mathrm{Pa}(X)$ have only changed between indifferent values and so the X preference order is unchanged). Thus, o' can be reached from o via indifferent flips if and only if $o[X]$ and $o'[X]$ are indifferent under $\mathrm{Pa}(X) = o[\mathrm{Pa}(X)]$ for all X where $o[X] \neq o'[X]$. Checking this requirement requires consulting each such $\mathrm{CPT}(X)$ once and, thus, we can answer indifference queries in $O(n)$ time.

Note that least rank improvement ($L(X)$) terms are still well defined in the case of CP-nets with indifferences, as they are defined exclusively in terms of the CP-net structure. Thus, Lemma 2.17 still holds in the case where N has indifference statements. Further, for the case of CP-nets with indifference, we have the following analogous result to Corollary 2.18.

Corollary 2.25. *Let N be a CP-net over variables V, which may have indifference statements within its CPTs. Let o_1 and o_2 be associated outcomes and let $D = \{X \in V \mid o_1[X] \neq o_2[X]\}$. Then,*

$$N \vDash o_1 \succ o_2 \implies r_G(o_1) - r_G(o_2) \geq min_{X \in D}\{L(X)\} > 0.$$

Proof. The proof of this result is very similar to that of Corollary 2.18. If $N \vDash o_1 \succ o_2$, then there is an IFS $o' \rightsquigarrow o$. The rank difference, $r_G(o_1) - r_G(o_2)$, is again the sum of the rank differences of each flip in the IFS. Indifferent flips do not change the rank value so these flips result in a rank difference of zero. Thus, $r_G(o_1) - r_G(o_2)$ is the sum of the rank differences of the strictly improving flips in the IFS.

As $N \vDash o \succ o'$, the IFS $o' \rightsquigarrow o$ must contain at least one strictly improving (not indifferent) flip. Let I be the set of variables, X, such that the IFS includes a strictly improving X flip, then we must have $I \neq \varnothing$. Let X be a variable in I with minimal ancestors. This means that no parent of X has a strictly improving flip in the IFS (as parents of X have strictly less ancestors than X). Thus, any changes to the assignment of $\mathrm{Pa}(X)$ in the IFS are indifferent flips. By our previous assumption about CP-nets with indifference (to ensure consistency in the case of indifference), such changes cannot alter the preference order over $\mathrm{Dom}(X)$. Thus, the preference order over X remains fixed throughout the IFS. As only indifferent and improving flips are allowed in an IFS and we know there is at least one improving X flip, we must have $o[X] \neq o'[X]$. This is because, in order for X to start and end at the same value (and have an improving flip), X would need to flip to a worse value at some point – not possible in an IFS. Therefore, we have $X \in D$ and we have thus proven that at least one variable in D has an improving flip in the IFS.

Thus, we know that $r_G(o_1) - r_G(o_2)$ is the sum of the improving flip rank difference and we know that at least one $X \in D$ has an improving flip in the IFS. The result, $r_G(o_1) - r_G(o_2) \geq \min_{X \in D}\{L(X)\}$, then follows if we prove that the rank difference of any strictly improving Y flip is $\geq L(Y) > 0$. This can be done in an almost identical way to the proof of Corollary 2.18.

As $N \vDash o_1 \succ o_2$, we know that $o_1 \neq o_2$ and so $D \neq \varnothing$. As $L(Y) > 0$ for all variables $Y \in V$ (by Lemma 2.17), we know that $\min_{X \in D}\{L(X)\} > 0$. □

Definition 2.26. Let N be a CP-net over variables V, which may have indifference statements in its CPTs. Let o and o' be associated outcomes and let $D = \{X \in V \mid o[X] \neq o'[X]\}$. The *minimum (entailed) rank difference* of o and o', denoted $M_D(o, o')$, is defined to be

$$M_D(o, o') = \min_{X \in D}\{L(X)\}.$$

By Corollary 2.25, these terms can be used to prune dominance queries more effectively. Suppose we are answering the dominance query $N \vDash o \succ o'$, such that $r_G(o) \geq r_G(o') + M_D(o, o')$ and $o \neq o'$ (otherwise we already know it to be false). Then, starting at o', we build up the search tree as described above. Any outcome, o^*, such that $r_G(o^*) > r_G(o)$ can be pruned, as before. For any outcome, o^*, such that $r_G(o^*) = r_G(o)$, we can use an indifference query rather than further search, as above. Further, we may prune any outcome, o^*, such that $r_G(o^*) < r_G(o)$ and $r_G(o^*) + M_D(o^*, o) > r_G(o)$. This is because, if o^* is on an IFS $o' \rightsquigarrow o$, then either $N \vDash o \succ o^*$ or $N \vDash o \sim o^*$, but as $r_G(o^*) \neq r_G(o)$ we can not have $N \vDash o \sim o^*$ (by Theorem 2.24). However, we can not have $N \vDash o \succ o^*$ by Corollary 2.25, as $r_G(o^*) + M_D(o^*, o) > r_G(o)$. Thus, o^* is not on an IFS $o' \rightsquigarrow o$ and so we do not need to explore this direction further and can prune it from the search tree. In comparison to our previous method for dominance testing on CP-nets with indifference, we now have an additional pruning condition. This will further reduce the size of the dominance query search tree, making it easier to answer. We are also using a stronger initial condition for testing ($r_G(o) \geq r_G(o') + M_D(o, o')$ and $o \neq o'$), which means more queries will be answered immediately, without needing to construct a search tree.

As we mentioned above, $r_G(o) < r_G(o') + M_D(o, o')$ implies that $N \nvDash o \succ o'$. This condition (and the other direction) might be checked when performing ordering queries (as described above) as they can provide additional information. This can lead to a more definitive conclusion or even answer the associated dominance queries. Note that these conditions can be checked in $O(n^4)$ time.

We can again use any consistent ordering to prune the dominance query search tree, as we showed for CP-nets without indifference. Suppose again that we are answering the query $N \vDash o \succ o'$ and that we have a consistent ordering, $\succsim^C$. Suppose we reach outcome o^* in our search tree. If o^* is on an IFS $o' \rightsquigarrow o$, then we must have $N \vDash o \succ o^*$ (and so $o \succ^C o^*$) or $N \vDash o \sim o^*$ (and so $o \sim^C o^*$). If $o^* \succ^C o$, then either $N \vDash o^* \succ o$ or $N \vDash o \bowtie o^*$ and, thus, o^* does not lie on an IFS $o' \rightsquigarrow o$. Hence, if $o^* \succ^C o$, then we do not need to explore that direction further and o^* can be pruned from the tree. If $o^* \sim^C o$, then either $N \vDash o \sim o^*$ or o^* does not lie on an IFS $o' \rightsquigarrow o$. In this case, to improve efficiency, we can answer the indifference query $N \vDash o \sim o^*$ directly in $O(n)$ time, rather than exploring further in this direction. Thus, as for CP-nets with no indifference, we can use any consistent ordering to improve dominance testing efficiency. As CP-nets generally have multiple consistent ordering, this gives us several pruning conditions which can be combined or used in conjunction with other pruning methods, as before, for a more effective pruning schema.

We can actually answer dominance and indifference queries directly from $\succsim^C$ (as we did for CP-nets without indifference). This is because improving and indifferent flips can be identified directly from $\succsim^C$. Thus, for dominance queries we can construct (as well as prune) the search tree and for indifference queries, we can evaluate whether the distinct values are indifferent. The indifferent flips of o are $\{o' \in \Omega | \mathrm{HD}(o, o') = 1 \wedge o \sim^C o'\}$ and the improving flips of o are $\{o' \in \Omega | \mathrm{HD}(o, o') = 1 \wedge o' \succ^C o\}$. Similarly, we can answer (and prune) dominance and indifference queries directly from r_G (and $L(X)$) values as improving and indifferent flips can be similarly identified from relative r_G values. This is not surprising as consistent orderings (and, thus, r_G values) encode the same preference information as the original CP-nets.

In this section, we have shown how our rank definition can be generalised to allow for indifference. Further, we have demonstrated that all of our results now apply or can be adapted to CP-nets with indifference. In particular, we can obtain and update consistent orderings, answer ordering queries, and improve the efficiency of dominance queries in almost exactly the same way as for CP-nets without indifference. We intend to evaluate the performance of rank pruning experimentally in the case of CP-nets with indifference (as we did in §2.4.2 for the case of no indifference) in our future work, which we discuss in §2.6.

2.6 Discussion

In this chapter, we introduced a novel method of quantifying user preference over outcomes, given a CP-net representation of their preferences. These values are called outcome ranks. We have proven these ranks to be an accurate representation of the user preferences as the values reflect all entailed preferences. Thus, these ranks induce a consistent ordering over the outcomes. They can also be used to order any subset of the outcomes consistently with user preferences (more efficiently than the existing method) and obtain a consistent ordering for CP-nets with additional plausibility constraints. We have provided an algorithm that can calculate these outcome ranks in $O(n^4)$ time.

Our outcome ranks can also be used to improve dominance testing efficiency by pruning the associated search tree. We have experimentally evaluated the performance of this pruning method in comparison to (and in combination with) the existing pruning methods. These experiments showed rank pruning to be significantly more effective and efficient than the existing methods. By evaluating the performance of all possible pruning combinations, we also found rank pruning to be a necessary component for an effective pruning schema – the results showed that those combinations without rank pruning performed distinctly worse than those including rank pruning. In particular, we found the combination of rank pruning and suffix fixing to be the most efficient method of answering dominance queries (when functions use their optimal leaf prioritisation method).

In these experiments, we also varied the method of leaf prioritisation used in the search procedures (including two prioritisation heuristics suggested by ourselves – rank and rank + diff. prioritisation). While several prioritisation heuristics have been proposed previously, no experimental analysis of their effect has been performed. Our results found that changing the prioritisation method does not significantly affect dominance testing performance (neither the effectiveness of the pruning nor the overall efficiency). However, as certain pruning methods perform very similarly on average, changing the prioritisation method can be sufficient to affect which pruning methods perform better than others. Thus, leaf prioritisation can be a deciding factor in choosing the optimal pruning schema. For those functions where leaf prioritisation was varied, rank prioritisation was found to be optimal in every case, both in terms of pruning efficacy and overall efficiency. Thus, we have introduced new methods for pruning and leaf prioritisation that both outperform the existing techniques.

All of the dominance testing methods that we compared in our experiments have some initial conditions they check prior to performing the dominance testing search. These conditions are simple to check and, if they hold, prove the dominance query is false – meaning a search is not necessary and, thus, improving dominance testing efficiency. In Appendix C.2, we examined the performance of these initial conditions in our experiments. These results showed our rank condition was significantly stronger than penalty and answered the majority of dominance queries immediately. Combining with the penalty condition results in only a minor improvement, but this is worth it as the conditions are simple to check. We also found that these initial conditions (in particular our rank condition) could provide efficient and reasonably accurate predictions for dominance query results (particularly in the case of binary CP-nets). In the binary case, using both the penalty and rank initial conditions correctly classifies almost 95% of dominance queries as either true or false. In the non-binary case, over 89% of queries are correctly classified. Most of these cases can be correctly classified by the rank condition alone, whereas this is a large improvement upon the performance of the penalty condition alone. However, as initial conditions are simple to check, it is worth checking the penalty condition as well, even though the resulting performance improvement is small. The relative performance of the rank and penalty conditions (and their combination) is unsurprising as these initial conditions are equivalent to their respective pruning conditions. However, in pruning, we found that the cost of implementing penalty pruning was not worth the minor improvement to performance.

We have also demonstrated that both ordering queries and dominance queries can be answered directly from rank values. This is unsurprising as we have also shown that reducing a CP-net to rank values loses no information. More generally, we have shown that reducing a CP-net to any consistent ordering does not result in a loss of information. Further, ordering and dominance queries can be answered directly from any consistent ordering. CP-nets usually have multiple consistent orderings, which answer ordering queries in distinct ways. These methods can be combined in order to yield more information. We have also shown that we can use any consistent ordering to formulate a pruning condition to improve dominance testing efficiency (analogously to how we used outcome ranks to prune dominance query search trees). These methods can also be combined (with each other and existing pruning techniques) to form more effective pruning schemas. Moreover, we showed in Appendix A how any consistent ordering (for CP-nets and preference structures in general) can be iteratively updated in order to be consistent with additional learned preference information. For CP-net consistent orderings, this can be done directly, without consulting the CP-net.

These results illustrate the usefulness of consistent orderings in representing and reasoning with CP-net preferences. While consistent orderings have the drawback of being exponentially large structures, most of the above applications do not require the full ordering. Rather, they need only the ability to assess the relative positions of outcomes in the ordering (for example, by calculating only the outcome ranks for the outcomes of interest). Consistent orderings are also fairly simple to construct, as we, Boutilier et al. (2004a), and Domshlak et al. (2003) illustrate. They are also more directly applicable than CP-nets, as they provide an explicit outcome ordering that is consistent with all known user preferences. Despite these many useful properties, consistent orderings have received little attention in the existing literature.

We generalised our outcome rank definition to be defined in the case of CP-nets with indifference statements in their CPTs, making our results more widely applicable. We have proven that these generalised ranks also reflect all entailed relations and indifferences. Thus, we can directly extend our methods for obtaining consistent orderings for (any subset of) the outcomes and CP-nets with additional plausibility constraints to the case of CP-nets with indifference. Further, these generalised outcome ranks can be used to answer ordering queries and dominance queries and improve dominance testing efficiency (with a rank based pruning condition) in an analogous manner to the outcome ranks for CP-nets without indifference. Again, we can prove that reducing a CP-net (with indifference) to these generalised outcome ranks loses no information. The above results regarding general consistent orderings are also extended to this case.

These generalised outcome ranks are calculable in the same time complexity as the original outcome ranks. This means that all complexity results directly transfer to the case of indifference. In particular, we can consistently order any outcome subset in the same time complexity as the case of no indifference. In contrast, the method by Boutilier et al. (2004a) has unknown complexity in the case of indifference, though they conjecture that it is hard.

In our future work, we would like to extend our outcome ranks so that they are also defined for consistent cyclic CP-nets. Such CP-nets can express more complex preference structures and so this extension would broaden the applicability of our results. Another generalisation we may consider is to define outcome ranks for CP-net extensions such as TCP-nets (CP-nets with additional relative importance statements) (Brafman et al., 2006).

We have shown that many of the applications of outcome ranks can be performed analogously using any consistent ordering. This includes answering ordering queries and implementing pruning conditions to improve dominance testing efficiency. In both of these cases, we have discussed that multiple methods can be combined either to give more information (in the case of ordering queries) or improve effectiveness or efficiency (in the case of dominance query pruning). Thus, in our future work, we would like to see whether we can define a new consistent ordering that would further improve ordering queries (by adding to the information and improving their accuracy as predictors for dominance queries) or dominance testing efficiency (possibly by combining with existing methods). Further, if this ordering can be computed more efficiently than rank values, then we can improve our complexity results in general.

In our dominance testing experiments, the error intervals get fairly wide for the larger n values. Repeating these experiments (for the larger n cases) on a larger query set would give more accurate performance estimates. We would also like to extend any such future experiments to larger n values, in order to see whether the observed patterns continue. It would also be of interest to see how our time elapsed results compare to the other methods of dominance testing, such as the model checking approach by Santhanam et al. (2010), in future experiments. We may also include prefix fixing by Wilson (2004b) in future experiments. Further, we might examine whether least variable leaf prioritisation by Boutilier et al. (2004a) has a more significant effect on performance than the leaf prioritisations considered here.

In §2.5, we extended our rank definitions and theoretical results to the case of CP-nets with indifference statements. In our future work, we would like to perform our experimental evaluations and comparisons of dominance testing performance (and prediction) in this case also.

In Appendix C.2, we evaluate how accurately different initial conditions (of pruning methods) predict dominance query results. These initial conditions are equivalent to ordering query tests (they give us a consistent ordering of the given outcome pair). However, as we discussed in §2.4.1, performing ordering query tests in 'both directions' can yield more information. In particular, this can lead to the conclusion 'o and o' are incomparable'. The initial conditions we considered can only yield two conclusions – the dominance query is false, $N \not\models o \succ o'$ ($o' \succ o$ is a consistent ordering), or $N \not\models o' \succ o$ ($o \succ o'$ is a consistent ordering), in which case we would 'predict' the query to be true. If we checked these initial conditions in 'both directions', then we would have the capacity to classify dominance queries into three classes (true, false, and incomparable), rather than two (true and false). In this case, the only uncertainty would be whether some of those predicted as true were actually incomparable. As we are using more information (and we can now predict all possible scenarios), the prediction accuracy should increase. This will also make them stronger when used as initial conditions as more dominance queries can be answered immediately. This will improve dominance testing efficiency. In our future work, we would like to evaluate how much the prediction accuracy and dominance testing efficiency improves. Such future experiments should also compare the prediction accuracy of other ordering query methods and combinations, rather than considering only those that are used as initial conditions. Note that, technically, any ordering query test could be used as an initial condition, we simply utilise the tests that correspond to the pruning method used. We may also compare these methods to existing techniques for approximately answering dominance queries in our future work.

In Appendix C.1, we described our method of randomly generating CP-nets. From examining the produced structures, we conjecture that the generator favours sparser CP-nets, though the exact CP-net distribution produced by this generator is unknown. The absence of real CP-net data means that we do not know what a realistic CP-net distribution is. However, it would be of interest to analyse the distribution of CP-nets produced by our random generator (for example, the distribution of structural densities produced). This would show what types of CP-net our experimental results are most applicable to. Alternatively, one could repeat our experiments with a CP-net generator that allows more specific parameters. For example, if the generator allowed you to specify structural density, then we could evaluate how density affects dominance testing performance. A uniform CP-net generator would show how the various dominance testing methods perform on average when all CP-nets are equally likely. Thus, another possible direction is to determine how we can perform uniform random generation in practice for sufficiently large n. This could be via making the Allen et al. (2017a) generator work in practice for larger n, though ideally we would also adapt this generator to allow different domain sizes.

Another interesting direction for future experiments is to use real elicited CP-nets rather than simulated ones. This would ensure that a realistic distribution was being used and so we would be able to evaluate how efficient the dominance testing methods are on real world data sets. Further, we could evaluate the predictive power of different ordering queries (initial conditions) more accurately as incomparable cases can be answered definitively (either there is a true preference or indifference) by querying the user. Users could also be asked to evaluate proposed consistent orderings – perhaps one method is more likely to produce accurate (to the user's true preferences) consistent orderings than another.

In Appendix A, we show how consistent orderings can be iteratively updated as additional preference information is learned. However, we require that each additional preference be consistent with all previous information. This is an unrealistic assumption in many contexts due to both natural human inconsistency and a user's preferences changing over time. As we discussed in Appendix A, how to deal with inconsistent information is likely to be context dependent. For example, the context of the problem may dictate how often a user is likely to change their preferences and, thus, how quickly we should adapt their ordering to suit new preference information over (contradictory) historic preferences. We would like to find a method for updating consistent orderings that can handle inconsistent information and can be tuned appropriately to a given context. For example, we might employ an evidence threshold (that may be calibrated), required before new preferences can be incorporated, in order to protect against one-off events. This problem is similar to learning user preferences from data (though with an informed starting point), which we tackle in Chapter 4.

Chapter 3

CP-Net Preprocessing for Efficient Dominance Testing

3.1 Introduction

In Chapter 2, we defined quantitative outcome ranks that reflect the user's level of preference for the given outcome. These qualitative representations have many applications but, primarily, they are used to improve the efficiency of dominance testing. This is done by using rank values to prune the search tree as it is constructed. As we discussed in Chapter 2, in order for CP-nets to be practical models of preference, we must be able to answer dominance queries efficiently. However, they have been shown to be complex problems to answer (see §2.4.1). There has been a lot of work done on improving dominance testing efficiency, as we review in §2.2.3. Many of these methods work by pruning the search tree as it is constructed. Our experimental comparisons in §2.4.2 showed rank pruning to be more effective than existing methods.

In this chapter, we introduce another method for improving dominance testing efficiency. This method works by preprocessing the CP-net in two stages. The result of preprocessing is a reduced CP-net (possibly several) and, thus, a dominance query (or queries) that is much simpler to answer. Our preprocessing works by reducing the number of variables in the CP-net of interest. As the preference information encoded by a CP-net has size exponential in the number of variables, removing variables reduces the size of the problem exponentially. In particular, it reduces the preference graph size exponentially, which is the space one must search when answering dominance queries. Thus, by reducing the number of variables we simplify our dominance query.

The first stage of our preprocessing technique repurposes the suffix fixing result by Boutilier et al. (2004a) and the prefix fixing result by Wilson (2004b) (we provide a proof of completeness for the latter) from pruning methods to preprocessing. We combine these results with the removal of conditionally degenerate parents in order to iteratively identify and remove variables that are unimportant to our dominance query. We find that this reduces the query in a manner distinct from the reduction obtained by using the combination of prefix and suffix fixing as pruning methods. Both methods reduce the problem in ways the other cannot affect and we can improve the efficiency of answering our preprocessed query by using prefix and suffix fixing pruning.

The second stage of our preprocessing identifies whether the resulting query can be separated into smaller sub-queries that are independent of one another. As they are independent, they can be answered separately. Their answers can then be combined to answer the original query. As the size of CP-nets is exponential in the number of variables, partitioning the query in this manner does more than split up the problem, it exponentially reduces it, again. To see this, consider a dominance query over a binary CP-net with six variables. To answer this query, we must search for an IFS over the space of $2^6 = 64$ outcomes. Suppose we can split this into two queries over CP-nets with three variables each. Now we need to conduct two searches over spaces of size $2^3 = 8$. This separated problem requires searching a total space of size 16 rather than 64.

Forward pruning by Boutilier et al. (2004a) is the existing method of CP-net preprocessing to improve dominance testing efficiency. The full details of this method are given in §2.2.3. Rather than removing irrelevant variables, forward pruning removes impossible variable values. We show that our preprocessing and forward pruning remove distinct (though overlapping) sections of CP-nets. That is, both methods remove CP-net aspects that cannot be affected by the other. Via experimental evaluation, we find that our preprocessing is significantly more effective than forward pruning at improving dominance testing efficiency for binary CP-nets. Further, we can combine these two preprocessing methods to obtain an even more effective reduction procedure and, as we find experimentally, a more efficient procedure for answering dominance queries. We show that using these two techniques in combination can enable our method to be more effective than it would be when used individually. Thus, this combination is more powerful than the sum of its individual components.

The other existing methods of improving dominance testing efficiency all provide a method of answering the dominance query faster (usually by constructing a search tree and applying some pruning condition to the branches). Alternatively, preprocessing results in a new (smaller) dominance query (or possibly several queries) that we then need to answer. We can utilise any of these efficient answering methods to complete our dominance testing procedure. In our experiments, we use the most effective pruning method from our §2.4.2 experiments to answer the resulting queries. These preprocessing performance experiments show that using our method of preprocessing can reduce dominance testing time by approximately half and using the combination of our method with forward pruning can reduce time by up to 60%. Thus, our preprocessing significantly improves dominance testing efficiency even when we are already using one of the most efficient answering methods. Hence, by combining this preprocessing with our work on query pruning in Chapter 2, we obtain an even more efficient procedure for dominance testing than those we looked at in Chapter 2.

The rest of this chapter is structured as follows. In §3.2, we explain how our preprocessing procedure works and prove that it does not affect dominance testing completeness. We also show how our procedure can be combined with forward pruning and explain why this combination is more powerful than using both methods individually. In §3.3, we provide an experimental analysis and comparison of the performances of our preprocessing method, forward pruning, and their combination. Finally, in §3.4, we provide a discussion of these results and related future work.

3.2 CP-Net Preprocessing Method

In this section, we present our novel method of simplifying dominance queries by preprocessing the relevant CP-net. This procedure has two stages. First, we iteratively remove variables that are unimportant to the dominance query in question. This is explained in §3.2.1. Second, we partition the resulting query into smaller, independent sub-queries, which can be answered separately. This stage is given in detail in §3.2.2. We refer to our preprocessing as UVRS (unimportant variable removal and separation) preprocessing. In §3.2.3, we explain how UVRS can be combined with the existing preprocessing method, forward pruning (Boutilier et al., 2004a), to make a preprocessing procedure that is more powerful than the simple sum of using UVRS and forward pruning separately.

3.2.1 Removal of Unimportant Variables

Suppose we wish to answer the dominance query $N \vDash o \succ o'$. In this section, we will show how we can simplify this query by iteratively removing variables from N that are unimportant to this query. This results in a smaller CP-net and a simplified query that is equivalent to the original.

Let us first formally define what we mean when we say that a variable is 'unimportant' to a given dominance query.

Definition 3.1. Let N be a CP-net over variables V and let G denote the structure of N. Suppose we are interested in the dominance query $N \vDash o \succ o'$. Let $D = \{X \in V \mid o[X] \neq o'[X]\}$. A variable $Y \in V$ is *important* to the query if either $Y \in D$ or there exists $X, Z \in D$ such that there is a directed path $X \rightsquigarrow Y \rightsquigarrow Z$ in G. That is, Y has both an ancestor and a descendant in D. If $W \in V$ is not important, then we say it is *unimportant* to the query.

If a variable, $X \in V$, is unimportant to our query, then we must have $X \notin D$. Thus, X takes the same value in both o and o', say $o[X] = o'[X] = x$. We remove X from the CP-net by fixing it at this value, $X = x$, as follows. First, we remove all unimportant variables and any adjacent edges from the structure. We also remove their CPTs. This is because these variables are now fixed values and, thus, no longer variables in our problem. Second, to ensure that the resulting CP-net is fully defined, we must adjust the CPT of any variable that has lost a (unimportant) parent. Suppose Z was a parent of Y in N but it has now been removed. Then Z must be an unimportant variable and we have $o[Z] = o'[Z] = z$, for some $z \in \text{Dom}(Z)$. We remove all rows of $\text{CPT}(Y)$ that correspond to a parent assignment where $Z \neq z$. We do this for every parent Y has lost. The resulting CPT contains exactly the rows corresponding to parent assignments in which all unimportant variables take their fixed values. We restrict to these rows as they are now the only possible parental assignments. As only the remaining parents are allowed to vary, this CPT corresponds to a well defined CPT for the new, reduced, parent set of Y, as we wanted. To obtain this CPT, we simply ignore the removed parent assignments (which are fixed). Thus, the resulting structure is a smaller, fully defined, acyclic CP-net. We illustrate our method of identifying and removing unimportant variables in the following example.

Example 3.2. Let N be a CP-net with the structure given in Figure 3.1 (ignore the variable colourings for now). For clarity, we shall denote the i^{th} variable of N by X_i (for $1 \leq i \leq 10$). We assume all variables to be either binary or tertiary. For ease, let binary variables have domain $\{0, 1\}$ and tertiary variables

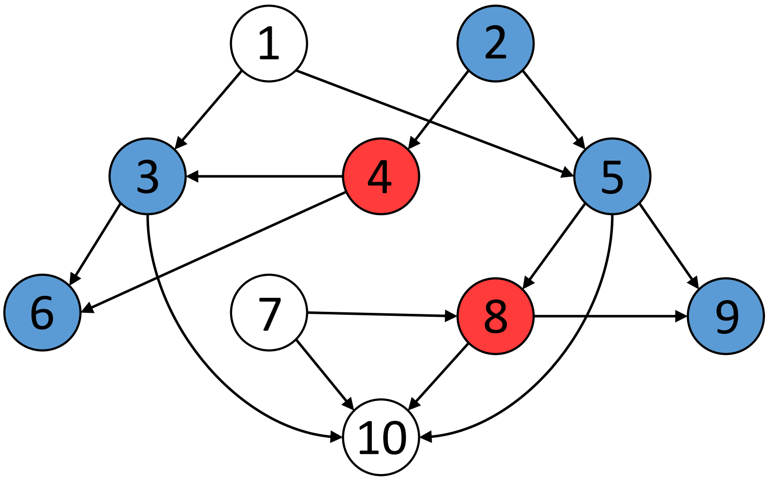


Figure 3.1: CP-Net Structure with Unimportant Variables

Variable	1	2	3	4	5	6	7	8	9	10
Binary/Tertiary	B	B	B	B	T	T	B	B	T	T
o	1	1	0	0	2	0	1	1	1	0
o'	1	0	1	0	0	1	1	1	2	0

Table 3.1: Dominance Query Example

have domain $\{0, 1, 2\}$. Thus, for this example, outcomes may be represented by length 10 vectors with entries in $\{0, 1, 2\}$. Table 3.1 shows which variables are binary and which are tertiary. It also gives two outcomes, o and o'.

Suppose we wish to answer the query $N \vDash o \succ o'$. By Definition 3.1, the first criterion for variable importance is being in the set $D = \{X \in V \mid o[X] \neq o'[X]\}$. In this example, we have $D = \{X_2, X_3, X_5, X_6, X_9\}$. The set D is shaded blue in Figure 3.1. The second criterion for importance is being on a directed path between two members of D. These variables are shaded red in Figure 3.1. Thus, the coloured variables in Figure 3.1 are exactly the set of variables that are important to our query. The unimportant variables are, therefore, $U = \{X_1, X_7, X_{10}\}$ (the variables that are not coloured).

We want to remove the variables in U by fixing them at the values they take in both o and o' ($X_1 = 1$, $X_7 = 1$, $X_{10} = 1$). Removing these variables and their adjacent edges results in the structure given in Figure 3.2 and the reduced outcomes (that is, the reduced query) given in Table 3.2. We also remove the CPTs of the variables in U, though this is not illustrated here as it is a simple case of deletion.

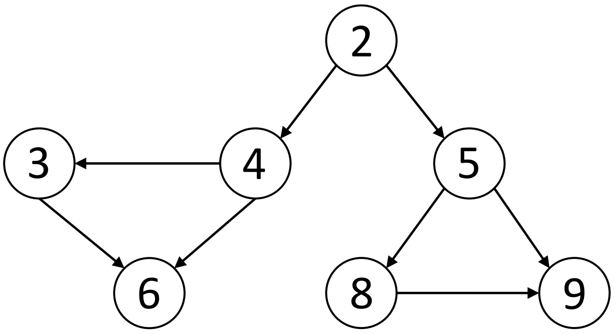


Figure 3.2: Reduced CP-Net Structure

Variable	2	3	4	5	6	8	9
Binary/Tertiary	B	B	B	T	T	B	T
$o[V \backslash U]$	1	0	0	2	0	1	1
$o'[V \backslash U]$	0	1	0	0	1	1	2

Table 3.2: Reduced Dominance Query

Out of the remaining variables, X_3, X_5, and X_8 have lost parents and, thus, need their CPTs adjusting. We do not provide the full set of example CPTs as they are large and mostly unnecessary. However, we give the relevant CPTs and their adjustments below for illustration.

CPT(X_3):

X_1	X_4	Preference
0	0	$0 \succ 1$
0	1	$1 \succ 0$
1	0	$1 \succ 0$
1	1	$0 \succ 1$

$\xrightarrow{X_1=1}$

X_4	Preference
0	$1 \succ 0$
1	$0 \succ 1$

CPT(X_5):

X_1	X_2	Preference
0	0	$1 \succ 2 \succ 0$
0	1	$0 \succ 1 \succ 2$
1	0	$2 \succ 1 \succ 0$
1	1	$2 \succ 1 \succ 0$

$\xrightarrow{X_1=1}$

X_2	Preference
0	$2 \succ 1 \succ 0$
1	$2 \succ 1 \succ 0$

CPT(X_8):

X_5	X_7	Preference
0	0	$1 \succ 0$
0	1	$1 \succ 0$
1	0	$0 \succ 1$
1	1	$1 \succ 0$
2	0	$0 \succ 1$
2	1	$1 \succ 0$

$\xrightarrow{X_7=1}$

X_5	Preference
0	$1 \succ 0$
1	$1 \succ 0$
2	$1 \succ 0$

In each case, we have obtained the new CPT by restricting to the rows in which the unimportant variables take their fixed values (in the parental assignment). As these parents are now fixed, we eliminate them from the CPT, giving the CPTs on the right. These CPTs depend only on the new (reduced) parent sets, making them well defined for the new structure. The CPT of any variable that did not lose a parent remains well defined in the new structure. Thus, we now have a reduced, well defined CP-net over the important variables only and an associated reduced dominance query.

The CP-net that we obtain by removing unimportant variables in this manner encodes the user's preferences under the constraint $U = o[U] = o'[U]$, where U denotes the unimportant variables. Let M denote the CP-net obtained by removing the unimportant variables from N. Let Ω_N denote the outcomes associated with N and Ω_M the outcomes associated with M. By construction, Ω_M is the Cartesian product of the domains of $V \backslash U$. These correspond to all possible outcomes under the constraint $U = o[U] = o'[U]$ (we simply ignore the fixed assignment to U). We interpret the outcome $\alpha \in \Omega_M$ to represent $\alpha o[U] \in \Omega_N$. That is, we assume that $\alpha \in \Omega_M$ implies $V \backslash U = \alpha$ and $U = o[U]$. By this interpretation, M is a preference structure over the outcomes of N that satisfy the constraint $U = o[U]$. The following proposition shows that M encodes exactly the preferences implied by N under this constraint.

Proposition 3.3. *Let N be a CP-net over variables V and let o and o' be associated outcomes. Let $U \subseteq V$ denote the variables that are unimportant to the query $N \vDash o \succ o'$. As the variables in U are unimportant, we must have $o[U] = o'[U]$. Let M be the CP-net obtained by removing U from N as described above (by fixing $U = o[U]$). Let C denote the constraint $U = o[U]$ and let N_C denote the CP-net N with this additional plausibility constraint. Let o_1 and o_2 be any two outcomes associated with N that obey constraint C, that is, $o_1[U] = o_2[U] = o[U]$. Then $N_C \vDash o_1 \succ o_2$ if and only if $M \vDash o_1[V \backslash U] \succ o_2[V \backslash U]$.*

Proof. See Appendix E.7.

As we discussed above, the outcomes of M can be considered as the outcomes of N satisfying the constraint $U = o[U](= o'[U])$. This proposition shows that the preferences implied by M over these outcomes are equivalent to the entailments implied by N under this constraint. Thus, reasoning with M is equivalent to reasoning with N under the constraint $U = o[U]$. This is as we would expect as M is obtained from N by fixing U at the values they take in o. Now that we know the preferences encoded by our reduced CP-net, it remains to prove that the reduced dominance query, $M \vDash o[V \backslash U] \succ o'[V \backslash U]$, is equivalent to the original, $N \vDash o \succ o'$. It is sufficient to prove that, if there exists an IFS $o' \rightsquigarrow o$ in N, then there exists an IFS $o' \rightsquigarrow o$ that does not change the value of any unimportant variable. We will formally prove that this is a sufficient condition later on. Informally, this result shows that, if the preference $o \succ o'$ holds, then it holds under the constraint that U is fixed ($U = o[U]$). The other direction of this result is trivial as an IFS that does not change U is still an $o' \rightsquigarrow o$ IFS. Thus, the preference $o \succ o'$ holds if and only it holds under the constraint $U = o[U]$. As we have seen, M encodes user preference under this constraint and so the original query is equivalent to its reduction to M.

We now suppose that $N \vDash o \succ o'$ holds and, thus, there exists some IFS $o' \rightsquigarrow o$ in G_N. To prove that there is an IFS $o' \rightsquigarrow o$ that does not change any variables in U we use two results – suffix fixing by Boutilier et al. (2004a) and prefix fixing by Wilson (2004b). Suppose $\{X_1, ..., X_n\}$ is any topological ordering of the variables (with respect to the structure of N) and let $\alpha \in \Omega_N$ be any outcome. We define the k^{th} *suffix* of α as $o[X_k, X_{k+1}, ..., X_n]$. Boutilier et al. (2004a) proved that if o and o' have a matching suffix and there is an IFS $o' \rightsquigarrow o$, then there is an IFS $o' \rightsquigarrow o$ that preserves this suffix (that is, that does not change any variable value in the matching suffix). Boutilier et al. (2004a) propose utilising this result to prune the dominance query search tree as it is constructed. This is done by pruning any improving flips that do not preserve a matching suffix with o. Let S denote the set of variables, Y, such that Y and all descendants of Y take the same values in both o and o'. As $Y \in S$ implies all descendants of Y are in S, it is possible to construct a topological ordering in which S comes at the end. For this topological ordering, S constitutes a matching suffix of o and o'. This is in fact the largest matching suffix as it is the union of all matching suffices. The suffix fixing result by Boutilier et al. (2004a) then proves that there exists an IFS $o' \rightsquigarrow o$ that does not change any variable in S.

Prefix fixing was proposed by Wilson (2004b) to be used in conjunction with suffix fixing to improve dominance testing efficiency for CP-theories (a strict generalisation of CP-nets). Let P denote the set of variables, Y, such that Y and all ancestors of Y take the same values in both o and o'. Wilson (2004b) suggests that, if there is an IFS $o' \rightsquigarrow o$ in a CP-theory, then there is an $o' \rightsquigarrow o$ IFS that does not change any variable in P or in S. In fact, the author claims that all $o' \rightsquigarrow o$ IFSs preserve P. Consequently, when searching for an IFS, any improving flips that change variables in either P or S can be pruned. However, Wilson (2004b) does not provide a proof of these claims in either the CP-theory case or for CP-nets specifically. The suffix fixing result above, by Boutilier et al. (2004a), proves that there does exist an IFS preserving S in the case of CP-nets. We prove in the following proposition that the claim that all $o' \rightsquigarrow o$ IFSs preserve P also holds in the CP-net case. We do not address whether their claims hold in the more general case of CP-theories as we are only interested in CP-nets here.

Proposition 3.4. *Let N be a CP-net over variables V and let o and o' be any two associated outcomes. Let P denote the set of variables, Y, such that Y and all ancestors of Y take the same values in both o and o'. If there is an IFS $o' \rightsquigarrow o$ in N, then no variable in P is flipped in this IFS.*

Proof. See Appendix E.8.

Recall that we are assuming there exists some $o' \rightsquigarrow o$ IFS in N. By the Boutilier et al. (2004a) suffix fixing result, there exists some IFS that does not flip any variable in S. By the above result, all $o' \rightsquigarrow o$ IFSs preserve P throughout. Thus, there exists an $o' \rightsquigarrow o$ IFS that does not flip any variable in $S \cup P$. The following result shows that $U = S \cup P$. From this we can conclude that, if there is an $o' \rightsquigarrow o$ IFS, then there is an $o' \rightsquigarrow o$ IFS that does not change the value of any unimportant variable.

Proposition 3.5. *Let N be a CP-net over variables V and let o and o' be any two associated outcomes. Let $U \subseteq V$ denote the variables that are unimportant to the query $N \vDash o \succ o'$. Let S denote the set of variables, Y, such that Y and all descendants of Y take the same values in both o and o'. Let P denote the set of variables, Z, such that Z and all ancestors of Z take the same values in both o and o'. Then $U = S \cup P$.*

Proof. Let $D = \{X \in V | o[X] \neq o'[X]\}$. Let $X \in V$ be any variable. Let $\text{Dec}(X)$ denote the set of descendants of $X \in V$ in the structure of N and let $\text{Anc}(X)$ denote the ancestors. By the definition of unimportant variables (Definition 3.1), we have the following equivalences:

$$\begin{aligned} X \in U &\iff (X \notin D) \wedge (\mathrm{Anc}(X) \cap D = \varnothing \vee \mathrm{Dec}(X) \cap D = \varnothing) \\ &\iff (o[X] = o'[X]) \wedge \\ &\qquad (\forall Y \in \mathrm{Anc}(X), o[Y] = o'[Y] \vee \forall Y \in \mathrm{Dec}(X), o[Y] = o'[Y]) \\ &\iff (o[X] = o'[X] \wedge \forall Y \in \mathrm{Anc}(X), o[Y] = o'[Y]) \vee \\ &\qquad (o[X] = o'[X] \wedge \forall Y \in \mathrm{Dec}(X), o[Y] = o'[Y]) \\ &\iff (X \in P) \vee (X \in S) \\ &\iff X \in P \cup S. \end{aligned}$$

Thus, U and $S \cup P$ must be the same set of variables, as we wanted to prove. □

The combination of these results shows that, if there is an $o' \rightsquigarrow o$ IFS, then there is an IFS that does not change the value of any variable in U, as we wanted to show. That is, when searching for an IFS, it is sufficient to consider only those sequences that keep the unimportant variables fixed. As M encodes user preferences under the constraint that U is fixed, this shows that answering the reduced query over M is equivalent to answering the original query. This is proven formally by the following corollary.

Corollary 3.6. *Let N be a CP-net over variables V and let o and o' be two associated outcomes. Let $U \subseteq V$ be the set of variables that are unimportant to the query $N \vDash o \succ o'$. Let M be the CP-net obtained from N by removing U (by fixing $U = o[U] = o'[U]$). Then we have*

$$N \vDash o \succ o' \iff M \vDash o[V \backslash U] \succ o'[V \backslash U].$$

Proof. We know that $N \vDash o \succ o'$ if and only if there is a directed path (that is, an IFS) $o' \rightsquigarrow o$ in the preference graph of N, G_N. Let S denote the set of variables, Y, such that Y and all descendants of Y take the same values in both o and o'. Let P denote the set of variables, Z, such that Z and all ancestors of Z take the same values in both o and o'. From the Boutilier et al. (2004a) suffix fixing result and Proposition 3.4, we know that, if there is an $o' \rightsquigarrow o$ IFS, then must be an IFS that preserves both S and P (that is, does not change the value of any variable in $S \cup P$). By Proposition 3.5, this means there must be an $o' \rightsquigarrow o$ IFS that preserves U. Trivially, if there is an $o' \rightsquigarrow o$ IFS that preserves U, then there is an $o' \rightsquigarrow o$ IFS. Thus, $N \vDash o \succ o'$ if and only if G_N contains a directed path $o' \rightsquigarrow o$ that does not change the value of any variable in U. That is, a directed path in which every outcome (node) satisfies $U = o[U]$.

Let C be the constraint $U = o[U]$. Let N_C be the CP-net N with this additional plausibility constraint. The preference graph of N_C is the induced graph of G_N

on the outcomes that obey C. We denote this preference graph G_{N_C}. Thus, G_N contains a directed $o' \rightsquigarrow o$ path in which every outcome satisfies $U = o[U]$ if and only if G_{N_C} contains a directed path $o' \rightsquigarrow o$. Thus, we now have shown that $N \models o \succ o'$ if and only if G_{N_C} contains a directed path $o' \rightsquigarrow o$. That is, $N \models o \succ o'$ if and only if $N_C \models o \succ o'$. By proposition 3.3, this holds if and only if $M \models o[V \backslash U] \succ o'[V \backslash U]$, as we wanted. □

We have now proven that our reduction of the CP-net is equivalent to fixing the unimportant variables, $U = o[U]$. Further, we have used suffix and prefix fixing to show that this constraint (reduction) does not affect the dominance query. That is, answering the reduced query is equivalent to answering the original.

Example 3.7. Consider the CP-net from Example 3.2. By Corollary 3.6, answering the reduced query ($M \models o[V \backslash U] \succ o'[V \backslash U]$, given in Table 3.2) over seven variables is equivalent to answering the original query over ten variables. For a more accurate picture of how this has reduced the complexity of our problem, let us consider the number of outcomes associated with the CP-nets – this is the space we must search for an IFS in order to answer the queries. The original CP-net, N, had 5,184 outcomes. The reduced CP-net, M, has 432, so we have reduced the size of the problem by over 90% already.

In general, if a CP-net has variables V, then the outcome space has size $|\Omega_N| = \prod_{X \in V} |\text{Dom}(X)|$. If we remove variables U, then the new outcome space has size

$$|\Omega_M| = \prod_{X \in V \backslash U} |\text{Dom}(X)| = \frac{|\Omega_N|}{\prod_{X \in U} |\text{Dom}(X)|}.$$

Thus, removing the unimportant variables reduces the outcome space by a factor of $\prod_{X \in U} |\text{Dom}(X)|$. In the binary case, this is $2^{|U|}$. In the non-binary case, it can be even larger. Thus, by removing variables, we reduce the outcome space by an exponential factor (in the number of variables removed). Therefore, the space we need to search over to find an IFS is exponentially smaller, making our problem much simpler to answer. This reduction factor will continue to grow exponentially as we remove more variables as we describe below.

As we claimed at the start of this section, the removal of unimportant variables can be performed iteratively. This is because the CPT adjustments that accompany the removal of unimportant parents can result in degenerate parent-child relations, even if all parents are non-degenerate in N. A degenerate parent is a parent whose value does not affect the preference over the child. If X is a degenerate parent of Y, then the relation $X \rightarrow Y$ in the CP-net structure contributes no information as Y is not preferentially dependent upon X. Thus, removing such edges does not change the preference structure encoded by the CP-net. That is, removing degenerate edges does not semantically alter the CP-net (the preference graph is not changed by such removals). Note that only variables that lose a parent in the reduction of N have their CPTs adjusted, so we only need to check for degeneracy in the remaining parents of such variables.

Adjusting the CPTs can cause remaining parents to become degenerate as this process restricts the CPT to those rows in which $U = o[U]$ in the parental assignment. Such a restriction can eliminate the cases where the child preference depends upon the parent. If removing the unimportant variables from N results in X being a degenerate parent of Y, we call X a *conditionally degenerate* parent. This is because X becomes a degenerate parent under the condition that $U = o[U]$ (as it is degenerate in M). All such degenerate parent-child relations can be removed from the structure of M without affecting the preference structure M represents. The CPTs of the children that lose degenerate parents can be reduced trivially as their preference order is not dependent upon the removed parent. This is different from the previous CPT adjustments as the removed parent does not need to be fixed to a specific value.

Removing the conditionally degenerate relations changes the structure of M, though it still implies exactly the same set of preferences. This change in structure can result in variables that are unimportant to the reduced query. Note that, after the initial removal of unimportant variables from N, all remaining variables are important to the reduced query. This is because D is the same set for both queries and the removal of unimportant variables from N does not remove any edges between important variables. Thus, any variable on a path between D variables in N is on the same path in M. Therefore, any variable that was important in N is also important in M. As all unimportant variables in N are removed, this means that all variables in M are important to the reduced query. Thus, all conditionally degenerate relations that we remove from M are between pairs of important variables. Removing such edges changes the structure and can result in unimportant variables. Note that variables in D are important regardless of structure. Thus, the variables that become unimportant are those that were on a path between D variables but, after the removal of degenerate relations, are no longer on any such path (and are not in D themselves).

Suppose that removing degenerate parents results in variables that are unimportant to the current query, $M \vDash o[V \backslash U] \succ o'[V \backslash U]$. Such variables can then be removed from M to give a smaller CP-net and dominance query. This reduced query will be equivalent to $M \vDash o[V \backslash U] \succ o'[V \backslash U]$ (and, thus, to $N \vDash o \succ o'$) by the same reasoning as $M \vDash o[V \backslash U] \succ o'[V \backslash U]$ is equivalent to $N \vDash o \succ o'$. Let us call this new reduced CP-net P. We can then repeat this process with P. If there are any new conditionally degenerate relations, these can be removed from the structure of P. If this results in more unimportant variables, we remove them to produce an even further reduced CP-net and query that is still equivalent to the original, though much simpler to answer. We continue to apply this procedure until we reach a CP-net that has no unimportant variables after any degenerate relations are removed. The result is a simplified dominance query that is equivalent to our original problem.

At each simplification stage we are identifying variables that are unimportant to the current query and fixing them to reduce the problem. Thus, the same reasoning as used with the first reduction can be applied to show the equivalence of the increasingly reduced queries with the original. However, we can also consider this process as repeatedly identifying additional constraints that simplify our problem further without affecting completeness. After each simplification, the next set of unimportant variables can be considered ‘conditionally unimportant’ to our query under the current simplifying constraints.

Example 3.8. We now illustrate how to perform iterative removal of unimportant variables, using our running example. In Example 3.2, after removing the unimportant variables and adjusting the CPTs, the resulting CPT(X_5) and CPT(X_8) have invalid (degenerate) parents, whereas the resulting CPT(X_3) remains non-degenerate. Changing the value of X_2 no longer affects preference over X_5. Thus, the parent-child relationship $X_2 \rightarrow X_5$ is now degenerate. Similarly, the relation $X_5 \rightarrow X_8$ is also now degenerate. Notice that this occurred despite the fact that all parent-child relationships were valid in N. These relations become degenerate under the condition $U = o[U]$ (they are conditionally degenerate). Note that we only have to check the adjusted CPTs for conditional degeneracy as all other CPTs remain the same as in N.

As we have discussed above, degenerate edges do not add anything to a CP-net and can, thus, be removed without semantically changing the CP-net or any associated dominance queries. Thus, we remove these edges from the reduced CP-net, M (Figure 3.2). The resulting structure is given on the left hand side of Figure 3.3. We then reduce the CPTs of the variables that have lost a parent by removing the parent from the CPT. This is a trivial reduction as degenerate parents do not affect the child’s preference. Thus, all values of the degenerate parent result in the same preference order in all cases. The reduced CPT is obtained simply by ignoring the removed parent(s). For example, by removing the edge $X_2 \rightarrow X_5$, CPT(X_5) will become the unconditional preference $2 \succ 1 \succ 0$. This was the preference over X_5 given by every possible assignment to X_2. This is different to the previous CPT adjustments as we do not have to fix the invalid parent (X_2) at any particular value – all values result in the same preference rule.

As we have removed only degenerate edges, there has been no change to the preference structure represented by the CP-net. Thus, our reduced dominance query has not changed and remains equivalent to the original. We now attempt to identify and remove further unimportant variables. The set D has not changed as they are not affected by unimportant variable removal. The variables in D are coloured blue in our current structure (left hand side of Figure 3.3). These are important to our reduced query by definition. The second criterion for importance is to be on a directed path between two variables in D. Variables that meet this condition are coloured red. The remaining variables without a colour are unimportant to our reduced query by definition. Thus, X_8 is unimportant and can be removed from the structure by the same reasoning as before. Removing X_8 leaves us with the structure on the right hand side of Figure 3.3.

We have now reduced our original dominance query to an equivalent query over a CP-net with six variables and 216 outcomes. This is about 4% of the size of the original outcome space to be searched. The reduced query is given in Table 3.3 (where U denotes the total set of unimportant variables removed). The removal of variable X_8 requires only the adjustment of CPT(X_9) (which we omit from this example). Thus, only the relation $X_5 \rightarrow X_9$ has the possibility of being conditionally degenerate. If so, we remove it from the structure. In this new structure, every variable except X_4 is in D as the set D is not affected by this process. As variables in D are permanently important, only X_4 can possibly become unimportant to our query. However, as X_4 is on a path between X_2 and X_6 (or X_3), it will remain important as the only possible degenerate edge to be removed is $X_5 \rightarrow X_9$. Thus, all variables remain important and so we cannot identify any further unimportant variables. We therefore cannot reduce the CP-net any further via this method and so we move onto our second preprocessing stage. However, for other dominance query examples, we can iterate the process of removing unimportant variables several more times.

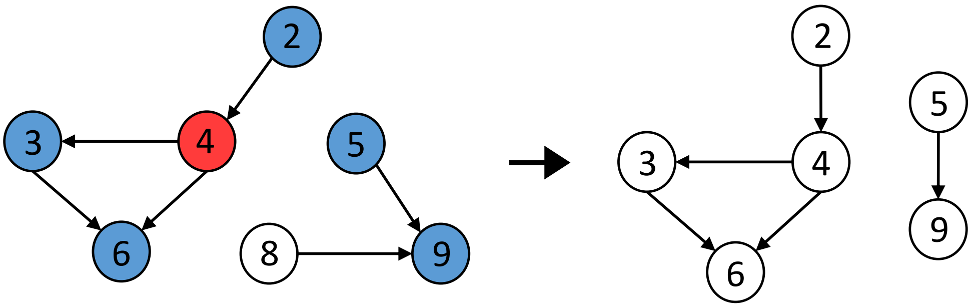


Figure 3.3: Second Iteration of Removing Unimportant Variables

Variable	2	3	4	5	6	9
Binary/Tertiary	B	B	B	T	T	T
$o[V \backslash U]$	1	0	0	2	0	1
$o'[V \backslash U]$	0	1	0	0	1	2

Table 3.3: Twice Reduced Dominance Query

Applying suffix fixing and prefix fixing as pruning measures, as Boutilier et al. (2004a) and Wilson (2004b) suggest, one would prune all improving flips in the IFS search that change a variable in $P \cup S$ (the matching prefix and suffix variables for o and o'). By Proposition 3.5, this is equivalent to removing the unimportant variables from the original CP-net, N, by fixing their values. However, from here our preprocessing method diverges from prefix and suffix fixing pruning and the two reduce dominance query complexity in distinct ways.

The prefix and suffix fixing results show that, when searching for an IFS, we only need to explore directions that preserve any matching suffix and prefix (that is, that do not change any of $P \cup S$). A more effective way of using this result is to apply it to every new outcome reached by the IFS search. Suppose we reach the outcome o^*. We have pruned any directions that change $P \cup S$, so o^* must have the same or more matching prefix and suffix variables with o than o' does. Let $P' \cup S'$ be the matching prefix and suffix variables of o^* and o. When continuing the search from o^*, we are essentially searching for an IFS $o^* \rightsquigarrow o$. Thus, by the prefix and suffix fixing results, when searching from o^* onwards, we can now prune any direction that does not preserve $P' \cup S'$, where $P \cup S \subseteq P' \cup S'$. Thus, by re-evaluating the matching prefix and suffix at each new outcome, we obtain stronger and stronger pruning conditions as we move through the search.

Eventually, as our search continues, the matching suffix and prefix may contain variables such that $o'[X] \neq o[X]$ originally. Thus, this method will prune flips of variables in D, important variables. In general, this method can prune flips of any variable, including those that we cannot remove by iterative unimportant variable removal. As our preprocessing cannot remove these variables, these flips could not be removed (pruned) by our preprocessing procedure. Note that, as preprocessing removes variables (by fixing them) at the beginning (before dominance testing commences), it prunes all possible flips of these variables from the entire search. Whereas these stronger pruning conditions offered by suffix fixing and prefix fixing are only in place for certain sections of the search – once the larger matching prefix or suffix is obtained.

Conversely to the above, our preprocessing method can also remove aspects of the dominance query that are not affected by using prefix and suffix fixing alone. Any conditionally unimportant variables (unimportant variables identified after the first iteration) that we identify and remove from the CP-net are removed from the entire search (no flips of these variables are considered). As these variables are not in the original matching prefix or suffix, $P \cup S$, prefix and suffix fixing can only prune the flips of such variables for certain sections of the search. In particular, such directions can only be pruned once an outcome is obtained that has the conditionally unimportant variable in its matching prefix or suffix with o. As the two reduce dominance query complexity in distinct ways, they can be used together to be more effective – that is, using suffix and prefix fixing pruning when answering a preprocessed query will reduce the query complexity further.

Remark. One can consider the iterative removal of unimportant variables to be a stronger version of prefix and suffix fixing. It is stronger due to the addition of fixing conditionally unimportant variables (not just the original unimportant variables). Thus, one might consider applying this stronger version as a pruning condition, as we do with prefix and suffix fixing. Like prefix and suffix fixing, the iterative removal of unimportant variables would be a successively stronger condition as the search progressed as the number of unimportant variables can only grow (as previously identified unimportant variables remain fixed). However, performing the iterative removal of unimportant variables at each new outcome and storing the reduced structures is likely to result in a high computational cost. Recall that the existing pruning methods generally have linear or polynomial cost in n. Further, we could not use the second stage of our preprocessing (separation of queries) in this manner as it does not prune or reduce the search tree, but rather, turns it into several smaller search trees. Thus, we decided to apply our method as a preprocessing procedure, rather than a successively stronger pruning condition. This means that we apply it only to the original query (not for each outcome obtained in the subsequent dominance query search). In our experimental results, we find that our preprocessing significantly improves dominance testing efficiency. We conjecture that applying it as a pruning method would not improve efficiency by much. In fact, the additional computational cost is likely to reduce overall efficiency.

We have now shown how we can iteratively remove unimportant variables in order to reduce the complexity of a given dominance query. The resulting, reduced query is equivalent to the original. However, in general it is far simpler to answer as we have exponentially reduced the outcomes space we need to search over to find an IFS (by fixing the variables we removed). We have also shown that, despite being based on the same theoretical results, this method reduces dominance queries in a distinct manner from prefix and suffix fixing pruning (in fact, they can be combined for a more efficient procedure). Once the CP-net and the dominance query have been reduced via unimportant variable removal, we then apply the second preprocessing stage – separation into independent sub-queries – to simplify the query further. This stage is described in the following section.

3.2.2 Separation of Connected Components

In this section, we describe the second stage of our preprocessing method. Once a query has been reduced by iteratively removing unimportant variables, we then partition it into independent sub-queries that can be answered separately. Each sub-query will be much easier to answer and the overall complexity of the problem will be reduced exponentially again. Moreover, as these sub-queries are independent, they can be answered simultaneously for further improved efficiency.

If a CP-net's structure is disconnected, then the variables in one connected component cannot impact preferences over variables in another component. This is because the CP-net structure represents preferential dependence. Thus, two unconnected variables cannot be preferentially dependent. Hence, if we wish to answer the query $N \models o \succ o'$, then we can address each connected component of N individually, as the preference structure over each component is independent. We prove this formally in the following proposition. First, we must define the sub-CP-nets of N that are represented by its connected components.

Definition 3.9. Let N be an acyclic CP-net with structure G. Let G' be a connected component of G. Let N' be a CP-net with structure G'. For every variable, $X \in G'$, the CPT for X in N' is the same as in N. As G' is a connected component, if $X \in G'$, then all parents of X are also in G'. Thus, any $X \in G'$ has the same parents as in G. Thus, the CPTs from N are also appropriate for the structure of N' and so N' is a well defined, acyclic CP-net. We call N' the *induced sub-CP-net of* N *over* G'.

These sub-CP-nets essentially partition the preference structure encoded by N. This is because the preference structure over each connected component is independent of the others. Thus, evaluating a dominance query for each sub-CP-net separately is equivalent to evaluating the query for the whole CP-net.

Proposition 3.10. *Let N be a CP-net over variables V with structure G. Let $G_1, G_2, ..., G_m$ be the connected components of G. Let $V_i \subseteq V$ denote the variables in G_i. Let N_i be the induced sub-CP-net of N over G_i. Let o and o' be any two outcomes associated with N such that $o \neq o'$ (otherwise the dominance query is trivially false). Then we have*

$$N \vDash o \succ o' \iff \forall i \ \left(o[V_i] = o'[V_i] \vee N_i \vDash o[V_i] \succ o'[V_i]\right).$$

Proof. See Appendix E.9.

This result can be simplified in the case where N has had all unimportant variables iteratively removed, as will be the case in our preprocessing procedure. In this case, every connected component must differ between o and o' on at least one value. If a whole connected component is the same in both o and o', then the variables in this component would be unimportant to our query and, thus, would have been removed previously. This is proven below.

Corollary 3.11. *Let N be a CP-net and let o and o' be associated outcomes such that $o \neq o'$. Let M be the CP-net over variables V obtained by iteratively removing variables from N that are unimportant to the query $N \vDash o \succ o'$. Let U denote the total set of unimportant variables removed. Let G be the structure of M and let $G_1, ..., G_m$ be the connected components of G. Let $V_i \subseteq V$ denote the variables in G_i. Let M_i be the induced sub-CP-net of M over G_i. Then we have*

$$M \vDash o[V \backslash U] \succ o'[V \backslash U] \iff \forall i \ \ M_i \vDash o[V_i] \succ o'[V_i].$$

Proof. First note that $o \neq o'$ implies $o[V\backslash U] \neq o'[V\backslash U]$. All variables removed from N are unimportant to the query $N \vDash o \succ o'$. Thus, every $X \in U$ must satisfy $o[X] = o'[X]$. Therefore, we have $o[U] = o'[U]$. If $o[V\backslash U] = o'[V\backslash U]$, then we would have $o = o'$, a contradiction, so we must have $o[V\backslash U] \neq o'[V\backslash U]$. The result $o[U] = o'[U]$ also shows that $U \neq V$ and so M is a non-trivial CP-net with variables $V\backslash U \neq \varnothing$. The result now follows directly from Proposition 3.10 if we can show that $o[V_i] \neq o'[V_i]$ holds for all V_i.

Suppose, for the sake of contradiction, that there exists some M_i such that $o[V_i] = o'[V_i]$. This means that, for every variable, X, in the connected component G_i, we have that $o[X] = o'[X]$. Let X be a variable in G_i (there must be at least one as we assume connected components to be non-empty). Let D be the set of variables in M (that is, in $V\backslash U$) that take different values in $o[V\backslash U]$ and $o'[V\backslash U]$. As $o[X] = o'[X]$, we know that X is not in D. As G_i is a connected component, all ancestors of X (in G) are in G_i also. Thus, for every $Y \in \text{Anc}(X)$, we have $o[Y] = o'[Y]$. This shows that no ancestors of X are in D either (by a similar argument, we can also show that no descendants of X are in D). Thus, by definition, X is unimportant to the query $M \vDash o[V \backslash U] \succ o'[V \backslash U]$. This is a contradiction as the iterative removal of unimportant variables continues until it reaches a CP-net with no unimportant variables. Thus, X would have been removed by this process. As we have reached a contradiction, we have proven that there is no sub-CP-net M_i such that $o[V_i] = o'[V_i]$. Thus, we have shown that $o[V_i] \neq o'[V_i]$ for all i and so our result follows from Proposition 3.10. □

The process of iteratively removing unimportant variables repeatedly removes variables and edges from the CP-net structure. This is likely to disconnect the structure into two or more connected components. In this case, applying the above result to separate the resulting query into sub-queries will again reduce the query complexity exponentially. Therefore, despite being a fairly intuitive separation mechanism, this can be a very powerful addition to our preprocessing method.

Example 3.12. Let N be the CP-net given in Example 3.2. In previous examples, we iteratively removed unimportant variables from N (with respect to the dominance query $N \vDash o \succ o'$). This resulted in a CP-net with the structure given on the right hand side of Figure 3.3 and a reduced dominance query given by Table 3.3. The resulting structure is disconnected and has two connected components. Let M_1 denote the induced sub-CP-net over the connected component with variables $\{X_2, X_3, X_4, X_6\}$ and let M_2 denote the sub-CP-net over the $\{X_5, X_9\}$ connected component. By Corollary 3.11, our reduced query over M is true if and only if it locally true for both M_1 and M_2. That is, if any only if $M_1 \vDash (1,0,0,0) \succ (0,1,0,1)$ and $M_2 \vDash \ (2,1) \succ (0,2)$. By Corollary 3.6 and our argument regarding iterative variable removal, our original query, $N \vDash o \succ o'$ is equivalent to the reduced query over M. Thus, $N \vDash o \succ o'$ is true if and only if the M_1 and M_2 queries are both true. This means that we have reduced the original query over CP-net N, with 5,184 outcomes, to two queries over CP-nets with 24 (M_1) and 9 (M_2) outcomes, respectively. Thus, the original problem has been reduced to searching over a total space of 33 outcomes, less than 1% of the size of the original space we needed to search for an IFS.

In general, suppose that we obtain CP-net M with n variables by iteratively removing unimportant variables from N. We then split the reduced dominance query into $k > 1$ sub-queries (if $k = 1$, separation does not reduce the problem) over k sub-CP-nets, $M_1, ..., M_k$. Let n_i be the number of variables in M_i, then the n_i values must sum to n. In the binary case, the space of outcomes we must search through to answer the M_i sub-query is 2^{n_i}. Thus, we can consider the size of our new problem (answering all k sub-queries) to be

$$\sum_{i=1}^{k} 2^{n_i}.$$

This is the total size of the space we must search through to answer these sub-queries by searching for their relevant IFSs. This size is maximal when one connected component has $n-k+1$ variables and the other $k-1$ components each have 1 variable. In this case, the new size of the problem is $2^{n-k+1}+2(k-1)$. The best scenario (when the new problem size is minimised) is when all connected components have equal size (if possible). In this case, the new problem has size $k2^{n/k}$. The reduced query over M originally had size 2^n. Thus, by separating the query, we have reduced the size of the problem by an exponential factor (which increases with k), regardless of how well the components partition the variables. The results are similar in the non-binary case, though the reduction factors are larger in general as the domain sizes (multiplicative factors) can be greater than two. Thus, the proportional reduction in the size of the dominance query problem will be greater, however, the original size of the problem will also be greater than the binary case.

The individual sub-queries produced by separation are much simpler to answer than the reduced query over M. Further, we have shown above that the overall task of answering all of these sub-queries is still significantly simpler than answering the query over M. An even more efficient way to answer our query would be to answer all k sub-queries simultaneously, rather than successively. This is possible because they do not rely on one another. However, we answer sub-queries successively in our experiments, as this time elapsed more accurately represents the size of the reduced task. Further, by answering sub-queries successively, it is possible to determine the query to be false without answering every sub-query – once one sub-query is found false, we know the original query to be false also by Corollary 3.11.

Our complete preprocessing method is called *UVRS (unimportant variable removal and separation)* preprocessing. This method iteratively removes unimportant variables and then separates the resulting query into independent sub-queries. The result, as we have demonstrated, is a query or set of queries that are equivalent to the original but with exponentially reduced complexity. Algorithm 2 gives the pseudocode for this process. Note that Algorithm 6 (see Appendix B.3) is implicitly called here to calculate any ancestor sets. To get descendant sets, we simply reverse all edge directions in the structure (in practice, we do this by transposing the corresponding adjacency matrix) before applying Algorithm 6.

Algorithm 2: UVRS CP-Net Preprocessing

```
Input   : $N \vDash o \succ o'$ – Dominance query ($o \neq o'$)
Output: $M_1 \vDash o[V_1] \succ o'[V_1], ..., M_k \vDash o[V_k] \succ o'[V_k]$ – Set of (equivalent)
        reduced dominance queries
$D = \{X \in V | o[X] \neq o'[X]\}$;
$I = D$;                                  // $I$ - set of important variables
for $X \in V \backslash D$ do
    Anc($X$) – Set of ancestors of $X$;
    if $Anc(X) \cap D \neq \varnothing$ then
        Dec($X$) – Set of descendants of $X$;
        if $Dec(X) \cap D \neq \varnothing$ then
            Add $X$ to set $I$;
        end
    end
end
$U = V \backslash I$;                     // $U$ - set of unimportant variables
$M = N$;
$V' = V$;
// Remove unimportant variables iteratively until a CP-net
   with no unimportant variables is reached:
while $U \neq \varnothing$ do
    Remove variables $U$ from the structure of $M$;
    Adjust CPTs of every variable that has lost a parent;
    For each variable that lost a parent, remove any parent edges that are
      now degenerate (and adjust their CPTs appropriately);
    Remove $U$ from $V'$;                 // $V'$ - variables left in $M$
    $I = D$;
    for $X \in V' \backslash D$ do
        Anc($X$) – Set of ancestors of $X$ (in $M$);
        if $Anc(X) \cap D \neq \varnothing$ then
            Dec($X$) – Set of descendants of $X$(in $M$);
            if $Dec(X) \cap D \neq \varnothing$ then
                Add $X$ to set $I$;
            end
        end
    end
    $U = V' \backslash I$;      // Variables unimportant to the reduced query
end
// Continued on the next page
// Continuation of Algorithm 2
// Identify connected components in the reduced structure:
$V'$ – The set of variables in the reduced CP-net, $M$;
$C$ – Empty list of connected components;
$T$ – Set of variables in $V'$ with no parents in $M$;
$T_D$ – List of descendant sets for variables in $T$;
if $|T| = 1$ then
    // The structure must be connected
    $C = \{V'\}$;
end
else
    while $T \neq \varnothing$ do
        Select at random $X \in T$;
        Remove $X$ from $T$;
        // Identify the connected component of $X$, use $T_D$ to
           look up descendant sets:
        $G = \{X\} \cup \text{Dec}(X)$;
        repeat
            for $Y \in T$ do
                if $Dec(Y) \cap G \neq \varnothing$ then
                    $G = G \cup \{Y\} \cup \text{Dec}(Y)$;
                    Remove $Y$ from $T$;
                end
            end
        until $G$ does not increase;
        Add $G$ to list $C$;
    end
end
// Separate the reduced query into sub-queries over the
   identified connected components:
$k = \text{length}(C) \geq 1$ ;                // Number of connected components
$Q$ – Empty list of sub-queries;
for $i \in \{1, ..., k\}$ do
    $V_i$ – Variables in $i^{th}$ component in list $C$;
    $G_i$ – Induced structure of $M$ over $V_i$;    // $i^{th}$ connected component
    $M_i$ – Sub-CP-net of $M$ over $G_i$;
    Add query $M_i \vDash o[V_i] \succ o'[V_i]$ to list $Q$;
end
return $Q$;
```

As Algorithm 6 has time complexity $O(n^3)$, our preprocessing method (Algorithm 2) has total complexity $O(n^5 + mn^2p)$, where m is the maximum parent set size and p is the maximum number of parental assignments for any variable in the original CP-net, N. The existing method for CP-net preprocessing, forward pruning, has complexity $O(nrd^2)$, where r is the maximum number of conditional preference rules for a variable and d is the maximum domain size (Boutilier et al., 2004a). Note that r and p are essentially equivalent. For reasonably small values of d and particularly in the binary case, these worst-case complexities suggest that UVRS would be slower than forward pruning in general. However, our experiments find that UVRS is in fact significantly faster than forward pruning in practice, even though we are looking at the binary case. The p (and r) term grows exponentially with m (which can be as large as $n-1$), meaning both methods have intractable complexity. However, in our experiments, we find that the improvements in query complexity outweigh their computational cost, even as n increases. This is possible because dominance testing is also an intractable task and so reduction in query complexity can outweigh exponential preprocessing times. Furthermore, as we find UVRS to be more efficient than forward pruning in practice, this suggests that in general the preprocessing times are faster than the worst case complexity.

3.2.3 Combining UVRS with Forward Pruning

In this section, we explain how UVRS can be combined with forward pruning (Boutilier et al., 2004a), the existing method of CP-net preprocessing for efficient dominance testing (full details of forward pruning can be found in §2.2.3). We show that this combination is more effective at reducing dominance query complexity than the sum of the two methods used individually. In particular, combining UVRS with forward pruning enables UVRS to reduce the CP-net further than when used in isolation.

Rather than removing irrelevant variables from the CP-net, forward pruning removes impossible variable values from the variable domains. However, if a variable's domain is reduced down to one value, this is equivalent to fixing (and removing) the variable, as we do for unimportant variables in UVRS. We shall remove any variables that have a reduced domain size of one from forward pruning, in the same way as we remove unimportant variables. Further, we will remove any degenerate parent-child relations resulting from forward pruning. Such removals are not required by Boutilier et al. (2004a). While these modifications do not semantically change the CP-net produced by forward pruning, they allow forward pruning to affect the CP-net structure, not just the domains. This is what enables forward pruning to improve the efficacy of UVRS when used in combinations. Thus, these technical modifications are necessary here, in order to obtain a combination that is more effective than the simple sum of the two methods used individually.

Forward pruning has the advantage that it can prove the dominance query to be false in some cases, meaning no dominance testing is required. If forward pruning reduces a variable's domain to the empty set, then there are no 'possible' values for this variable (that is, there are no plausible values this variable can take in the required IFS). Thus, in this case, the dominance query is automatically false and no further action (preprocessing or dominance testing) is required. This can only be achieved by UVRS in the case where $o = o'$ as all variables are unimportant and, thus, removed. However, $o = o'$ is a trivially false case that it is routine to check for prior to commencing any dominance testing or preprocessing. The above failure condition gives forward pruning an advantage over UVRS as it can reduce non-trivial dominance queries to a problem of size zero (as they are answered).

UVRS and forward pruning reduce CP-nets in distinct but overlapping ways. Any variable that is identified by UVRS as unimportant or subsequently conditionally unimportant because it does not have ancestors in D will also have its domain reduced to a single value (thus removing it) by forward pruning. However, if an unimportant or conditionally unimportant variable has ancestors in D but no descendants, then it is not guaranteed to be removed, or even reduced, by forward pruning. Further, separation of the query reduces the dominance query search in ways that cannot be affected by forward pruning. This is because applying forward pruning and unimportant variable removal results in two reduced CP-nets with overlapping variables (if forward pruning does not automatically find the query false), though they may have smaller domains in the forward pruning case. Any dominance testing performed after forward pruning must consider all possible outcomes associated with the reduced CP-net. That is, all possible variable assignment combinations are considered. However, when we apply separation to the CP-net with unimportant variables removed, we are essentially reducing the combinations we need to consider. Thus, due to the overlap with the forward pruning reduced CP-net, separation can remove combinations that are unaffected by forward pruning. Conversely, forward pruning can prune the domains of (and possibly remove) important variables, which cannot be reduced by UVRS. Thus, the two methods both remove aspects of the CP-net that are unaffected by the other. As they are distinct in this manner, their combination must reduce the CP-net further than either method can alone. We will actually show that their combination is stronger than the sum of both methods used individually.

Combining UVRS with forward pruning is simple. Suppose $N \models o \succ o'$ is the dominance query of interest. First, we apply UVRS to reduce it to a simpler query (or set of queries). We then apply forward pruning to each of the reduced queries. If forward pruning does not alter the structure of the CP-net associated with a given query, then we stop reducing that query at this point. Note that, even if a domain is reduced, if no edge or variable is removed, then the structure is unchanged and we stop preprocessing this query. For those queries where forward pruning does alter the CP-net structure, we then re-apply UVRS one last time. As UVRS and forward pruning both produce reduced queries (or sets of queries) that are equivalent to the original, each new query (or set of queries) generated by this method must be equivalent to the previous. Thus, in order for $N \models \ o \succ o'$ to hold, all produced queries must also be true. Therefore, if forward pruning finds any query to be false in this process, then we can terminate the preprocessing as $N \models o \succ o'$ must also be false. The pseudocode for this combined preprocessing procedure is given by Algorithm 3.

We assume that the query is not the trivial case, $N \models o \succ o$, as this should be checked before commencing. As $o \neq o'$, UVRS cannot find any of the produced dominance queries false (by removing all variables) in this procedure, by the following reasoning. If N has a variable X such that $o[X] \neq o'[X]$, then X is important to the original query and any reductions. Thus, X cannot be removed by UVRS and, so, UVRS cannot remove all variables. Furthermore, as we showed in the proof of Corollary 3.11, if $o \neq o'$, every connected component produced by UVRS must differ on some variable between o and o'. Thus, every sub-query produced by the first UVRS application must contain a variable, X, such that $o[X] \neq o'[X]$ (that is, each sub-query is non-trivial). Similarly, such variables cannot be removed by forward pruning (without resulting in a failed query). Thus, if UVRS is applied a second time, then it cannot remove all variables (and find the query false) by the same reasoning as above as it is, again, applied to a non-trivial query. Hence, UVRS cannot find a query to be false in this process and always returns a reduced query or set of queries.

As we apply only forward pruning and UVRS, each new query (or set of queries) produced is equivalent to the previous. Thus, the resulting reduced query or set of queries must be equivalent to the original, unless the query is found false by preprocessing. If Q_{final} denotes the set of resulting queries, then $N \models o \succ o'$ holds if and only if every query in Q_{final} is true.

We stop reducing a query if forward pruning results in no structural changes because such queries cannot be further reduced by either UVRS or forward pruning. Suppose forward pruning is applied to $M \models a \succ b$ and makes no alterations to

Algorithm 3: UVRS and Forward Pruning Combined Preprocessing

```
Input  : $N \vDash o \succ o'$ – Dominance query ($o \neq o'$)
Output: $M_1 \vDash a_1 \succ b_1, ..., M_\ell \vDash a_\ell \succ b_\ell$ – Set of (equivalent) reduced
        dominance queries
        OR
        The dominance query is false
// Apply UVRS (Algorithm 2) to the query
$Q$ – Set of reduced queries produced by applying UVRS to $N \vDash o \succ o'$;
// $Q_{continue}$ - Set of reduced queries that require a second
   application of UVRS
$Q_{continue}$ – Empty list of dominance queries;
// $Q_{final}$ - Set of reduced queries produced by applying our
   combined preprocessing
$Q_{final}$ – Empty list of dominance queries;
// Apply forward pruning to each resulting query:
for $M \vDash a \succ b \in Q$ do
    Apply forward pruning to $M \vDash a \succ b$;
    if Forward pruning finds $M \vDash a \succ b$ false then
        // Procedure terminates
        return $N \vDash o \succ o'$ is false;
    end
    else
        $M' \vDash a' \succ b'$ – Query produced by forward pruning;
        if $M$ and $M'$ have identical structures then
            Add $M' \vDash a' \succ b'$ to $Q_{final}$;
        end
        else
            Add $M' \vDash a' \succ b'$ to $Q_{continue}$;
        end
    end
end
// For those queries where forward pruning does change the
   structure:
for $M' \vDash a' \succ b' \in Q_{continue}$ do
    Apply UVRS to $M' \vDash a' \succ b'$;
    $M_1 \vDash a_1 \succ b_1, ..., M_k \vDash a_k \succ b_k$ – Resulting set of reduced queries;
    Add each $M_i \vDash a_i \succ b_i$ to $Q_{final}$;
end
return $Q_{final}$;
```

the structure of M (and does not find the query to be false). Let M' be the CP-net produced by forward pruning (note that, as the set of variables has not changed, a and b are not altered). Re-applying forward pruning to M' will not have any effect as we have already removed all impossible values for the query $a \succ b$. By our procedure, $M \vDash a \succ b$ is a query produced by UVRS. Thus, M has a connected structure with no unimportant variables or degenerate parents. Reducing M to M' did not change the structure, so the structure remains connected and all variables remain important. Forward pruning removes degenerate relations so, as no edges are removed, all relations must remain valid (non-degenerate). Thus, applying UVRS to $M' \vDash a \succ b$ will not have any effect. Therefore, if forward pruning does not affect the structure (when applied after UVRS), we cannot further reduce this query via either method.

Alternatively, if a query is reduced by UVRS, forward pruning, and then UVRS again (as forward pruning changes the structure), we stop preprocessing as, again, neither method can make any further progress. As UVRS was applied last, the resulting structure(s) are connected and cannot contain any unimportant variables or degenerate relations. Thus, re-applying UVRS to the resulting structure(s) will have no effect. Suppose $M_i \vDash a_i \succ b_i$ is one of the queries produced by the final application of UVRS. Then this query is the (partial) result of applying UVRS to some query, $M \vDash a \succ b$. As forward pruning was applied previously, M contains no impossible values. Suppose that UVRS identifies and removes an unimportant variable, X, from M. If X has no ancestors in D, then X would have been fixed and removed by forward pruning. Thus, X must have ancestors in D but no descendants, by definition of unimportance. Thus, all descendants of X are also unimportant and would be removed by UVRS. Hence, for every unimportant variable removed, their children are also removed. As no remaining variable has lost a parent, no CPT adjustment is required and, thus, all relations remain valid. The unimportant variable removal process then terminates after the first iteration and we go on to apply separation. Therefore, any variable in M_i has the same parent set as in M. Applying forward pruning to M_i would, therefore, progress exactly as before for the remaining variables. As all impossible values were removed previously, this means that no further values are removed and so forward pruning has no effect. Thus, once UVRS, forward pruning, and UVRS again have all been applied, a query cannot be further reduced by either method.

While a second application of forward pruning would have no effect, our second application of UVRS can result in further reduction. This is because, applying our modified forward pruning (which removes any fixed variables and degenerate parents) can enable UVRS to be more effective. In general, UVRS does not benefit from being applied multiple times. The resulting CP-nets contain only important variables and valid edges and they have connected structures. Thus, applying UVRS again would do nothing. However, forward pruning may remove variables or edges, changing the structure produced by the initial UVRS reduction. Altering the structure in this manner can result in new unimportant variables[1] or a disconnected structure, enabling further UVRS reduction. This is why our forward pruning modifications are necessary – the reductions made by forward pruning must be reflected in the structure to enable identification of newly unimportant variables or disconnected structures.

Hence, this combination incorporates both UVRS and forward pruning reduction and then additional UVRS reductions enabled by the effects of forward pruning. This means that our combination reduces the problem further than simply applying both methods individually (in fact, as further UVRS reduction is applied, it will be reduced by a further exponential factor). Recall that using both methods is, in turn, strictly better than using either method alone as they both prune distinct aspects of the CP-net, unaffected by the other.

This combination reduction can also be performed by applying forward pruning first and then applying UVRS to the reduced query. By applying forward pruning first, the single UVRS application has its full effect (that is, including UVRS reductions enabled by applying forward pruning). Neither method can further reduce the query from this point, by the same explanations as above. We choose to apply the combination in the manner detailed by Algorithm 3 instead for efficiency. As we mentioned in §3.2.2, we find that UVRS is more efficient to apply than forward pruning in practice. By implementing UVRS first, we only have to apply forward pruning (and the subsequent UVRS application) to CP-nets of reduced size. As preprocessing time increases with CP-net size, this is generally more efficient than applying forward pruning to the original query and then UVRS to the reduced. We can see this from our experimental results, as our combination of methods is more efficient to apply than forward pruning alone. If we instead apply forward pruning and then UVRS, this must take longer than forward pruning alone and, thus, be slower than our implementation of the combination. These results can be seen from the experimental results in Figure 3.6, given in §3.3.2.

Remark. The code we use to apply the UVRS and forward pruning combination in practice differs slightly from the pseudocode given in Algorithm 3. In particular,

[1] As we showed in our previous explanations of how this second UVRS application acts, these new unimportant variables must have ancestors but no descendants in D. Further, they are all identified and removed by the second UVRS application in the first iteration of removing unimportant variables.

our code will apply UVRS, then repeatedly attempt to apply forward pruning and UVRS alternately to the reduced queries until one does not alter the sub-CP-net structure (unless the query is found false at some point). This causes a distinction only in the cases where UVRS is implemented a second time and makes a structural alteration. Our code will attempt (unsuccessfully) to apply forward pruning a second time to the resulting query before terminating. On the other hand, Algorithm 3 would not attempt this second forward pruning application. This means that the combination preprocessing times (and net improvement to dominance testing efficiency) that we see in the §3.3 experimental results could be slightly improved upon by using the Algorithm 3 procedure exactly.

Previously, we introduced a novel method of preprocessing CP-nets for more efficient dominance testing. We have now demonstrated that this method is distinct from the existing technique, forward pruning, and shown how they can be combined. We have also shown this combination to be more powerful (by an exponential factor) at reducing the CP-net than simply applying both methods individually. That is, by combining the techniques in this manner, we obtain a preprocessing procedure that is more effective than the sum of its components.

3.3 Experimental Evaluation of Preprocessing Performance

In this section, we evaluate the performance of our preprocessing method (UVRS) experimentally. We compare the performance of UVRS to the existing preprocessing method, forward pruning (Boutilier et al., 2004a), and their combination (as described in §3.2.3). In §3.3.1, we give the details of our experiment and in §3.3.2, we analyse the results of these experiments. These results show UVRS to be significantly more effective than forward pruning at improving dominance testing efficiency in the binary CP-net case. The combination of methods is even more effective, reducing dominance testing times by up to 60% on average, even as n increases.

3.3.1 Experiment Details

In this section, we give the details of the experiments we conducted in order to evaluate and compare the performance of UVRS, forward pruning, and their combination.

First, let us consider the performance measures we use in these experiments. To evaluate the performance of a preprocessing procedure, we must consider how effectively it reduces dominance query complexity. In §3.2, we used the reduction in CP-net outcomes to measure the change in dominance query complexity. This measures the reduction in the associated CP-net size, which is related to query complexity – the set of CP-net outcomes is the space we must consider when evaluating whether or not a dominance query holds (usually by searching for an IFS in this outcome space). Thus, the size of the CP-net reflects the plausible size of the theoretical dominance testing problem. Further, the reduction in CP-net size does not depend either on the method we use to answer the resulting queries, nor the specific code used to implement the preprocessing.

However, CP-net size is not an accurate measure of specific query complexity; different dominance queries for the same CP-net can have different complexities – this depends how close the specific outcomes are within the preference graph. CP-net size reflects the worst case complexity, when the entire outcome space (preference graph) must be explored in order to answer the query. Further, two CP-nets can have the same number of outcomes but distinct preference graph structures. In the non-binary case, they can be distinct even when undirected. As preference graph structure determines query complexity, two CP-nets with the same number of outcomes can have distinct distributions of dominance query complexity. Thus, we cannot predict a 'likely' query complexity given only the number of CP-net outcomes (that is, the worst case query complexity). Thus, considering CP-net size shows us how preprocessing affects the CP-net and the worst case query complexity, rather than the exact complexity of the query (or set of queries) we are interested in.

Alternatively, we can record the time it takes to answer both the original and reduced queries. This provides a measure of the reduction in specific query complexity. Comparing this reduction in time to the time elapsed by preprocessing allows us to determine whether the cost of preprocessing is worth the reduction in query complexity and evaluate the net impact of preprocessing. Time elapsed is also our primary interest as we are aiming to improve dominance testing efficiency. These time elapsed results are, however, dependent upon the specific preprocessing code used and the dominance testing method (and the specific code) used to answer the queries in addition to preprocessing performance, which we are trying to evaluate. We generally consider relative efficiency, rather than exact time elapsed, which mitigates the dependence upon dominance testing code to a degree. Every preprocessing method uses the same dominance testing code, so their relative performance is not dependent upon the specific code implementation.

Finally, we can also consider the outcomes traversed when answering the original and reduced queries, as we did in our Chapter 2 experiments. Outcomes traversed is a theoretical measure of how difficult a query is to answer. Thus, the reduction in outcomes traversed is not dependent upon the specific code used for preprocessing or answering dominance queries. This gives us a theoretical measure of how query complexity has been reduced. However, as different dominance testing methods prune the search tree (and the outcomes traversed) differently, this measure is still dependent upon the method used to answer the resulting queries. In particular, the reduction in outcomes traversed reflects how well preprocessing reduces aspects of the query complexity that are not already removed by the dominance testing method. Thus, the reduction in outcomes traversed is dependent upon the overlap in reduction between the preprocessing method and the dominance testing method. Further, we cannot compare the reduction in query outcomes traversed to the cost of applying the preprocessing. Thus, we cannot use this measure to determine whether the benefit of preprocessing is worth the associated cost, or determine the net benefit of preprocessing when taking computational cost into account (as we can with time elapsed).

Each of the above measures has its own distinct benefits and disadvantages. We will use all three to evaluate the various preprocessing methods. This will give a more detailed picture of how the preprocessing methods perform, as each measure illustrates aspects of performance that the others cannot.

While UVRS, forward pruning, and their combination are all applicable to CP-nets with multivalued variables, our experiments consider only the case of binary CP-nets. In the multivalued case, we may expect different results as UVRS benefits from binary variables (and smaller variable domains in general), whereas forward pruning is likely to be more effective for CP-nets with larger domains, as we discuss in more detail in §3.4. We intend to evaluate performance in the multivalued case in our future work.

In order to evaluate and compare their respective performances in the binary case, we performed the following experiment. Given n (the number of variables), we generated 1000 random binary CP-nets and then generated 10 random dominance queries for each CP-net. To generate the CP-nets, we used the same random generator as we used in the Chapter 2 experiments. Full details are given in Appendix C.1. Generating a dominance query for a binary CP-net over n variables is done by randomly generating two binary vectors of length n.

We answered each generated query, recording the time elapsed and outcomes traversed. Then, for each of the three preprocessing methods, we applied the preprocessing (with slight modifications discussed later in this section) and (if necessary) answered the resulting dominance query (or queries). We recorded both the preprocessing time elapsed and the total time taken to preprocess and then answer any resulting queries. We also recorded the number of outcomes traversed in answering the resulting query and the number of outcomes in the reduced CP-net. If preprocessing results in multiple queries, we record the sum of the number of outcomes in each associated sub-CP-net and the sum of the outcomes traversed over the queries that we answer. If preprocessing answers the query (in particular, finds it to be false), then we recorded zero outcomes traversed for the reduced query and zero outcomes remaining after CP-net reduction. We repeated this experiment for each of $n = 3, 4, ..., 20$.

If preprocessing results in multiple queries, then they must all be true in order for the original dominance query to be true. Thus, if we find any of these queries to be false, then we do not need to continue answering the remaining queries, as we already know the original query to be false. In our experiment, we answer the resulting queries in increasing order of the number of variables in the associated CP-nets. This ordering aims to minimise the dominance testing time for the reduced queries in the cases where the original query is false; ideally, answering queries in increasing order of CP-net size until one is found false will avoid answering more complex queries than necessary.

Let us now consider the method of dominance testing we use in these experiments, in order to answer the original and preprocessed queries. As these experiments require us to answer a large number of queries, for practicality, we need to use an efficient dominance testing method; using a basic search algorithm, or similar, is not feasible. Such methods are also unrealistic in practice and will be maximally improved by preprocessing as there is no overlap between the query reduction performed by the preprocessing and answering methods. Thus, using such basic dominance testing methods will exaggerate the effects of preprocessing in practice – in reality, we are likely to already be using a more efficient dominance testing method which may overlap with preprocessing in its reduction methods.

There are many existing methods of answering dominance queries efficiently, as we reviewed in §2.2.3. Most of these methods improve efficiency by pruning the associated search tree. In Chapter 2, we introduced a new pruning method and experimentally compared its performance to several of the existing methods that preserve search completeness. We also evaluated the performance of all possible combinations of these methods. For our preprocessing experiments, we have chosen to use the combination of rank pruning, suffix fixing, and penalty pruning (with rank prioritisation) for our dominance testing. Out of the methods compared in Chapter 2, this pruning schema is the most effective (it produces the smallest search tree for any given dominance query). This ensures maximal theoretical overlap with the preprocessing, out of those we considered. Thus, the effect of preprocessing will not be unrealistically exaggerated. Further, this method is also one of the most efficient dominance testing methods that we tested. Thus, it is a realistic choice for dominance testing and will allow our experiments to run in a practical time frame. Also, as we are aiming to improve dominance testing efficiency, we naturally want to evaluate whether preprocessing can improve upon the more efficient dominance testing methods that exist.

Remark. Recall that UVRS uses suffix fixing in the removal of unimportant variables. In §3.2.1, we showed that our iterative removal of unimportant variables is distinct from suffix fixing as methods of reducing dominance query complexity. In particular, we showed that suffix fixing prunes parts of the dominance query that are unaffected by UVRS (and vice versa). Thus, even though we employ suffix fixing within UVRS, using suffix fixing pruning when answering the preprocessed query is not obsolete; it can prune the search tree non-trivially, further improving dominance testing efficiency. Thus, including suffix fixing in the pruning schema applied after UVRS (or the combination with forward pruning) can improve the dominance testing efficiency.

We have elected to use the most effective pruning schema (dominance testing method) from our Chapter 2 experiments, rather than the most efficient, for two reasons. Firstly, this method is the union of all of the other pruning techniques considered. Thus, it will reduce the size of the dominance query search maximally for every query (rank prioritisation is also shown experimentally to result in the most effective pruning). This means that this dominance testing method will have the largest overlap (in regards to query reduction) with the preprocessing method. Thus, our results will more realistically depict (without exaggeration) the impact of preprocessing on dominance testing complexity.

Secondly, the CP-nets and dominance queries returned by different preprocessing methods can have different distributions – different from one another as well as different from the distribution we tested in our Chapter 2 experiments. It is possible that certain types of preprocessing may produce queries that are particularly well suited to one pruning method more than another. In general, it is possible that the relative performances observed in Chapter 2 may not be the case for these new distributions. However, regardless of distribution, the combination of all three (suffix fixing, penalty pruning, and rank pruning) will always be the most effective pruning method, as we explained. By using the combination of all three, we cannot inadvertently favour one set of reduced queries over another. As this combination is also one of the most efficient in our Chapter 2 experiments, this solution is also practical for our experiments here and a realistic choice in practice. Further, using this method still illustrates whether preprocessing can improve upon the more efficient of the existing dominance testing methods.

There are other methods of dominance testing that we did not consider in our Chapter 2 experiments (see §2.2.3) which may be more efficient. However, our focus is on how the dominance testing efficiency can be improved by preprocessing. Thus, it is sufficient for these experiments to use a dominance testing method that is efficient enough for practical use and that does not exaggerate the impact of preprocessing.

Our chosen dominance testing method includes a set of three initial conditions that, if true, prove the dominance query to be false (see Appendix C.2 for details). These conditions are simple to check and are evaluated before dominance testing commences. If any of these conditions are found to hold, then no search is required. Thus, as they are simple conditions to evaluate, these checks improve dominance testing efficiency. Prior to dominance testing, preprocessing reduces the original query into several, successively smaller but equivalent queries (or sets of queries) to which we can apply these simple checks and potentially determine the query to be false earlier. Thus, we have integrated these checks into our preprocessing procedures as they allow us to identify cases where no further preprocessing is required (as we can already determine the query to be false).

For UVRS, we check these conditions for the initial query, then again after each removal of unimportant variables (and subsequent degenerate relations). After separation, the initial conditions are checked for each of the new sub-queries before any dominance testing commences. If any initial condition holds at any stage, we can stop preprocessing as the original query must be false. This enables UVRS to determine non-trivial queries to be false and, thus, improves the reduction power of UVRS as well as the overall efficiency of answering dominance queries with UVRS. To make our comparisons fair, we have also added these checks to forward pruning. The initial conditions are evaluated for the original query and the reduced query produced by forward pruning. For the combination of methods, we implement the same checks as above each time UVRS or forward pruning is applied. This will improve the effectiveness and overall efficiency of using forward pruning or the combined preprocessing.

This modification is particularly likely to improve efficiency in the case of UVRS and the combined preprocessing. This is because, over the course of its application, UVRS produces several new (increasingly reduced) queries due to the iterative removal of unimportant variables and the final separation into multiple sub-queries.

This offers multiple opportunities to check the initial conditions and, potentially, answer the query without further preprocessing, nor any dominance testing. It is worth checking the conditions for each new query, both because the remaining preprocessing time is saved and because it is possible for a query to satisfy one of the conditions (proving the query false) prior to reduction but not after. Thus, checking each new query maximises the chance of finding the query false without needing to perform dominance testing.

The impact of adding these initial conditions may contribute to why we find UVRS to be faster than forward pruning in practice, despite its greater theoretical complexity. Initial conditions are simple to check and, therefore, add only a minor cost to preprocessing. Thus, we would recommend always integrating these checks (or similar), especially when using UVRS as they are particularly likely to improve efficiency. Here we have utilised the initial checks associated with our dominance testing method, however, other initial checks are possible – as we discuss in Chapter 2, any method of answering an ordering query can be used (several such methods are given in Chapter 2).

3.3.2 Results

The results of our experiments are summarised in Figures 3.4–3.8. In each plot, the shaded areas represent the $\pm SE$ (standard error) interval for the function of the corresponding colour. This interval depicts where we expect the true mean performance of the function to lie. The uncertainty represented by this interval has different causes in the various graphs, which we discuss below.

Figure 3.4 shows the average proportion of outcomes removed by preprocessing. All dominance queries that can be found immediately false by the initial conditions are not included in these averages. Such queries are answered in the preprocessing stage and, thus, the preprocessed CP-net is recorded as size zero. However, whether or not preprocessing is applied, these queries are answered in the same way – by the first check of initial conditions. We exclude them from the average because it is inaccurate to say that preprocessing has removed 100% of the problem in these cases.

As all original CP-nets have size 2^n, the variation in performance here is entirely due to variation in the effectiveness of the various preprocessing methods at reducing the CP-net size. How many variables are removed by UVRS depends upon which variables take the same value in the outcome pair, the CP-net structure, and on certain CPTs (to determine whether edges become degenerate). How many values are removed by forward pruning depends upon the exact outcome pair

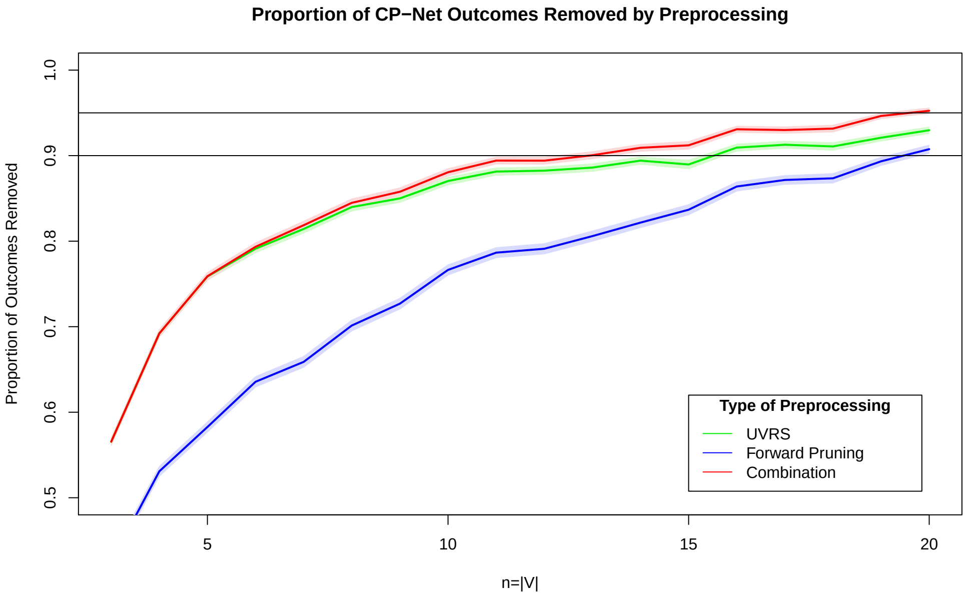


Figure 3.4: Proportion of Outcomes Removed by Preprocessing
Horizontal reference lines at 0.9 and 0.95

and the specific preference graph structure. These factors are dominance query specific and, thus, all three preprocessing methods will show varied performance in our experiments. The effect of checking initial conditions (which is also query specific) will add further variation to preprocessing performance. However, for these results, the intervals remain fairly small, meaning that due to the size of our experiment, we can estimate the true mean performance (proportional outcome removal) reasonably precisely.

This graph shows that UVRS significantly reduces the CP-net size, initially removing 60% of the original problem and quickly increasing (with n) to around 90% and then to almost 93% on average. This shows UVRS to be very effective at reducing the size of our (worst case) dominance testing problem. As CP-nets get exponentially larger with n and, thus, dominance queries get harder, we are most interested in effective preprocessing for larger n values. Thus, it is advantageous that UVRS becomes more effective as n increases, removing more than 90% of the problem for all $n > 15$. These results also show that UVRS performs significantly better than forward pruning at reducing CP-net size, though the difference decreases for larger n.

The combination of the two methods performs even better than UVRS alone and the degree of improvement appears to be increasing with n. For larger values of n ($n > 15$), the combination removes approximately 2% more of the original outcomes. If one considers that UVRS alone removes over 90% of outcomes, then applying the combination instead reduces the problem size further by more than 20%. Thus, by this measure, it is worth applying the combination of methods over UVRS alone. For scale, consider the largest case, $n = 20$, which generally gives the most difficult dominance queries to answer. The original CP-nets have $2^{20} = 1,048,576$ outcomes. On average, UVRS alone reduces this to roughly 74,000 outcomes, but the combination reduces it further to approximately 50,000. This is a significant further reduction in size.

The lines of this graph appear to be beginning to plateau as n increases, particularly UVRS and the combination. It appears likely that the combination will plateau at an average of over 95%, a substantial reduction in size. Note that, as CP-net size is exponential in n, even if this proportion stops increasing, the number of outcomes removed by the combination (and the other methods) is still increasing exponentially with n. To illustrate this, consider the $n = 15 - 20$ results. The percentage removed by the combination increases only slightly here, from approximately 91% to 95%. However, the size of the reduction in outcomes is exponentially increasing. For $n = 15$ it reduces CP-nets of size 32,768 to approximately 2881 outcomes on average, whereas in the $n = 20$ case, it removes approximately 1,000,000 outcomes on average, as we saw above. The size of this reduction will continue to grow exponentially with n (regardless of whether the proportion removed plateaus or continues to grow).

Figure 3.5 shows, for each preprocessing method, the total time taken to apply preprocessing and answer the reduced queries as a proportion of the total time taken to answer the unprocessed queries. Note that, if preprocessing answers the query, then the subsequent dominance testing time is recorded as zero.

For this graph, the standard error interval represents uncertainty due to the variation in both the time it takes to perform preprocessing and the time it takes to answer any resulting queries. UVRS preprocessing time depends on how many iterations of unimportant variable removal are performed, how many variables lose a parent (meaning CPT adjustment and a degeneracy check is required), and whether separation is required. Forward pruning time depends on whether the query is found false at some stage, the number of un-pruned parent values for each variable and, again, the number of variables that lose a parent. For the combination, preprocessing time also depends upon whether we must re-apply UVRS for a second time and the size of CP-nets passed to forward pruning and the second UVRS application. All three preprocessing times also depend upon whether initial conditions are met at any stage. All of these factors will depend upon the specific CP-net and dominance query and, thus, preprocessing times vary

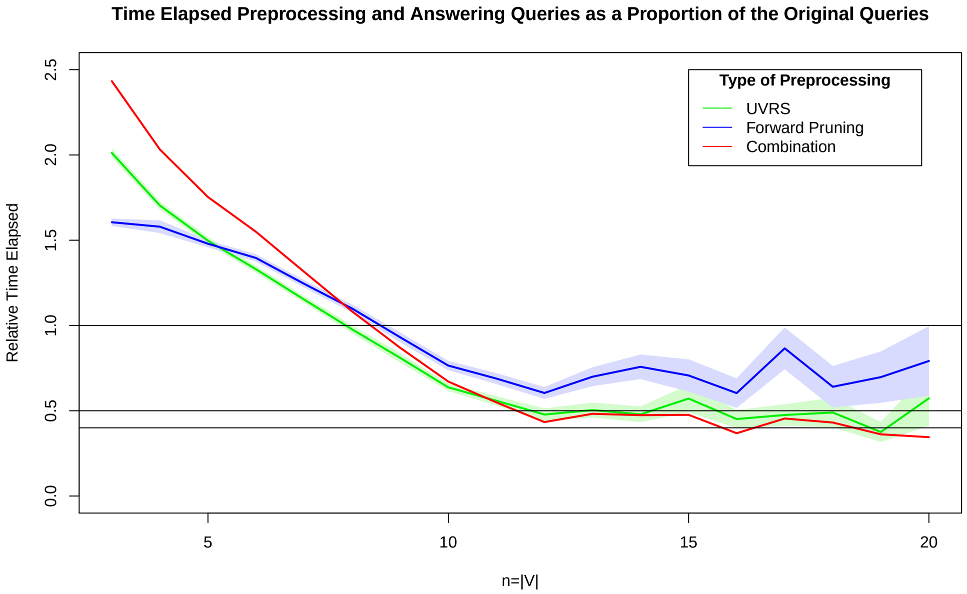


Figure 3.5: Time Elapsed by Preprocessing and Dominance Testing Proportional to Original Query Dominance Testing
Horizontal reference lines at 0.4, 0.5, and 1

in our experiments. We will see in Figure 3.6 that the variation in preprocessing times increases with n. This is because the factors above can vary more as the number of variables increases.

The second factor contributing to the variation in Figure 3.5 is the time it takes to answer the resulting dominance queries. As we discussed in §2.4.2, the complexity of a dominance query depends on both the CP-net and the specific outcomes. Thus, there will be variation in the time elapsed when answering different queries (regardless of what method is used). Further, as we discussed above, the performance of preprocessing varies. Thus, even if all original CP-nets have n variables, the reduced queries will be over CP-nets of varying sizes (and possibly over multiple sub-CP-nets). As CP-nets with more variables generally result in harder queries, this will result in further variation in the dominance testing times.

Figure 3.4 suggests that the size of the reduced CP-net(s) generally increase with n. Larger CP-nets have more variation in their query complexities as the preference graph is larger and so distance between outcomes can vary more (also the convolution of the preference graph can vary more). Thus, as the size of the reduced CP-nets grows with n, the variation in the reduced dominance testing times will also increase with n (as we can see in Figure 3.7). The variation in the number and sizes of the reduced CP-nets will also increase with n, which will also contribute to the increasing variation in dominance testing time.

Hence, both aspects of variation in Figure 3.5 increase with n, which explains why our error intervals becomes wider with n. In fact, as it is the error of the proportion that is growing, the variation must be growing faster than the denominator – the average dominance testing time of the unprocessed queries. Thus, for the larger values of n and any $n > 20$, a larger experiment is required in order to obtain accurate estimates of the average proportional time elapsed. However, for most of our data points, we have reasonably accurate estimates of the true mean performance.

The results in Figure 3.5 suggest that, for $n < 9$, it is not worth applying any of the preprocessing methods; in these cases, the average time taken is longer than the average time to answer the original, unprocessed query. Thus, the time cost of preprocessing is not worth the reduction in the dominance testing time in these cases, which we look at in detail in later plots. However, for the larger values of n (which are of more interest for preprocessing), utilising UVRS reduces the average time by approximately half and the combination is even more effective, tending towards 40% of the original time as n increases (even though we are already using an efficient dominance testing procedure). Thus, both of these methods result in a substantial improvement in efficiency. As the combination is more efficient than UVRS (for larger n), these results suggest that the additional outcomes removed by the combination (as we saw in Figure 3.4) are worth the extra complexity of applying the combination over UVRS alone. These results also show that (for larger n) using UVRS is significantly more efficient than using forward pruning; forward pruning does not reduce average time to less than 60% of the original time for any n. While the combination of methods is not significantly better than UVRS here, it does consistently perform best for $n > 11$. Thus, when considering the tradeoff of preprocessing complexity and performance, we find the combination of methods to be the optimal choice, though it is only worth applying for the $n \geq 10$ case.

These relative efficiency results echo the reduction performance we saw in Figure 3.4, at least for the larger n case (once preprocessing becomes viable). One might suspect that the relative powers of dominance query reduction seen in Figure 3.5 follow from these results regarding their ability to reduce CP-nets (though adjusted for preprocessing costs). However, as we shall see from the following plots, this is not the case.

To give a sense of the scale of these problems, the average time taken to answer these queries (with or without preprocessing) for $n < 15$ is less than 0.01 seconds. These average times grow rapidly from $n = 15$ to 20. For unprocessed queries, average time goes up to 3.02s. The average time when using forward pruning grows to 2.39s. UVRS times go up to 1.73s and, for queries preprocessed by the combination, average time increases to 1.04s. Thus, in our experimental cases, the preprocessing methods are only seconds faster on average than answering the unprocessed queries. However, such differences will become significant for any application that must perform large numbers of queries. Furthermore, dominance testing times rapidly increase with n. Thus, if the proportional preprocessing times plateau, as they appear to be doing, (or if they continue to decrease) the time saved by preprocessing will quickly become minutes and then hours for $n > 20$ (similarly to how outcome reduction increases exponentially with n, even if the proportions removed plateau). If these proportions do plateau then, using the combinations of methods, we will be able to reduce the dominance testing time by around 60%, on average, regardless of how large n becomes.

The results in Figure 3.5 do not follow a smooth curve like those in Figure 3.4. In particular, they become more noisy as n increases. We believe that this is due to the increasing variation resulting in estimates of decreasing accuracy from our sample size. As we discussed above, the numerator varies increasingly with n. Unlike the outcomes removed results in Figure 3.4, the denominator (time taken to answer the original queries) also varies in these results. This variation increases with n as the complexity of dominance queries varies increasingly with n, as we discussed above. Thus, we believe the fluctuations occurring for larger values of n are caused by the increase in variance of both the numerator and denominator. To obtain more precise proportion estimates in these large n cases, an experiment with greater sample size is required.

Figure 3.6 shows the average time it takes to apply preprocessing over the 10,000 queries, for each of the three different preprocessing methods. In this graph, the uncertainty represented by the error intervals is due to variation in how long it takes to perform preprocessing. We discussed above the causes of this variation and why it increases with n, meaning the intervals get wider as n increases. As you can see from the error intervals in Figure 3.6, we have reasonably accurate estimates of the mean times for the majority of data points, despite the fact that variation is increasing with n.

Note that all preprocessing times are growing increasingly rapidly with n. This is to be expected as, for all methods, preprocessing complexity generally grows with n. However, dominance testing times also grow increasingly rapidly with n and Figure 3.5 shows us that preprocessing continues to improve efficiency overall as n increases. Thus, despite the growth in preprocessing times, they remain outweighed by the dominance testing time preprocessing saves. Thus, it remains

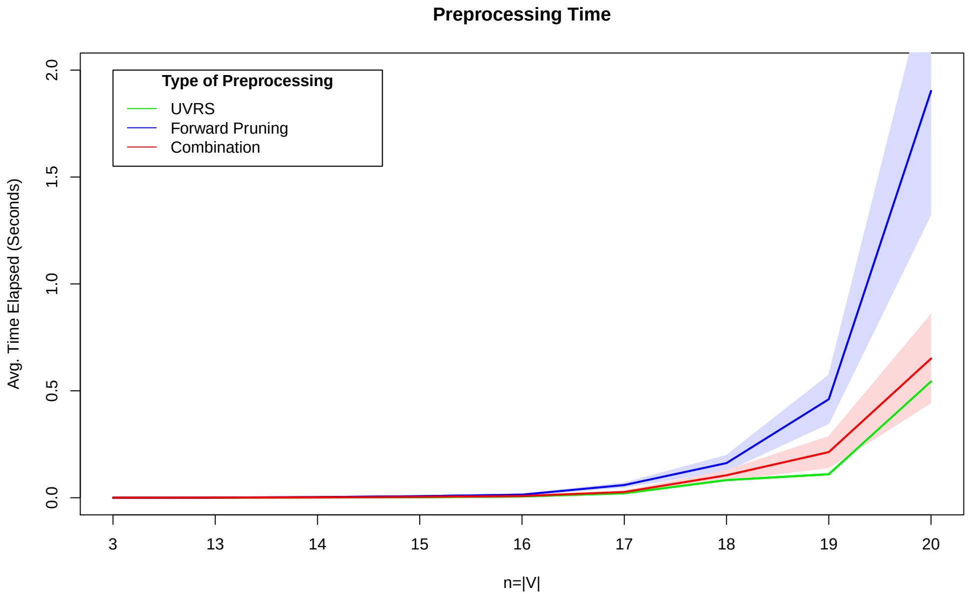


Figure 3.6: Preprocessing Times
Note: n values between 3 and 13 are compressed in order to improve plot clarity for larger n values

faster to use preprocessing than answer the original, unprocessed query, even as n increases.

Figure 3.6 shows that forward pruning takes significantly longer than the other methods for larger values of n. This will contribute to why forward pruning takes significantly longer than the other methods in Figure 3.5.

The combination of preprocessing methods also takes longer than UVRS. This is as expected, as the combination starts by applying UVRS and then goes on to apply forward pruning and then UVRS again (if the query is not answered). However, applying the combination of methods is more efficient than forward pruning, for larger n. This is because UVRS is applied first in the combination and, thus, forward pruning is applied to an already reduced CP-net (as is the second UVRS application, if used). This is more efficient than applying forward pruning to the original query as UVRS is more efficient and preprocessing time increases with n. The fact that the combination is faster to apply than forward pruning alone shows that our implementation of the combination is more efficient than if we applied forward pruning first and then UVRS (which would be theoretically equivalent), as we discussed in §3.2.3.

Despite the fact that the combination takes longer than UVRS to implement, Figure 3.5 shows that, overall, it is more efficient to use the combination than

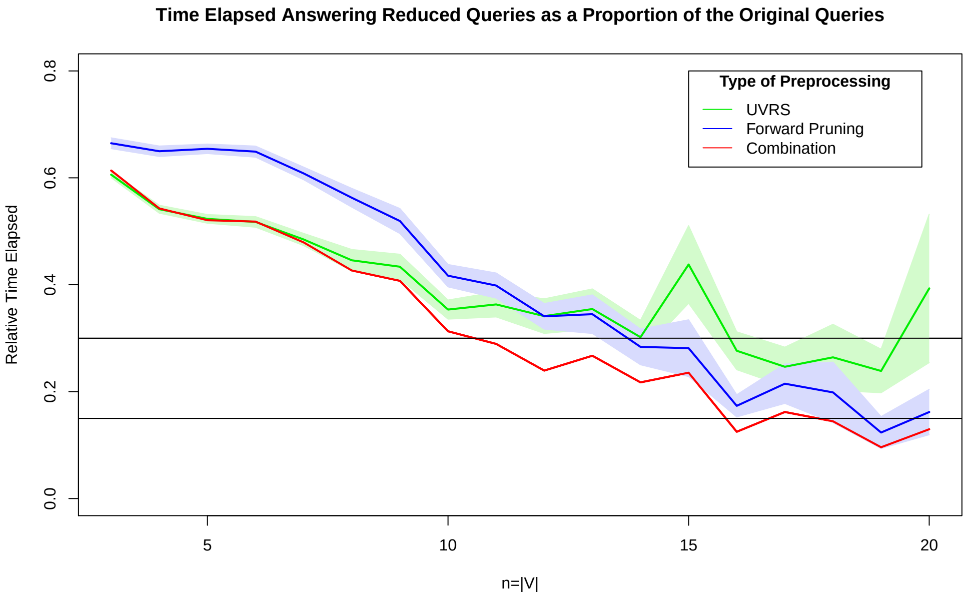


Figure 3.7: Time Elapsed Answering Preprocessed Queries
Proportional to Original Queries
Horizontal reference lines at 0.15, and 0.3

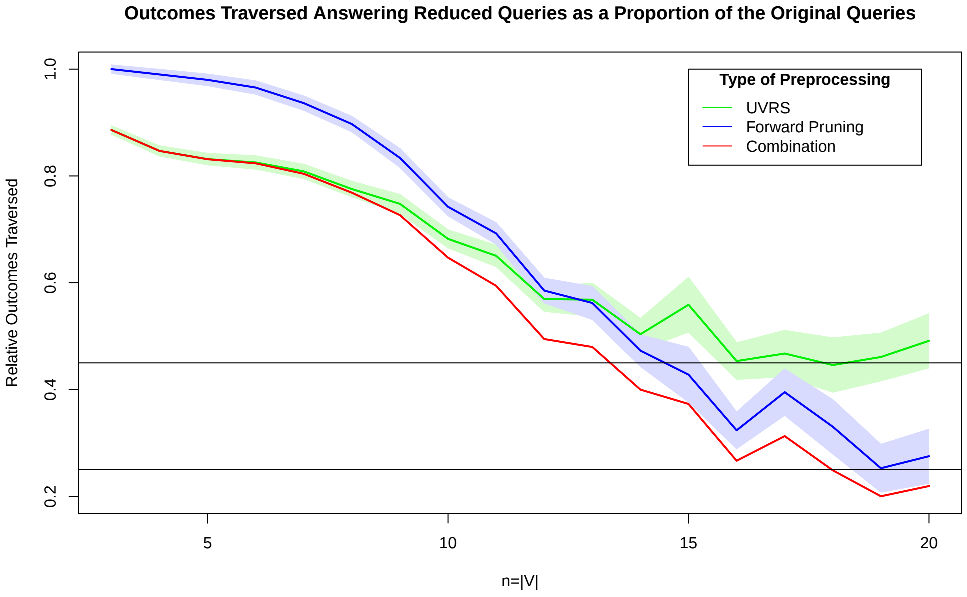


Figure 3.8: Outcomes Traversed Answering Preprocessed Queries
Proportional to Original Queries
Horizontal reference lines at 0.25, and 0.45

UVRS alone. This shows that the the further reduction in query complexity provided by the combination outweighs this extra preprocessing time.

Figure 3.7 shows the total time elapsed when answering the reduced queries as a proportion of the total time elapsed when answering the original, unprocessed queries. Figure 3.8 similarly shows the proportional outcomes traversed. Recall that when a query is answered by preprocessing, the time elapsed and outcomes traversed for the subsequent dominance testing are both recorded as zero. In both graphs, all cases where the original queries are answered immediately by initial conditions are excluded. This is because such queries are answered by preprocessing, but it is not accurate to say preprocessing reduced query complexity by 100% – such queries are answered immediately by initial conditions, regardless of whether or not preprocessing is applied. In such cases, preprocessing is essentially not even applied. As these cases do not illustrate the effect of preprocessing reduction, we exclude them from our results.

In Figures 3.7 and 3.8, the uncertainty represented by the error intervals is due to the variation in time elapsed and outcomes traversed, respectively, when answering the reduced queries. We explained previously, for Figure 3.5 (where time elapsed contributes to variation), the causes of variation in dominance query complexity and why this variation increases with n. The latter explains why these error intervals become wider for larger values of n. In fact, these variations must be growing faster than the respective denominators (the unprocessed query times and outcomes traversed), as the error of the proportion is growing with n. However, the variation in proportional time elapsed and in outcomes traversed is less than the variation of the proportional total time, depicted in Figure 3.5. This is as we would expect for time elapsed, as the total time variation also incorporates variation in preprocessing times (as well as the variation in reduced dominance testing times). Thus, we have more accurate estimates of the true average proportions in Figures 3.7 and 3.8. In these figures, the error intervals remain reasonable sizes, even for the larger values of n.

Figures 3.7 and 3.8 again do not follow a smooth curve and become more erratic for larger values of n. This is for the same reasons as Figure 3.5. As in Figure 3.5, the denominators of the proportions here are the complexity of the original query and the numerators are the complexity of the reduced queries. Thus, both numerator and denominator vary increasingly with n. However, unlike in Figure 3.5, only non-trivial queries are considered and we do not include the preprocessing time. This removes some of the variation we had in Figure 3.5 and perhaps explains why these plots behave less erratically for the larger n values. Despite the fact that Figures 3.7 and 3.8 have more precise estimates and fluctuate less, larger experiments are again required for the larger values of n and any $n > 20$, in order to obtain more precise estimates of average performance.

In both of Figures 3.7 and 3.8, we can see that the reduced query (or queries) is always simpler to answer than the original, both in time cost and theoretical complexity (outcomes traversed), as all data points are < 1 ($n = 3$ forward pruning outcomes traversed is the one exception, where complexity is the same as the original). This is as we would expect, as all preprocessing methods must produce an equivalent query over the same or a strictly smaller CP-net(s). In particular, all time elapsed data points show the reduced queries take less than 70% of the time required by the original queries. This shows that the reason that preprocessing does not improve efficiency for small n values in Figure 3.5 is not because dominance testing efficiency is not improved, but because the improvement is not worth the time cost of applying the preprocessing. From the Figure 3.6 results, we can see that, for such small n values, the average preprocessing times are between 0.000026 and 0.00013 seconds. Thus, as such minimal time costs outweigh a reduction of $30-60\%$ in dominance testing time, the original dominance testing must already be incredibly efficient in these cases. Recall that we are already using one of the most efficient pruning methods from our Chapter 2 experiments to answer these queries. Thus, while preprocessing does significantly reduce complexity in the small n cases, our dominance testing method is already too efficient for the reduction to be worth even minimal preprocessing costs. As dominance testing is already very efficient in these cases, further improvement to efficiency is not as important.

As n increases, the proportional reduction of dominance query complexity (both time elapsed and outcomes traversed) increases for all three preprocessing methods. For the larger values of n, UVRS reduces the time elapsed to 30% of the original and reduces outcomes traversed to 45% of the original. UVRS appears to be plateauing at these proportions, suggesting that, regardless of how large n becomes, UVRS will, on average, return significantly simpler dominance queries. The performances of forward pruning and the combination of methods do not show signs of plateauing in these graphs, though their rate of decrease is perhaps slowing as n gets closer to 20. As n increases, these methods get more effective, reducing time elapsed to around 15% of the original and outcomes traversed to around 25% for the largest values of n. Note that the combination of methods is significantly more effective at reducing queries than forward pruning, though the difference is shrinking as n increases.

Figure 3.6 shows that, as n increases and our preprocessing methods get increasingly effective at reducing dominance query complexity, the time required for preprocessing grows at an increasingly rapid pace. However, from Figure 3.5, we can see than the the proportional total dominance testing time when using preprocessing stabilises for larger values of n. In particular, using forward pruning takes approximately 70% of the original time and using UVRS or the combination takes less than 50% of the original time. This shows that the increasing reduction in query complexity for larger n outweighs the growing preprocessing cost. That is, the time saved by preprocessing must be growing faster than the time it takes to preprocess, as the proportional net time saved plateaus. Thus, for larger n, the reduction of query complexity is worth the preprocessing costs and so preprocessing becomes an effective way of improving dominance testing efficiency.

For UVRS, the reduction in dominance testing time and total time both appear to plateau for larger n (Figures 3.5 and 3.7). This shows that, even though UVRS preprocessing time is growing at an increasingly rapid rate (Figure 3.6), this growth must be proportional to the growth of the original dominance testing time. As both the preprocessing time and the reduction in dominance testing time grow proportionally to the original dominance testing time, this suggests that UVRS will continue to reduce overall time by over half as n continues to grow. On the other hand, forward pruning and the combination of methods reduce dominance query complexity by increasingly large proportions as n increases, but the overall reduction in time appears to level off (proportionally). This suggests that the preprocessing times are growing at a faster rate than the original dominance testing times. This will be a concern if the proportional reductions in query complexity level off for larger n as Figures 3.7 and 3.8 seem to suggest they may. If this proportion becomes constant, but the preprocessing times continue to grow faster than the original dominance testing times, then the overall proportion of time saved by these methods will begin to decrease – the preprocessing costs will begin to eclipse the benefits again for sufficiently large n. This is more likely to be a concern for forward pruning, which already has significantly slower preprocessing times that are growing faster than those of the combination of methods.

Now let us consider the scale of the reductions presented by Figures 3.7 and 3.8. Recall that these results exclude any trivial dominance queries. The average time taken to answer non-trivial queries without any preprocessing is less than 0.03 seconds for all $n < 15$ cases. Between $n = 15$ and 20, this average time rapidly increases to an average of 18.3 seconds. For all three preprocessing methods, the preprocessed queries are answered on average in less than 0.008 seconds for the $n < 15$ cases. This is as we would suspect from our results in Figures 3.7 and 3.8 as they reduce query times by up to 70 or 80% for $n < 15$. For $n =$ 15 to 20, UVRS preprocessed query times increase to 7.2 seconds on average, forward pruning reduced queries increase to 3.0 seconds, and queries reduced by the combination up to 2.37 seconds on average. Thus, by applying preprocessing, we save several seconds per query, on average. These savings are particularly large for greater n values and when utilising the combination of methods. These seconds saved will quickly add up if a large number of queries are required. Further, if the proportions either plateau or continue to decrease, these reductions in query complexity will quickly start saving minutes and hours as n continues to increase because dominance testing time grows rapidly with n.

In both of Figures 3.7 and 3.8, forward pruning is the least effective reduction method for small n values. However, as n increases, the proportional reduction in query complexity plateaus for UVRS and forward pruning continues improving. Thus, for larger values of n, forward pruning is more effective. However, the combination of methods is the most effective at reducing query complexity in all cases. This is as we would expect, as the combination of methods is a strictly stronger preprocessing technique than either of UVRS or forward pruning (or both), as we showed in §3.2.3. Thus, the combination must result in simpler dominance queries. The improvement in query reduction between forward pruning and the combination shows that adding UVRS to forward pruning results in a notable improvement, even though forward pruning has overtaken UVRS in effectiveness. This is because the two methods reduce queries in distinct manners and, thus, the combination will always be more effective than either method used individually (it is also better than using both in this case).

The relative performance of the three methods in Figures 3.7 and 3.8 is distinct from what we have seen in Figures 3.4 and 3.5. In these plots, UVRS significantly outperforms forward pruning for larger values of n, in fact, for all cases in Figure 3.4. Combining the Figure 3.7 and 3.8 results with Figure 3.4 suggests that, while UVRS is more effective at reducing the CP-net size, forward pruning results in simpler queries. This is possible because, as we have mentioned previously, dominance query complexity depends upon the specific CP-net and outcome pair and is not fully determined by CP-net size, even though larger CP-nets generally have more complex queries.

Looking at Figure 3.5, we can see that, in the cases where preprocessing is effective (for larger n values), applying UVRS is more efficient than forward pruning. In this graph, we consider how long it takes to apply preprocessing and then answer the reduced query, as a proportion of the time taken to answer the unprocessed query. Thus, for UVRS to be more efficient than forward pruning for large n here, despite the fact forward pruning results in faster reduced queries, it must be down to the difference in preprocessing times. That is, the longer preprocessing times for forward pruning (seen in Figure 3.6) outweigh the slight further reduction in average query complexity.

Alternatively, using the combination of methods results in the simplest and most efficient reduced queries, as we can seen from Figures 3.7 and 3.8. Further, because of how we implemented this combination, its preprocessing time is significantly faster than forward pruning and not that much slower than UVRS alone. In this case, the superior reduction of query complexity over UVRS outweighs the slightly slower preprocessing time. We can see this as the combination is shown to be more efficient than UVRS overall in Figure 3.5. Thus, by combining forward pruning with UVRS in this manner, we have produced a more effective method of reducing query complexity that is also significantly more efficient to apply than forward pruning. As a result, the combination maximally reduces the overall dominance testing time (by approximately 60% for larger n values – see Figure 3.5).

Notice that the proportions in Figure 3.7 are lower than those in Figure 3.8. That is, preprocessing reduces the query times (proportionally) further than the number of outcomes traversed (proportionally). This discrepancy could be due to the fact that outcomes traversed does not consider the complexity of constructing the search tree, only how big the tree becomes. When constructing a search tree, we must evaluate the improving flips of a given leaf, apply pruning conditions to each, and add the un-pruned flips. To find the improving flips, we consider changing the value of each of the n variables. Finding these improving flips requires evaluating parents and consulting CPTs, both of which are likely to be larger for greater values of n. The larger n is, the more improving flips a leaf is likely to have and we must evaluate the pruning conditions for each. Checking these pruning conditions will also take longer for larger values of n. Furthermore, when n is greater, the search tree will be a larger computational object and, thus, more complex to manipulate. Therefore, the time it takes to add α outcomes to the search tree can depend on the size of the CP-net. This means that two dominance queries can have the same number of outcomes traversed but take different amounts of time, particularly if they are over CP-nets of distinct sizes. As outcomes traversed is a theoretical measure of query complexity, it is blind to some important practicalities that contribute to the complexity of answering dominance queries. It is possible that the reason the outcomes traversed proportion is higher (suggesting less reduction in complexity) is because it does not take into account that the outcomes traversed for the reduced queries were more efficient to perform (than the original) as the associated CP-net is smaller. In this sense, time elapsed is perhaps a more reliable measure of query reduction, though it is sensitive to the specific code implementation used whereas outcomes traversed is not.

The above results from Figures 3.4 – 3.8 show a general pattern of improvement for all preprocessing methods as n increases; all three preprocessing methods reduce both CP-net size and query complexities more effectively for larger n and overall improvement in efficiency increases with n. This is beneficial, as dominance queries get harder with n and so we are more interested in improving efficiency in the larger n cases. Furthermore, these results suggest that preprocessing will continue to be this effective (or better) as n continues to increase.

These results show that UVRS is significantly more effective than forward pruning at reducing CP-net size, removing on average over 90% of outcomes for larger n values. Further, UVRS results in more efficient dominance testing overall than forward pruning for large n. In particular, using UVRS reduces the average dominance testing time by approximately 50% from the unprocessed query time (even when using an already efficient method for dominance testing). However, forward pruning is found to be more effective than UVRS at reducing query complexity. This does not result in forward pruning being more efficient than UVRS overall because this is outweighed by the cost of applying forward pruning (which is significantly slower than UVRS). Alternatively, our combination of the two methods is the most effective at reducing both CP-net size and query complexity. This combination removes between 90 and 95% of outcomes from the CP-net and results in queries that can be answered in 15% of the original time for larger n. As it has modest preprocessing times, the combination is also the most efficient method overall, reducing the average dominance testing time by up to 60% for larger n. Hence, the combination of methods is both superior at reducing dominance query (and CP-net) complexity and the most successful at improving overall dominance testing efficiency, which was our original aim in this chapter.

3.4 Discussion

In this chapter, we have introduced a novel method of improving dominance testing efficiency by preprocessing the CP-net and, consequently, simplifying the dominance query. We call this method UVRS preprocessing. UVRS works by iteratively removing variables that are unimportant to the dominance query (using results based upon suffix fixing by Boutilier et al., 2004a, and prefix fixing by Wilson, 2004b) and then partitioning the reduced problem into several independent sub-queries that can be answered separately and are equivalent to the original query. Each of these steps reduces the size of the CP-net (and, thus, the size of our dominance query problem) by an exponential factor. We have also shown how UVRS can be combined with forward pruning, the existing method of CP-net preprocessing. As forward pruning and UVRS are distinct preprocessing methods, their combination must be more effective at reducing CP-nets than either method used individually. We have shown that this combination can also prune aspects of the CP-net that are unaffected by either of UVRS or forward pruning when used in isolation. Thus, this combined preprocessing technique is more effective than the sum of its component methods.

We performed an experimental evaluation of the performance of UVRS, forward pruning, and their combination. Whilst forward pruning was introduced by Boutilier et al. (2004a) as a heuristic for improving dominance testing efficiency, our experiments constitute the first evaluation of its effectiveness. In these experiments, we evaluated the effect of preprocessing when using an already efficient method of answering dominance queries. In particular, we used the most effective pruning method from our Chapter 2 experiments. This ensures that our results show a realistic impact of preprocessing, not exaggerated by using a basic or impractical search method.

Our results found that UVRS is significantly more effective than forward pruning at reducing the average dominance testing time. On average, UVRS reduces the time by approximately 50% for larger values of n. Furthermore, as n increases, this proportion appears to plateau, suggesting that UVRS will halve the average dominance testing time even as n continues to grow. This is beneficial as CP-nets with more variables generally have more complex dominance queries. Thus, we are particularly interested in improving dominance testing efficiency in these cases. Using the combination of UVRS and forward pruning is even more efficient, saving, on average, up to 60% of the original, unprocessed query time for larger values of n. This proportional performance also appears to plateau as n increases. Thus, we have introduced two methods (UVRS and the combination with forward pruning) of significantly improving dominance testing efficiency. In particular, these preprocessing methods can further improve the dominance testing efficiency we achieved via pruning methods in Chapter 2.

All experiments in this chapter were done on binary CP-nets. However, all three preprocessing techniques are also applicable to non-binary CP-nets. In the binary case, UVRS has an advantage in that the number of variables taking the same value in a given pair of outcomes is likely to be reasonably high, as every variable can only take one of two possible values. This is important to UVRS performance because only variables that take the same value in both outcomes can be removed.

In the multivalued case, domain sizes are larger and the probability of a variable taking the same value in an outcome pair is smaller. Thus, we might reasonably expect UVRS to be less effective at reducing CP-nets (and the associated queries) in the multivalued case. Conversely, for forward pruning, as domain sizes increase there are more possible values that may be pruned and more ways in which this can happen (as there are more possible CPTs and, thus, domain transition graphs). Thus, we may expect forward pruning to remove more domain values on average in the multivalued case and therefore be more effective.

Hence, we expect the results in the multivalued case to differ from those we have seen in our binary experiments. In particular, we expect UVRS and forward pruning to either have closer performance results or perhaps for forward pruning to be more efficient for multivalued CP-nets (particularly as domain sizes increase). However, as the two methods reduce CP-nets in distinct ways, their combination will still be the most powerful preprocessing method (in fact, we have shown this combination to be more effective than using both methods individually). As the combination also has reasonable preprocessing times, from what we have seen, we expect that it will also be the most efficient method overall in the multivalued case. Evaluating and comparing preprocessing performance in the multivalued case is one of our main priorities for our future work on CP-net preprocessing. Preliminary experiments in the multivalued case suggest, as we predicted above, that forward pruning is more effective than UVRS in this case. Also as we predicted, using the combination appears to be the most efficient method of dominance testing again. Further, the combination appears to reduce dominance testing time by a larger proportion and for smaller values of n than in the binary case. We intend to explore these results in a larger, more comprehensive experiment in our future work.

Several of our performance measures for preprocessing have variances that increase with n, as we discussed in §3.3.2. In some cases, this has lead to somewhat imprecise estimates of the mean preprocessing performance in our experimental results. Thus, for the larger values of n, we would like to repeat our experiments with a larger sample size. We would also like to extend our experiments to values of $n > 20$ (where a larger sample size is again a necessity for the above reasons). These $n > 20$ results will answer some of our outstanding questions about how preprocessing performance behaves as n continues to increase. For example, whether various proportional performance values are plateauing (and at what value) or are going to plateau for larger n values. Also, whether the overall proportional efficiency of forward pruning or the combination of methods will decrease if their proportional query reductions do plateau for larger n.

In future experiments, we would also like to explore how the complexity of the CP-net structure (measured, for example, by graph density) and the Hamming distance of the relevant dominance query affect preprocessing performance. We would expect, in general, that all preprocessing methods would be more effective given CP-nets with sparser structures and dominance queries with small Hamming distance. By including such parameters in our evaluations, we get a more detailed picture of preprocessing performance. In particular, we can more precisely evaluate where preprocessing will be most effective, as well as when it is not worth applying. This will show us more accurately where preprocessing should and should not be applied and, thus, help us to answer dominance queries as efficiently as possible.

In our experiments, we chose the most effective pruning method from Chapter 2 to answer the queries (both before and after preprocessing). This is one of the most efficient methods we tested in Chapter 2 and utilises the strongest pruning condition. The latter means that it has maximal overlap with preprocessing in how it reduces query complexity, as we discussed in §3.3.1. This ensures that our results represent a realistic impact of preprocessing on dominance query complexity. However, the proportional reduction in query complexity performed by preprocessing is dependent upon this choice of dominance testing method. Thus, although we have attempted to use the most reasonable choice of method in our experiments (see full discussion in §3.3.1), our results remain specific to this method (though we expect to find preprocessing equally or more effective when using the other methods considered in Chapter 2, due to our choice having maximal theoretical overlap). In the future, we would like to evaluate the impact of preprocessing when a variety of dominance testing methods are used.

Another choice we made in our experiments was to incorporate the initial conditions of our dominance testing method into the preprocessing methods. In particular, for all three methods, we checked these initial conditions for the original query and then again for each subsequent reduced query produced throughout the preprocessing procedure. If any conditions is met at any point, then we can conclude the original query to be false, meaning that no further preprocessing or dominance testing is required. This improves efficiency of preprocessing as the conditions are quick to check and can identify cases where no further reduction (or testing) is necessary. However, we did not evaluate the effect of this modification on our preprocessing.

In our future work, we would like to evaluate the impact of adding these checks, to see if they are worth implementing and what impact they have on efficiency. In particular, we would like to evaluate how often these checks find a query to be false (in non-trivial cases, where the original query cannot be answered by initial conditions), how much time is saved by these checks on average, and at what stage of the preprocessing do they find the query false. The latter will tell us whether it is worth checking every query produced by preprocessing or if, for example, only queries produced later on are generally found false by the checks. In our experiment, we used the initial conditions associated with our dominance testing method. However, as we discussed in §3.3.1, there are other initial conditions we can use. In our future work, we would like to consider using other initial conditions alongside preprocessing and evaluate which conditions give the most efficient dominance testing times.

When multiple queries are produced by preprocessing, we chose to answer them in increasing order of the number of variables in the associated CP-nets. Recall that, if one query is found to be false, then we can stop answering queries as the original query must also be false. If the original query is true, then all of the reduced queries must be answered. By answering in increasing CP-net size, we are aiming to avoid answering any more complex queries than necessary in the case where the original query is false. However, we did not explore whether this heuristic was successful at improving dominance testing efficiency. In our future work, we would like to explore how this ordering affects efficiency and whether there is a better ordering to use. For example, we might also consider using (in addition to CP-net size, or separately) the dominance query Hamming distance or the structural complexity of the associated CP-net (measured, for example, by graph density) as measures to order the reduced queries. These are different ways of, again, trying to avoid answering queries that are more complex than necessary.

Another approach we may consider is whether we can efficiently assess which of the reduced queries are more likely to be false. If we put such queries first in our ordering, then we are likely to determine the query to be false earlier and avoid answering the remaining queries. One heuristic we could use to determine whether a query is likely to be false is to consider how close the relative rank sizes (see Chapter 2) of the outcomes are (this would also suggest that the query is likely to be efficient to answer). Alternatively, if the original query is false, then perhaps the query with the largest CP-net is most likely to also be false as its associated CP-net contains the most of the original preference information. Similarly, perhaps the sub-CP-net containing the largest proportion of D (the set of variables that take different values in the original query) is the most likely to have a false query as the original dominance problem is primarily about the variables in D. A third approach we could consider is a tradeoff between ordering according to query efficiency and ordering according to the likelihood of a query being false. In our future work, we intend to consider these various approaches to ordering reduced queries and evaluate which leads to the most efficient dominance testing procedure for multiple queries.

In our future work we would also like to analyse which stages of UVRS (individually and when used in combination with forward pruning) actually have a significant effect, on average, on query complexity. For example, does repeatedly iterating unimportant variable removal remove enough outcomes on average to be worth the cost of repeated application, or would it be more efficient to only attempt removal a maximum of two or three times? By evaluating the progressive CP-net (and query) reduction performed by the different stages of UVRS and the combination, we can consider streamlining the processes by removing stages that cost more to apply than their average benefits. It will also help us to understand how and why they work as preprocessing techniques.

In our experiments, we evaluated preprocessing performance on simulated dominance queries, generated using the random CP-net generator from Chapter 2 (see Appendix C.1 for details). Naturally, this makes our results specific to the CP-net and query distribution produced by this generator (see Appendix C.1 for details). In our future work, we would like to evaluate preprocessing performance on real world data, so that we can see the true efficacy of preprocessing in practice.

In Chapter 2, we generalised our outcome ranks and all their applications, including efficient dominance testing, to the case of CP-nets with indifference. Having users express indifference is likely in real world scenarios, particularly when there are a large number of choices. Thus, this generalisation extends the applicability of our outcome rank results. In our future work, we would like to similarly generalise or adapt UVRS so that it can be applied to CP-nets with indifference. Boutilier et al. (2004a) claim that suffix fixing still applies in the case of indifference. The prefix fixing result, Proposition 3.4, does not hold in the indifference case. This is because variables in a matching prefix can vary between indifferent values, though they must start and end at the same values. However, by our assumptions about indifference in Chapter 2, switching between indifferent values does not impact child preference. This means that such changes do not affect any flips of variables outside of the matching prefix and, thus, omitting such flips does not affect the validity of any IFS. Therefore, even though matching prefixes do not have to remain fixed in all IFS, we do not need to consider any flips of variables in the prefix. This is sufficient to continue using prefix fixing in UVRS as before. As separation can also be applied when there is indifference (by the same reasoning as when there is no indifference), it appears that UVRS can be applied in the same way in this case.

However, we may be able to modify UVRS to be more effective in the case of indifference. For example, if a variable takes two distinct but indifferent (in at least one of the outcomes) values in the query outcomes, then we can exclude this variable from D, meaning that it may potentially be an unimportant variable that is, therefore, removed. This works because we can change such variables to be the same in both outcomes, without affecting whether there exists an IFS (as we are changing between indifferent values). This modification improves the reduction performance as it increases the possible number of variables we can remove. In our future work, we would like to consider whether UVRS can be modified in any additional ways to improve performance in the case of indifference. Boutilier et al. (2004a) claim that forward pruning can be applied in the case of indifference but do not explain how. Thus, we would also like to consider how to apply forward pruning in the case of indifference and whether any similar modifications can be made here. We then would like to evaluate the best method of applying a combination of these preprocessing methods in this case. Once we have determined the best method of applying preprocessing in this case, we intend to perform a similar experimental evaluation of their performance. We may also go on to modify preprocessing so that it is applicable to other extensions of CP-nets such as TCP-nets (CP-nets with additional relative importance statements) (Brafman et al., 2006).

As they are currently defined, all methods of preprocessing are dominance query specific. If you then have a second query to answer, you must preprocess the CP-net again from the beginning. This is not a huge problem as, even when applying preprocessing every time, we still save significant time on dominance testing, as we have seen from the experimental results. However, in our future work we might consider whether we can improve efficiency by preprocessing a CP-net for multiple queries at once. In order for such preprocessing to be effective, it is probably necessary for the group of queries to be sufficiently ‘similar’ to one another. This will ensure that there are aspects of the CP-net that are mutually irrelevant to the whole group of queries (for example, unimportant variables that take same value in all queries) and can, thus, be removed by our group preprocessing procedure. Another hurdle for this problem is the additional complexity of grouping queries, if necessary, and considering multiple queries in the preprocessing procedure. These additional complexities (and the fact that group preprocessing is likely to be less effective than single query preprocessing) will need to be outweighed by the improved efficiency of only preprocessing once.

Chapter 4

CP-Net Learning

4.1 Introduction

In Chapters 2 and 3, we addressed the problem of efficiently answering queries about user preference. In these scenarios, we have been assuming that the user's CP-net is known. However, in order for CP-nets to be useful in practice, even with the previously discussed improvements, we first must be able to determine what a given user's CP-net is. In their introduction to CP-nets, Boutilier et al. (2004a) argue that the compact and intuitive nature of CP-nets makes it possible to elicit them from non-expert users. However, in many cases, it is not ideal to require a user to specify their preference structure prior to using a given service. It is time consuming and may be off-putting to potential users, particularly if one considers that many existing services such as Netflix, Amazon, and Apple News can approximate user preferences for recommendations without explicit user input. Furthermore, there is the possibility of the user supplying incorrect or inconsistent preference information. Thus, we introduce a method of identifying a user's CP-net from observational data such as the products they buy or the movies they watch. This allows us to extract the user's preferences from an accurate source without affecting user experience. Even if initial preferences are collected from the user, these preferences may change over time and our method could be used to check or update the user's preference structure without querying the user further.

The remainder of this chapter is structured as follows. We first review the existing work on learning a user's CP-net in §4.2. In §4.3, we introduce a new method for learning CP-nets from observational data. We then present an experimental evaluation of the learning performance in §4.4. Finally, in §4.5, we discuss these results and how our learning method might be developed further in the future.

4.2 Related Work

In this section, we review the existing work on CP-net learning. We define CP-net learning to mean, given some data on user preference, finding a CP-net that represents or approximates the user's preference structure. As there are many approaches to this problem, we split the existing methods into categories. First we consider the type of preference data the methods use. The most popular choice in the literature is to use pairwise outcome preferences. We choose to use a different type of data, as we discuss in §4.3.1. We further split the methods into *passive* and *active* learning methods. A learning method is said to be passive if it learns a CP-net from the data alone, an active learning method may also (or instead) query the user about their preferences. Other labels that we may use to describe methods are *batch learning* and *online learning*. Batch learning methods start with the whole training data whereas online learning methods receive the data over time and repeatedly update the learned CP-net when new data is received.

In some works, the authors claim that their method can handle noisy or inconsistent data (Allen, 2016; Allen et al., 2017b; Haqqani and Li, 2017; Labernia et al., 2018, 2017; Liu et al., 2014, 2013; Liu and Liu, 2019; Liu et al., 2018a,b). Sometimes these terms are used interchangeably and other times they are distinct concepts. These terms could have several different interpretations; the data is not consistent with the user's preferences, the data is not consistent with a CP-net, or the data implies impossible preferences such as $x \succ x$. Further, these irregularities could be due to the user supplying inconsistent preferences, or errors in data collection. Which interpretation is intended is not always made clear in the literature. In §4.3, we explain what we mean by noise in our data and, in general, we will not use the term 'inconsistent' without specifying what we are referring to with regards to consistency.

4.2.1 CP-Net Learning from Pairwise Outcome Preferences

Passive Learning Methods

Dimopoulos et al. (2009) provide the first method for learning a user's CP-net. Their aim is, given a set of pairwise preferences over the outcomes, to obtain a CP-net that entails all of these preferences. They show that this is an NP-hard problem. In order to learn such a CP-net, they begin with an empty structure and try to add one variable at a time by finding a valid parent set, out of the variables already in the structure. They start by only adding variables with no parents, then go on to allow parent sets of increasing size. A set of parents is valid if there exists a CPT that ensures the outcome preferences are entailed. For this they use a sufficient, but not necessary, condition that is shown to be a 2-SAT problem (see Appendix F for definition) to check. This process continues until all variables are in the CP-net or the algorithm cannot add any more variables (which constitutes a failure). If a CP-net is produced, then it entails the given preference set. However, they cannot guarantee that, given such a CP-net exists, one will be found. This is only guaranteed in the case where the preference set is transparently entailed (see Appendix F for definition), which is a stronger form of entailment. This is problematic in real world applications: beyond trivial cases, it is not clear how one would check whether a given preference set is transparently entailed by a CP-net. It is likely that any method of confirming this would also enable the construction of such a CP-net, making learning useless. Thus, it is not clear how one could gather a (non-trivial) transparently entailed preference set. This means we cannot gather data for learning in a way that guarantees the learning algorithm will produce a CP-net. These problems remain even if the user's preferences are known to be representable by a CP-net and all supplied preferences are correct (entailed). When the data is transparently entailed, the algorithm works in polynomial time. Under similar conditions, the algorithm is shown to be a PAC-learner, that is, the learned CP-net is probably approximately correct (Valiant, 1984) – see Appendix F for how this concept is defined for CP-nets.

In Michael and Papageorgiou (2013), an empirical evaluation of the performance of this learning algorithm is provided. These experiments are restricted to the binary CP-net case, despite the fact that the original paper claims the algorithm can be applied similarly for CP-nets with multivalued variables.

Allen (2013) extends the work by Dimopoulos et al. (2009) so that CP-nets with indifference in their CPTs can also be learned. They also allow the CP-nets to be non-binary. Dimopoulos et al. (2009) claim that their method can be applied to the non-binary case, but do not give the details. Allen (2013) allow the preference data to be of the form $o \succ o'$, $o \bowtie o$ (incomparable), or $o \sim o'$ (indifferent). Each of these statements can be expressed by assigning a true or false value to each of $o \succsim o'$ and $o \precsim o'$. Thus, the data can be considered to be statements of the form $o \succsim o'$ or $o \not\succsim o'$. The only difference from Dimopoulos et al. (2009) in the algorithm is how they determine whether, given a hypothetical parent set, there exists a valid CPT. They construct a 2-SAT problem in the same way, only using $\succsim$ and $\not\succsim$, rather than $\succ$. However, as the variables may be non-binary, they introduce additional clauses to ensure transitivity within CPTs. This makes the task of determining whether a valid CPT exists into a 3-SAT problem, which is NP-complete. The algorithm may have to solve these 3-SAT problems exponentially (in n) many times. Note that Dimopoulos et al. (2009) do not mention this additional requirement for transitivity when they claim their method can be extended to the non-binary case. Allen (2013) does not prove the soundness or completeness of this new algorithm. We conjecture that, in the cases when a CP-net is returned, the preferences and incomparable statements in the data will be entailed, but the indifference statements will not always be entailed.

Liu et al. (2014, 2013) were the first to allow the preference data to contain inconsistencies. In Liu et al. (2013), they use chi-squared hypothesis testing on the noisy data to identify the structure. However, the theoretical foundation for this chi-squared testing is unclear; they assume that the distribution of variables is independent, that is, $\mathrm{Pr}(X = x, Y = y) = \mathrm{Pr}(X = x)\mathrm{Pr}(Y = y)$ However, as their data is pairwise preferences, it is not clear how these probabilities are defined. They later claim that, due to this independence, the following probabilities are equal. Let $U = V \backslash X$,and $\mathbf{u} \in \mathrm{Dom}(U)$.

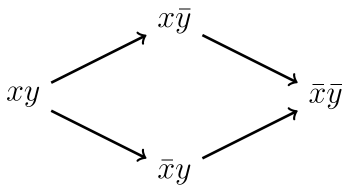
$$\mathrm{Pr}(o[U] = \mathbf{u} | o \succ o', o[X] = x, o'[X] = \bar{x}), \tag{4.1}$$

$$\mathrm{Pr}(o'[U] = \mathbf{u} | o \succ o', o[X] = x, o'[X] = \bar{x}). \tag{4.2}$$

There is no distribution assumed over the observed pairwise preferences, so we may assume that this probability is simply the sample frequency. Let us assume that the sample contains every distinct entailed preference (from the true CP-net) exactly once – so it is a noiseless sample. Then we can disprove this equality both in the case where X is preferentially dependent on U and where it is not – these are hypotheses of the chi-squared test where this probability is tested to determine dependence. We use two CP-nets in order to demonstrate a counterexample in each case. First consider a two variable CP-net with no edges. The variables are X and Y with CPTs $x \succ \bar{x}$ and $y \succ \bar{y}$. In this case, $U = \{Y\}$ and, if we let $\mathbf{u} = y$, Equations 4.1 and 4.2 become

$$\mathrm{Pr}(o[Y] = y | o \succ o', o[X] = x, o'[X] = \bar{x}),$$
$$\mathrm{Pr}(o'[Y] = y | o \succ o', o[X] = x, o'[X] = \bar{x}).$$

The preference graph for this CP-net is:



From this graph, we can see that the CP-net entails five distinct pairwise preferences. Of these, three preferences $o \succ o'$ satisfy $o[X] = x$ and $o'[X] = \bar{x}$:

$$xy \succ \bar{x}y, \quad xy \succ \bar{x}\bar{y}, \quad x\bar{y} \succ \bar{x}\bar{y}.$$

Out of these preferences, two satisfy $o[Y] = y$ and one satisfies $o'[Y] = y$. Thus, by our assumptions about the sample, we have

$$\begin{aligned} \Pr(o[Y] = y | o \succ o', o[X] = x, o'[X] = \bar{x}) =& \frac{2}{3}, \\ \Pr(o'[Y] = y | o \succ o', o[X] = x, o'[X] = \bar{x}) =& \frac{1}{3}. \end{aligned}$$

Therefore, the claimed equality does not hold in the case where X is not dependent on U. Now suppose we have the CP-net with structure $Y \to X$ and CPTs $y \succ \bar{y}$, $y : x \succ \bar{x}$, and $\bar{y} : \bar{x} \succ x$. The preference graph is now:

$$xy \longrightarrow \bar{x}y \longrightarrow \bar{x}\bar{y} \longrightarrow x\bar{y}$$

This CP-net entails six distinct preferences, two of which satisfy $o[X] = x$ and $o'[X] = \bar{x}$:

$$xy \succ \bar{x}y, \quad xy \succ \bar{x}\bar{y}.$$

Of these, both satisfy $o[Y] = y$, and only one satisfies $o'[Y] = y$. Thus, we have

$$\begin{aligned} \Pr(o[Y] = y | o \succ o', o[X] = x, o'[X] = \bar{x}) =& \frac{2}{2}, \\ \Pr(o'[Y] = y | o \succ o', o[X] = x, o'[X] = \bar{x}) =& \frac{1}{2}. \end{aligned}$$

Thus, the equality also does not hold if X is preferentially dependent on U. These simple counterexamples suggest that their claimed equality is incorrect.

Putting aside the above issue, the learning method by Liu et al. (2013) proceeds as follows. For each $X \in V$, they test whether X is preferentially dependent on $V \backslash X$ using a chi-squared test. If they are dependent, then they assign $\mathrm{Pa}(X) = \ V \backslash X$ and the CPT entries are determined by the frequency of opposing rules in the data; if $U = V \backslash X$, and the preference $\mathbf{u}x \succ \mathbf{u}\bar{x}$ occurs in the preference data more often than $\mathbf{u}\bar{x} \succ \mathbf{u}x$, then they assign the rule $\mathbf{u} : x \succ \bar{x}$ in CPT(X). All degenerate parents are then removed. This method is not guaranteed to produce an acyclic or a consistent CP-net in general, though it will converge to the true CP-net as the size of the preference data increases. Due to the requirements of Chi-squared hypothesis testing, this method is unsuitable for small data sizes.

In Liu et al. (2014), their aim is to find a CP-net that entails the maximum number of the supplied preferences. If the preferences are weighted, then the aim is to find a CP-net that satisfies the set of preferences with maximum weight. This is achieved by finding the optimal preference graph via a branch and bound search (along with certain pruning procedures) on a tree of all the possible preference graphs. Despite the fact that preference graphs and CP-nets are equivalent structures, the authors emphasise the fact that they are learning preference graphs and not CP-nets, even giving an algorithm for transforming the learned preference graph into a CP-net. They suggest that dominance testing and consistency testing are easier on preference graphs than CP-nets, which cannot be true as the two are equivalent. In general, preference graphs are actually more difficult to work with in practice, as they are exponentially larger than corresponding CP-nets (despite the fact they encode the same information).

The complexities of the above methods are, respectively, polynomial and exponential in the size of the preference graph (although Michael and Papageorgiou 2013 express scepticism regarding the claim that the Liu et al. 2013 method is polynomial). Note that the size of the preference graph is exponential in the number of variables. Thus, neither method is tractable and, as the authors comment, the method in Liu et al. (2014) is only appropriate in cases where there are a small number of variables. The latter would therefore not be of much use in practice, as real world problems are likely to have more variables than the algorithm can deal with in reasonable time. In fact, the performance experiments in Liu et al. (2013) are all done on binary CP-nets with up to five variables, and in Liu et al. (2014) they only consider the three variable case, which is almost the smallest possible non-trivial CP-net.

Both papers compare their methods to the algorithm described by Dimopoulos et al. (2009). Liu et al. (2013) find that their method performs better; however, due to the testing conditions, the algorithm by Dimopoulos et al. (2009) was not guaranteed to produce a CP-net. In fact, in some of the experiments, their algorithm resulted in failure over 90% of the time. Both of the methods by Liu et al. (2014, 2013) produce a CP-net every time. Thus, it is unsurprising that the method by Dimopoulos et al. (2009) was outperformed in these experiments. Note that, in the one experiment where the algorithm by Dimopoulos et al. (2009) succeeded in producing a CP-net more than 50% of the time, it actually performed better than the method from Liu et al. (2013). Liu et al. (2014) only compare their performance to Dimopoulos et al. (2009) in conditions where both are guaranteed to succeed in learning a CP-net. In these experiments, the method by Dimopoulos et al. (2009) performs similarly and sometimes slightly better than their method. Thus, while Liu et al. (2014, 2013) have expanded the applicability of CP-net learning to noisy data, they have not necessarily improved upon the techniques in the case of noiseless data.

Allen (2016), Allen et al. (2017b), and Haqqani and Li (2017) frame CP-net learning as an optimisation problem and use a set of pairwise outcome preferences, $\mathcal{P}$, as the data. Considering learning as an optimisation problem means that they can allow this preference set to contain noise. In all three cases, they are attempting to maximise the fitness function $f(N)$.

$$f(N) = \frac{|\{p \in \mathcal{P} | N \vDash p\}| - |\{p \in \mathcal{P} | N \vDash p'\}|}{|\mathcal{P}|},$$

where, if $p \in \mathcal{P}$ is the preference $o \succ o'$, then p' denotes the opposite preference $o' \succ o$. Optimising this fitness function is done by maximising the number of preferences in $\mathcal{P}$ that are implied by the learned CP-net and minimising the number that are contradicted by the learned CP-net. Note that these are not equivalent as it is possible that the learned CP-net does not entail either of p or p'. In this case, the preference is not encoded or contradicted by the learned CP-net, which could be considered as the neutral scenario. Allen (2016) and Allen et al. (2017b) give the same learning method, which restricts the search space to binary tree-structured CP-nets. This allows them to encode each possible CP-net as two vectors. They attempt to optimise the fitness score over this space using local search with random starting points and several restarts. Haqqani and Li (2017) use a genetic algorithm (see Appendix F for definition) to maximise the fitness function.

All three papers provide empirical evaluation of their techniques, but it is particularly worth mentioning that the experiments in Haqqani and Li (2017) allowed up to 100 variables in the CP-nets. In comparison, other empirical evaluations allow a maximum of 25 variables (recall that CP-nets grow exponentially with the number of variables). Thus, the fact that their method completes in reasonable time and with decent performance scores is a testament to the power of genetic algorithms in this application. The experiments by Haqqani and Li (2017) included a comparison to the methods by Liu et al. (2013) and Liu et al. (2014). All three papers use the same performance measures and the comparison by Haqqani and Li (2017) shows their method performs best, though it is slower for the cases with up to 10 variables. Haqqani et al. (2018) went on to apply their learning method to the problem of real world journey planning. Here they again provided a comparison to Liu et al. (2013) and Liu et al. (2014) and found that their method performs better.

Labernia et al. (2017) require the data to be swap preferences (preferences between outcome pairs that differ on exactly one variable), but they allow the data to contain noise. They introduce an online learning method that observes one preference at a time. As the preferences are between outcomes that differ on one outcome, each preference supports exactly one rule. Every time a preference is observed, a count is added to the appropriate rule and for each pair of opposing rules ($\mathbf{u} : x \succ \bar{x}$ and $\mathbf{u} : \bar{x} \succ x$), the rule with a higher count is used. Once enough swap preferences relevant to X are observed, they consider assigning additional parents to X if there are opposing rules in CPT(X) with similar counts. If this is the case, the parent that would improve the counts most is assigned, out of those choices that would not introduce cycles. This algorithm runs in polynomial time and, under certain conditions, is guaranteed to return the optimal CP-net. There is also an upper bound on the difference between the CP-net learned by this algorithm and the batch learning version.

The authors provide an experimental evaluation of the learning performance, including a brief comparison with the method by Guerin et al. (2013), which suggests their algorithm produces a higher rate of agreement with the data. However, the algorithm by Labernia et al. (2017) requires the data to be swap preferences, which are always entailed in some direction and, thus, would be determined by an ordering query. If the algorithm by Guerin et al. (2013) was trained on the same preferences, then the output should agree or disagree with every preference, there should not be any indecisive cases. However, the results show Guerin et al. (2013) has a positive percentage of indecisive cases, suggesting that the two algorithms were trained or tested on different data. If this is the case, then to some extent it could be the cause of the difference in learning performance.

Labernia et al. (2018) present a learning method similar to that by Koriche and Zanuttini (2010). However, they allow the user to provide inconsistent preferences. They learn from a set of pairwise swap preferences that may contain inconsistencies. They aim to learn an acyclic CP-net, which must therefore be consistent. As the user may provide inconsistent preferences, it may be impossible to learn a CP-net that agrees entirely with supplied preference set. Thus, the algorithm keeps a list, L, of swaps violated by the learned CP-net. They start with an empty CP-net. Like Koriche and Zanuttini (2010), if the learned CP-net does not agree with the user's preference set (ignoring L), a counterexample (swap) preference from the set is provided. If possible, a rule or parent is added to the CP-net in order to make it agree with the counterexample. However, if the opposite swap is entailed by the CP-net and it is not possible to add a parent to explain the counterexample, then the counterexample is added to L. If the learned CP-net now agrees with the user's preference set, then the process terminates, otherwise another counterexample is provided.

Suppose $o \succ o'$ is a counterexample that swaps variable X. When selecting a new parent, P, to make the CP-net agree with this counterexample, there are several criteria to determine which P to add. One is to minimise the number of user swap preferences, $o_1 \succ o_2$, that flip the same value of X as $o \succ o'$, but assign a different value to P. That is, where $o_1 \succ o_2$ is an X flip and $o_1[X] = o[X]$, $o_2[X] = o'[X]$, but $o_1[P] = o_2[P] \neq o[P] = o'[P]$. Unless P is the only parent of X, this is an unnecessary condition, as there is no reason two parent assignments cannot imply the same preference order over X. Also note that, as the user may supply inconsistent preferences, the order of counterexamples will affect the learned CP-net. If a false counterexample is generated prior to a true counterexample, then the CP-net could entail the incorrect preference or assign incorrect parents.

If the user's true preference order is representable by a CP-net, then this algorithm has time complexity polynomial in n, the size of the preference set, the time it takes to identify whether a counterexample exists, and 2^p (p is the maximum in-degree allowed in the learned CP-net). The latter means that in the worst case scenario ($p = n - 1$), the time complexity is exponential in n. They compare this method to Guerin et al. (2013) experimentally, showing that Guerin et al. (2013) is faster but their method returns a CP-net that entails a higher proportion of the user preferences. Note that, while Labernia et al. (2018) have access to the entire preference set, Guerin et al. (2013) learns from minimal information and has access only to the queries the user has answered so far. Thus, it is unsurprising that Guerin et al. (2013) is faster, but perhaps less accurate. Further, as we commented on Labernia et al. (2017), the Guerin et al. (2013) method has a positive number of indecisive cases in this comparison, indicating that it was trained or tested on general preferences, whereas Labernia et al. (2018) only consider swap preferences where indecisiveness is impossible. This may also explain the distinction in performance.

Liu et al. (2018a), Liu et al. (2018b), and Liu and Liu (2019) all present CP-net learning methods that are based upon identifying the structure by evaluating which variables are dependent in the data. In fact, Liu et al. (2018a) and Liu and Liu (2019) only learn the structure, not the CPTs. Liu et al. (2018a) suggest that the CPTs can be filled in afterwards using maximum likelihood estimation. However, in all three papers, the specifics of the learning methods are unclear.

In Liu et al. (2018a), they aim to determine the structure using chi-squared hypothesis testing. However, exactly what hypothesis test they are performing is not given explicitly. Their learning algorithm is also confusing as, within a loop over every pair of variables, it repeatedly initialises the CP-net structure with no edges. That is, for every new pair of variables, they re-set the structure to be empty, losing all previous progress. Furthermore, the proofs of completeness and correctness of this algorithm describe the process in a way that does not match up with the provided pseudocode. The authors claim that the algorithm runs in time polynomial in the number of variables and edges. In their performance experiments, they compare their method to Liu et al. (2013). These results show that Liu et al. (2013) is faster, but performs worse with respect to their 'similarity' measure. This measure shows the similarity between the structures learned from training and test data. Whilst this score demonstrates that a method performs consistently, it does not imply that the learned CP-net is an accurate representation of user preference. Furthermore, they only give results for two specific users, so we cannot see how the method performs in general.

Liu et al. (2018b) introduce an online learning approach. Pairwise preference relations are received in order and viewed in sliding windows of time. Note that this is not continuously sliding – there is a set granularity to how the window shifts. For each new window, certain statistics are recorded so that one has a cumulative database of information regarding all the preferences that have been observed so far. Their method for learning a CP-net, Algorithm 2, is iterated for each new window of data. Given a new window, for each pair of variables, their learning algorithm uses Algorithm 1 to calculate the degree of dependence between the variable pair. However, Algorithm 1 is defined to determine the relation between a variable pair, that is, whether one is a parent of the other. Note that even this is not entirely clear from the pseudocode as it takes the variable pair (U, V) as an input and yet, the algorithm iterates over all possible values for U and V. What is clear is that Algorithm 1 does not return a numerical value. However, Algorithm 2 specifies that Algorithm 1 is used to calculate dependency degrees. These degrees must be numerical values as they are then put into decreasing order. The degree of dependency is not defined anywhere else in the paper either. The algorithm goes on to add to the structure the dependencies that have (undefined) dependency degree over a certain threshold, removing any cycles that are created. The authors claim that this method is very efficient as it only deals with the data in the current window. Its complexity is polynomial in n, the number of windows, and the length of the windows. However, the authors do not provide proofs that the algorithm is sound or complete. Further, while they provide an experimental evaluation of the algorithm's efficiency, they give no results regarding its accuracy in representing user preference.

Liu and Liu (2019) aim to learn the CP-net structure only, from a set of swap preferences. To do so they define a function for each X that is maximised by a parent set U if the mutual information between U and X is high and U does not contain redundant information – this indicates that U is a good candidate set for the parents of X. Their learning method starts by identifying the variable with the highest level of mutual information with the empty set, that is, the variable most likely to have no parents. The authors provide an algorithm for performing this task; however, the method is unclear from the provided pseudocode. Firstly, it cycles through every pair of distinct outcomes, despite the fact that the rest of the calculations are not dependent upon a specific outcome. Secondly, within the loop over pairs of distinct outcomes, they use an 'if' statement that is only true if the pair of outcomes are equal. The result of this is that the whole loop should do nothing. The second part of the learning method aims to identify, using the above function, an optimal parent set for each variable. This algorithm also cycles through all pairs of distinct outcomes, despite the fact that the rest of the calculations are not outcome dependent. Furthermore, the given algorithm does not match up with the explanation used in the proof of correctness, nor with the illustrative example. Thus, the details of their learning method are unclear. The authors claim that their method always produces an acyclic CP-net that is locally optimal. Further, its time complexity is polynomial in n and the number of examples and its space complexity is linear in n. In their experimental results, they show that their method performs better than Liu et al. (2018a) in both efficiency and accuracy. Their measure of accuracy is the amount of mutual information captured by the learned structure in comparison to the true structure. As they are learning the structure only, they cannot assess directly whether the learned CP-net represents user preference – for example, whether the example preferences are entailed.

Active Learning Methods

Koriche and Zanuttini (2010) present two algorithms for CP-net learning, one for learning binary CP-nets from swap preferences, and one for learning binary tree structured CP-nets from arbitrary preferences. Both algorithms assume that the preferences supplied are always correct and that the user's preferences can be represented by a CP-net. They begin with the hypothesis of an empty CP-net. If a hypothesis does not represent the user's preferences, a counter example is supplied.

A small number of queries to the user are then used to add variables, parents, or rules to the hypothesis so that it becomes consistent with the counterexample. Note that users are only asked to provide swap preferences, which are more likely to be answered reliably. This process is iterated until a CP-net that accurately represents user preference is obtained. Both algorithms run in polynomial time, are attribute-efficient, and are shown to be quasi-optimal. By attribute efficient, the authors mean that the required number of queries is polynomial in $\log(n)$ (where n is the number of variables) and the size of the target CP-net (number of rules added to the number of edges in the CP-net). However, note that the latter may be exponential in n in a worst case scenario.

Guerin et al. (2013) also use pairwise outcome preferences as data and query the user as necessary. However, unlike Koriche and Zanuttini (2010), they may pose non-swap queries to the user. When queried, the user may respond with 'unable to decide', but the algorithm relies on the assumption that the provided preferences are consistent. The algorithm begins with a CP-net with no edges and each CPT(X) is elicited from the user directly. From here, the method is similar to that of Dimopoulos et al. (2009); all nodes begin classified as 'unconfident' and the algorithm attempts to reclassify them as confident by assigning a valid parent set, considering increasing parent set sizes. Given a hypothetical parent set, it is determined to be valid or invalid using the same condition as in Dimopoulos et al. (2009). However, there must also be sufficient evidence for the associated CPT. If there is not sufficient evidence, the algorithm poses a query to the user that is relevant, and then re-evaluates whether there is a valid CPT. The algorithm repeatedly attempts to assign parents in this manner until no more edges can be added. Unlike Dimopoulos et al. (2009), this algorithm will return the CP-net, even if there are still unconfident variables. This is positive because it means there is always a CP-net produced, however, it may not entail all of the preferences supplied by the user. Furthermore, it is possible that the supplied preferences only determine partial CPTs. In these cases, the remaining rows are filled with the variable's initial preference. Thus, even the confident variables may have inaccurate CPTs. This algorithm runs in polynomial time.

Alanazi et al. (2016) introduce a method for learning tree structured CP-nets by querying the user's preferences over swap pairs of outcomes. This algorithm relies on the supplied preferences always being correct and that the user's preferences can be represented by a CP-net. A conflict pair is defined as a pair of swap preferences (both entailed), where one swaps $X = x$ to $X = \bar{x}$ and the other swaps $X = \bar{x}$ to $X = x$. Such a pair demonstrates that X has a parent (recall that they are considering tree structures only, so X cannot have multiple parents) and that the parent takes different values in the two swaps. For each variable, Alanazi et al. (2016) use a series of swap preference queries to identify a conflict pair (if one exists). Given a conflict pair, they then use a small number of additional queries to identify which variable is the parent. The original queries can be used to fill in the CPT. In the binary case, this method requires fewer queries than the method for learning tree structured CP-nets by Koriche and Zanuttini (2010) – recall that they use both user queries and counterexample preferences in their learning. Alanazi et al. (2016) show that their method is close to optimal with respect to the number of required queries.

Alanazi (2016) expands upon this work and provides a method for learning general acyclic CP-nets from user queries over swap preferences. As the variables are binary, an unordered pair of outcomes that differ only on X is fully determined by the values taken by $V \backslash X$ – they call this the *context*. Thus, a preference query over an X swap is defined by the context. In order to learn a CP-net, they ask the user a predetermined set of queries that are sufficient to reveal a variable's parent set and CPT entries. This method again relies on the user always supplying correct preferences and that the true preference structure is representable by a CP-net. They also assume that the maximum number of parents is bounded by k. In order to determine the queries, they construct an $(n-1, k)$ universal set (see Appendix F for definition) and use this to provide the context for the queries. Note that the same set is used for each variable's swap queries. This process has time complexity polynomial in n and exponential in k, which, in the worst case scenario, makes it exponential in n. Alanazi (2016) also provides a slight variation for the cases where the user cannot answer, or answers incorrectly for a certain set of queries of limited size.

Alanazi et al. (2020) gives the same method as Alanazi (2016) for learning general acyclic CP-nets from user queries over swap preferences. However, in the 2020 paper, they use additional queries beyond those from the universal set. Alanazi et al. (2020) also adapts the learning methods in both the tree structure and general acyclicity cases in order to be able to learn incomplete CP-nets (that is, CP-nets with incomplete CPTs). In both cases, these adaptations increase the required number of user queries. In the tree-structured case, the algorithm now uses the same number of queries as the method by Koriche and Zanuttini (2010), which can also learn incomplete CP-nets. This improves upon the best existing upper bound on the required number of queries, given by by Chevaleyre et al. (2010). Alanazi et al. (2020) also show that the difference between their required number of queries (for both algorithms) asymptotically differs from the required number of samples for a PAC-learner by at most a factor of $\log(n)$.

4.2.2 CP-Net Learning from Other Data Types

All of the methods discussed in this section are passive learning methods.

Eckhardt and Vojtáš (2009, 2010) provide a method for learning a user model that is similar to a simple CP-net. The training data is a small set of outcomes to which the user has assigned a rating. Their user model has two parts; first, each variable has a local utility function that maps the value assigned to that variable to a value in $[0, 1]$. These functions map an outcome to a real vector. Then an aggregation function is used to map this vector to a value in [0,1] that indicates the user's preference for the outcome. The main focus of these papers is learning the local utility functions from the training data, particularly in the case where the utility function may be dependent upon the value taken by another variable (parent variable). The utility functions are estimated from the training data using different types of regression. In the dependent case, a separate utility function is estimated for each parent assignment. As they allow preferential dependency between variables, this is similar to the idea of learning user preferences for CP-nets. However, this method only considers dependency between two variables and they do not discuss how to detect this dependency between variables. In CP-net terms, this means that you would need to know the structure beforehand and no variable could have more than one parent. Furthermore, regression requires the variables to be numerical, which we do not require for CP-nets as they are qualitative models. The aggregation function is also not applicable to CP-nets as CP-nets can have multiple consistent orderings and so one cannot construct a utility function over the outcomes. Thus, while their model for user preferences has certain parallels to simple CP-nets, this method of preference learning cannot be applied to learning CP-nets in general as the other works in this section can.

Siler (2017) assumes the data to be conditionally optimal outcomes. He assumes the data comes in the form of a pair $(cond, opt)$, which means that under the condition *cond*, the outcome *opt* is optimal. This assumes that there is a true CP-net representation of user preferences, according to which all of these constrained optimality statements are true. The condition is assumed to be an assignment of values to some subset of the variables. This is a very restrictive assumption on the data, for example, in the non-binary case, the data cannot express the optimal value under the condition $X \neq x$. In the binary case, we cannot specify the optimal outcome under $X = x \vee Y = y$, for example.

The learning algorithm is almost identical to that given by Dimopoulos et al. (2009), except for the method of determining, given a hypothetical parent set U, whether a valid CPT exits. This is determined by checking all of the optimal outcomes in the data for the following property for each $\mathbf{u} \in \mathrm{Dom}(U)$; for all examples where $U = \mathbf{u}$ and the condition does not specify a value for X, X must take the same value. If there are no relevant examples, a row can be left blank and this is considered valid. If all relevant examples have $X = x$, then the row entry is $x \succ \bar{x}$ and similarly for $X = \bar{x}$. If every row is filled or left blank, then there is a valid CPT. If a CP-net is returned, it is acyclic and entails the conditionally optimal examples. Note that, if such a CP-net exists, then one will be returned. However, the returned CP-net is not guaranteed to be the truth or even a have a structure that is a subgraph of the true structure – a variable may be assigned a completely incorrect parent set by this process. Further, as the author notes, this method is only tractable when the imposed upper bound on parent set size is small.

Khoshkangini et al. (2018) use historical user outcome choices as data, the same as in our work. Their aim is to obtain a CP-net to use for recommender systems. They assume that in this scenario there is some 'target' variable for which we want to predict user preference and this variable is part of the CP-net. This is a somewhat unusual assumption as such variables would determine the value of all other variables. Hence, it is not clear how to interpret a conditional preference over this variable. One of their examples is movie recommendation and they suggest that the target variable would be film title. Knowing the film surely determines all other features such as year of release or genre. Thus, having the title as a variable within the CP-net seems somewhat at odds with CP-net semantics. Further, in this example, your data would only contain films the user has watched previously. By including titles as a variable, we limit the CP-net to considering previous choices when, naturally, one would want to recommend new films to the user.

The authors use information gain (with respect to the target variable) to perform feature selection on the other variables. This reduces the the variable set to a given size. A CP-net over this reduced variable set is constructed by building three layers. The first layer is the root layer, which consists of the variable with the highest information gain only, this is set to be the root of the CP-net and has no in-degree. The second is the intermediate layer, which consists of the remaining selected variables (not including the target). The final layer is the target layer which contains the target variable only, which has no out-degree. Note that these layers are in topological order; edges between layers must go from the root layer to the intermediate or target layers, or from the intermediate to target layer.

The edges within the intermediate layer are constructed as follows; two nodes are initially connected if their dependence in the data is above a given threshold. They then use hill climbing search to identify the Bayesian network over the intermediate variables which optimises various metrics used in Bayesian network learning. Bayesian networks have conditional probability tables that give, for every assignment to the parent variables, a probability distribution over the variable values. They convert their learned Bayesian network into a CP-net by simplifying these probability distributions into a strict ordering to form conditional preference tables. However, Bayesian networks and CP-nets are distinct due to their differences in symmetry (as well as other properties). By learning a Bayesian network, you are representing the probabilistic dependence in the data, which is symmetric. CP-nets represent preferential dependence, which is asymmetric. Thus, there is no guarantee that their learned Bayesian network has the same orientation as the user's CP-net. The edges between layers are determined by evaluating which variables have a sufficient dependence level with the root and target nodes. Note that edge direction in these cases is enforced by the fact the layers are in topological order. This CP-net construction is entirely based upon probabilistic dependence. Thus, for the reasons discussed above, the resulting structure is more of a simplified Bayesian network than a true CP-net representation.

Several papers have proven results about the complexity of learning CP-nets from outcome preferences under a variety of conditions (Alanazi, 2016; Alanazi et al., 2016, 2020; Chevaleyre et al., 2010; Koriche and Zanuttini, 2010; Lang and Mengin, 2008, 2009). These include both passive and active learning (where it is generally concluded that active learning makes learning possible in reasonable time), learning CP-nets with specific structure types, learning both complete and incomplete CP-nets, and cases where users provide incorrect or incomplete information. In many cases, these complexity results are proven by transforming the learning problem into a SAT problem (see Appendix F for definition).

4.2.3 How and Why Our Method Differs from Existing Work

In §4.3, we introduce a new method for passively learning acyclic binary CP-nets. We choose to learn passively as this is less intrusive to the user and we do not have to rely on the accuracy of user answers. Existing technology can estimate user preferences without explicit user input. Thus, requiring users to specify their preferences could be off-putting. Passive learning also allows us to learn the preference structure of a user in the event that the user is unwilling to reveal their preferences, for example in an auction or in general when playing a game against an adversary.

Our method is less restrictive in its assumptions than many of the works discussed here; we do not assume that the user's true preferences are representable by a CP-net, we do not impose any restrictions on the structure of the learned CP-net (beyond acyclicity), and we do not assume that the data is always consistent with the true preference order (noiseless). There are other distinctions from the existing work, which we discuss in §4.3. However, the main difference is that, unlike the vast majority of the above methods, we do not use pairwise outcome preferences as data. Instead, we use a history of outcomes chosen by the user, similar to the data used by Khoshkangini et al. (2018). It is more realistic that one could passively observe this type of data, as opposed to pairwise preferences, which makes our method more widely applicable.

In many contexts, there is no way to observe pairwise preferences without presenting pairwise outcome queries to the user. In most online scenarios (this could be shopping, social media, or content platforms such as Netflix or YouTube), a user has far more than two options and we observe only which outcome was successful, that is, which option they chose. Furthermore, if a user chooses outcome a out of the available options $\{a, b, c, d\}$, this does not automatically imply strict preferences $a \succ b$, $a \succ c$, $a \succ d$; if a user was presented with the same set of song choices on two separate occasions, they may make different selections. This could be because of the user's mood or activity or other unobservable variables – we discuss this further in §4.3. Thus, we cannot conclude that the chosen outcome is strictly the most preferred. Hence, in such scenarios, we cannot observe pairwise outcome preferences, but we can observe the user's choice.

In some scenarios, one may be able to observe or extract a set of initial pairwise preferences from the user. For example, the first time you set up a news app or create a Netflix account, you may be asked to answer some questions about your general preferences. In this scenario, one of the existing learning techniques can be used to learn an initial CP-net, N. Then, using N as a starting point, we can use our learning algorithm to update and fine tune the CP-net model as more user actions are observed, without needing to further query the user. We discuss the idea of using our learning method to update a model further in §4.3.2.

Siler (2017) uses conditional optimality statements, rather than pairwise outcome preferences as data. Such statements give the optimal outcome under a partial variable assignment. However, as we argued above, observing the choice of a from the available options $\{a, b, c, d\}$ does not mean that a was optimal in this case. Further, as we discussed in §4.2.2, requiring the condition to be a partial variable assignment is restrictive. It may not be possible to restrict to the set $\{a, b, c, d\}$ using a partial variable assignment. Thus, this data type is more restrictive than user choices and less realistic to passively observe. Furthermore, Siler (2017) require the user's preferences to be representable by a CP-net and that all conditional optimality statements are true under this CP-net. Our learning method does not require these assumptions to hold, as we discussed above.

Khoshkangini et al. (2018) uses the same data as we do. However, we do not use their concept of a 'target' variable. We assume the variables to be distinct features of the outcomes and our interest lies in determining user preference over the outcomes, as is usual for CP-nets. More importantly, we do not base our learning upon probabilistic dependency like Khoshkangini et al. (2018). As we mentioned above, constructing Bayesian networks and using probabilistic dependence may lead to incorrectly oriented structures. Our learning is instead based upon a score that we construct, which is motivated by the agreement between the data and the conditional preference rules encoded by CP-nets. We learn by identifying a CP-net that maximises this score.

4.3 A New CP-Net Learning Method

In this section, we introduce our method for passively learning binary CP-nets, which is distinct from the existing work because we use a history of user choices as data. As we are learning binary CP-nets, for the remainder of Chapter 4 we assume all CP-nets to be binary. This section is structured as follows; in §4.3.1, we discuss the data format. In §4.3.2, we introduce our scoring function for CP-nets as we utilise score-based learning. In §4.3.3, we give our CP-net learning algorithm. Finally, in §4.3.4, we discuss a possible variation to this method.

4.3.1 Historical User Choice Data

Most methods in the existing literature utilise pairwise outcome preferences as data for learning. However, as we discussed in §4.2.3, in many cases, it is not possible to passively observe such data. Instead, we can only see which outcome the user chose (for example, which item they bought or which film they watched).

We learn CP-nets from a history of such user choices. Passively observing this type of data is more realistic than pairwise preferences. This is particularly true when the authors required the data to be swap preferences or transparently entailed preferences.

Many of the methods described in §4.2 require one or both of the following. First, that the user's underlying preference order can be represented by a CP-net. Second, that all supplied preferences or data are consistent with the user's true preferences – they cannot handle noise in the data. We do not enforce either of these assumptions.

We assume that the universe of our user is deterministic, meaning that, if we could observe all of the variables, we would always know which item they would pick. However, we wish to model the user's preferences over a given set of observable variables only. There are likely to be external, possibly unobservable variables that affect the user's choices over this set of variables. We model the effect of external variables as noise. Due to this effect, we do not expect the user to pick the optimal outcome every time, but rather we expect the user to pick the optimal outcome the most often.

Example 4.1. Suppose we are trying to model a user's music preferences. The two variables of interest are the style of music, which could be Rock or Country, and the tempo of a song, which can be Fast or Slow. Suppose the user prefers rock to country. Suppose also that, if they are listening to a rock song, they prefer a fast tempo, but, if they listen to country, they prefer a slow tempo. This is a reasonable set of preferences and it is representable by a simple CP-net that has fast rock songs as the optimal outcome. This CP-net implies that the user will always choose fast rock songs when given the option. However, this is not how people act; they may have a preference for rock over country but still like and listen to both. This general preference for rock over country may be affected by additional information, for example, if the user is studying, they prefer to listen to country music. Thus, variables outside of our model can cause the user to pick non-optimal outcomes. However, one would assume that the general preference for rock means that overall the user picks rock more often than country. Carrying on this logic, one would thus expect the user to pick the optimal outcome most often.

In this work, we assume that the relevant observable variables are specified from the problem context. We leave the task of identifying the relevant factors from the observable data to future works. If we do not consider a variable, Y, that is a true parent of X in our learning, then our algorithm may be unable to detect some other true parents of X due to this missing information. Alternatively, the relationship between X and its other parents may be characterised incorrectly as we cannot distinguish between the different Y assignment cases. However, we would not expect this to impact variables that are not children of Y. Unfortunately, such effects are unavoidable if Y is not a variable we can observe.

Remark. Note that a user's preference and what they choose are not necessarily one and the same. For example, suppose we have two types of sandwich a and b, but b is much more expensive. The user may prefer b to a and yet they pick a most of the time. Thus, from user choice data, we cannot identify the user's preferences over sandwiches alone. Rather, we can only observe their sandwich choices under unknown contexts (assignments to the variables not in our model), which may not be a reflection of the former as preference may depend upon external variables. However, the point of modelling user preferences is usually to be able to reason about and predict what the user would pick, not what they prefer. For this purpose, it would be appropriate to model a as preferred to b. Thus, we can use the probability of an outcome being chosen as a proxy for preference.

We assume, in this deterministic universe, that for every outcome o_i associated with our set of observable variables, there is some true proportion of times p_i where the user picks o_i. We assume these p_i to be fixed, meaning that the user's preferences are not changing. Thus, if a user's preferences are likely to change, then one might apply this learning to only a limited history of user choices. We also assume that all outcomes are available each time a choice is made and that each choice is independent.

Let $d(o_i)$ be the number of times o_i is chosen in the data and let $\mathcal{O} = 2^n$ be the total number of outcomes. By the above assumptions, we may conclude that the $d(o_i)$ values have the following multinomial distribution:

$$(d(o_1), ..., d(o_{\mathcal{O}})) \sim MN(\mathcal{O}, (p_1, ..., p_{\mathcal{O}})). \tag{4.3}$$

These p_i values give us an ordering over the outcomes induced by how often they are picked. As we mentioned above, we are using this property as a proxy for preference. Thus, we have a linear preference ordering. Note that we do not assume that this preference ordering is representable by or consistent with a CP-net. Furthermore, if the p_i are consistent with some 'true' CP-net, the data may still contain non-optimal outcomes, which is contradictory to the CP-net (noise).

There are several different aims used in CP-net learning. Some authors try and learn a CP-net that can predict user preferences well (Dimopoulos et al., 2009; Guerin et al., 2013; Labernia et al., 2018, 2017; Michael and Papageorgiou, 2013). Others aim to recover the true CP-net (Alanazi, 2016; Alanazi et al., 2016; Koriche and Zanuttini, 2010). Some aim to learn a structure that entails some or all of the training set, whether that is outcome preferences or optimal outcomes (Allen et al., 2017b; Haqqani and Li, 2017; Liu et al., 2014; Siler, 2017). Lang and Mengin (2008) define three levels of agreement between the learned CP-net and a set of preference examples; the example set is 'implied' when all of the examples are entailed, it is 'strongly consistent' when all examples are included in a single consistent ordering, and it is 'weakly consistent' when every example is contained in some consistent ordering (that is, their reverse is not entailed).

As the p_i induced ordering of the outcomes is the user's true preference order, we would like our learned CP-net to agree with this ordering. Very few CP-nets entail a total order over the outcomes, so looking for a CP-net that implies the ordering is too strong. Ideally, we would like to learn a CP-net for which the p_i ordering is a consistent ordering. However, some linear orders are not consistent with any acyclic CP-net. As we do not require the user's true preference to be representable by or consistent with a CP-net, obtaining strong consistency is not always possible. Thus, we aim to learn a CP-net that is weakly consistent with the p_i outcome ordering.

Remark. In the previous chapters, we have considered CP-net outcomes to be the products or scenarios that the user is deciding between. However, if we consider an online store or a content platform like Netflix, in order for every product or film to be its own outcome, the CP-net would need a large number of variables or variables with large domain sizes, or both. Such a CP-net could be impractical to deal with and the user is unlikely to be able to specify all of the necessary preference rules, perhaps due to indifference or because the CP-net contains outcomes or variable values that the user knows nothing about. Further, users do not often buy the exact same product or watch the same film repeatedly, even if they really like it. In our learning method, we use the $d(o_i)$ values to represent the user's preference for outcome o_i (approximately). Thus, for CP-net learning we shall consider a slightly different interpretation of CP-nets. Instead of products, the outcomes will be categories of products. Take the Netflix example, instead of films, the outcomes will be categories of films or programmes. For example, 'British crime TV series' is a category specified by the three variables 'country of origin', 'genre', and 'film or TV'. In practice, one can make the categories more fine grained by using more variables or larger domain sizes.

In general, this interpretation of CP-nets will lead to smaller models and the user is more likely to have preferences over all of the domains (which should make it easier to detect the preference structure from the data). Furthermore, in this interpretation, it makes sense for $d(o_i)$ to be larger if the user prefers outcome o_i (that is, prefers items in category o_i). Once a user's preferences over the relevant categories are known, one could use a convenient or generic measure for sorting the products within a category. Netflix may sort the films within a category by the most recently released or added. Spotify may sort artists within a category based on whether it is liked by a user's friends, or by general popularity. Online retailers may sort a category of products by price, or by how often the products are purchased in general, or they may promote the products they need to improve sales for. Alternatively, once the preferred categories are identified, one could employ more intricate preference learning on the more preferred categories only (where there is likely to be more relevant data).

4.3.2 CP-Net Scoring Function

In this section, we define our scoring function for CP-nets, which is utilised in our score-based learning method in §4.3.3. There are two existing CP-net learning methods that use data similar to ours (Khoshkangini et al., 2018; Siler, 2017). Siler (2017) did not use score-based learning. Khoshkangini et al. (2018) used score-based learning for a part of their learning procedure, but this was in order to learn a Bayesian network. Thus, this is the first time a function has been defined that evaluates the agreement between a CP-net and user choice data.

The p_i values are all in the range $[0, 1]$ and, by definition, they must sum to one. This set of values is exactly the support space for an $\mathcal{O}$ dimensional Dirichlet distribution. Furthermore, such a Dirichlet distribution is conjugate with our multinomial data distribution. Thus, an appropriate prior distribution for these p_i values is the following Dirichlet distribution:

$$p_1, p_2, ..., p_{\mathcal{O}} \sim Dir(\beta_1, \beta_2, ..., \beta_{\mathcal{O}}).$$

In this work, we generally assume an uninformed Dirichlet prior is used. In particular, in our experiments we set $\beta_i = 0.01$ for all $i \in \{1, 2, ..., \mathcal{O}\}$. However, if our learning technique was applied as a method of updating a user's preference model, or in a context where one has prior knowledge of user preferences, an informed prior could be used by setting the β_i parameters to reflect this prior information.

In selecting an uninformed prior for our experiments, we wanted to use a small positive β_i value to reflect the circumstance of having observed no user choices and having no prior beliefs. We chose the value $\beta_i = 0.01$ arbitrarily, as any value close to 0 would have been appropriate. Due to time and computational constraints, we were unable to test other choices of uninformed prior parameter in our experiments. Thus, we cannot comment on the effect of this choice. We could also use larger values for the uninformed prior parameters, for example setting all $\beta_i = 1$, however this would correspond to the situation where the user picks every outcome exactly once, which is inaccurate. On the other hand, using larger parameters may make the prior more robust to small amounts of data. Tuning the β_i parameters and evaluating the effects of our choice of prior (both in the uninformed and informed case) is an important next step in our future work on CP-net learning, as we discuss in §4.5.

If we have prior beliefs about user preferences, then we would choose β_i to reflect this information (as we will see later, prior beliefs about variable relations can also be incorporated in the starting structure of our learning algorithm). If, for example, we believe the user prefers the property $X = x$ to $X = \bar{x}$, then one could make the β_i parameters corresponding to $o_i[X] = x$ larger than those corresponding to $o_i[X] = \bar{x}$. Alternatively, suppose we are using our CP-net learning algorithm as an update procedure. In this case, we already have a CP-net model of user preference, N_1, that we wish to update given newly observed data. In this case, we would use the existing CP-net as a starting structure for our algorithm and we could set the β_i values to reflect the preference order entailed by N_1 (larger β_i values for more preferred outcomes). If prior beliefs can be expressed via pairwise outcome preference, then another approach to configuring an informed prior could be to use an existing learning technique to obtain a starting CP-net model, which we then reflect in the Drichlet parameters (and starting structure), as in the updating case. The exact details of how to encode the various forms of prior information in our Dirichlet parameters is not examined here, this is another direction for future work.

Once the data (history of user choices) is observed, we can use the Dirichlet-Multinomial conjugacy to update the prior and obtain the posterior distribution:

$$p_1, ..., p_{\mathcal{O}} \sim Dir(\beta_1 + d(o_1), ..., \beta_{\mathcal{O}} + d(o_{\mathcal{O}})). \tag{4.4}$$

Thus, from the data, we obtain a distribution over the p_i values for which we aim to learn a consistent CP-net. We use this distribution to construct a scoring function that reflects how strongly a given CP-net agrees with the p_i values.

We take a Bayesian approach to modelling beliefs about the p_i values for several reasons. First, the Bayesian updating framework allows for expert knowledge about user preferences to be incorporated by utilising an informed prior, as we discuss above. Another advantage to the Bayesian approach, particularly in the case of no prior information, is that it enables us to encode a level of uncertainty around the observed proportions. This is essential in general as we have only a limited number of observed user choices and so we can only estimate the p_i values. In particular, this uncertainty ensures that outcomes that have not yet been chosen do not get written off as impossible. This is important as the exponential number of outcomes makes it likely for there to be many outcomes not chosen in our sample of user choices. The Bayesian model also provides a simple procedure for updating our beliefs given further observed data. In this scenario, one would use the previously learned CP-net as a starting structure and then perform learning again with the updated posterior. However, this convenient update procedure for the Dirichlet distribution does not allow us to adapt our learning procedure into an online learning method, as we explain in §4.5. This is another direction to be explored in our future work.

We will now formally define the scoring function for a given CP-net, N over variables, V. Suppose that $X \in V$ has parent set $U \subseteq V$ and let $W = V \backslash U \cup \{X\}$. A rule in CPT$(X)$ would then be of the form $\mathbf{u} : x \succ \bar{x}$ for some $\mathbf{u} \in \mathrm{Dom}(U)$. This rule represents $2^{|W|}$ many pairwise preferences. In particular, it dictates (due to the *ceteris paribus* nature of CP-nets) that, for every $\mathbf{w} \in \mathrm{Dom}(W)$, we have $\mathbf{u}x\mathbf{w} \succ \mathbf{u}\bar{x}\mathbf{w}$. We say that the preference $o_i \succ o_j$ is *supported* if $p_i > p_j$. Thus, the rule $\mathbf{u} : x \succ \bar{x}$ is supported if all of the $2^{|W|}$ associated pairwise preferences are supported. However, as the number of preferences represented by a rule is exponential, this is a complex condition to check and so we relax it slightly. This simplification also makes it possible for preference orders that are not representable by CP-nets to support preference rules.

Definition 4.2. Let N be a CP-net over variables V and let $U \subseteq V$ be the parent set of $X \in V$. Let us denote $W = V \backslash U \cup \{X\}$. For any $\mathbf{u} \in \mathrm{Dom}(U)$, we say that the rule $\mathbf{u} : x \succ \bar{x}$ is *supported* if

$$\sum_{\mathbf{w} \in \mathrm{Dom}(W)} p_{\mathbf{u}x\mathbf{w}} > \sum_{\mathbf{w} \in \mathrm{Dom}(W)} p_{\mathbf{u}\bar{x}\mathbf{w}}. \tag{4.5}$$

Note that we are sightly abusing notation here, using p_{o_i} to mean p_i.

However, we do not have the exact p_i values, but rather a distribution representing our beliefs about these values (Equation 4.4). Thus, while we cannot definitively determine whether or not a rule is supported, we can calculate the probability. We define the score of a rule to be the probability that it is supported.

Definition 4.3. Let N be a CP-net over variables V and let $U \subseteq V$ be the parent set of $X \in V$. Let us denote $W = V \backslash U \cup \{X\}$. For any $\mathbf{u} \in \mathrm{Dom}(U)$, the *rule score*, S_r, of the rule $\mathbf{u} : x \succ \bar{x}$ is

$$\begin{aligned} S_r(\mathbf{u} : x \succ \bar{x}) =& \Pr(\mathbf{u} : x \succ \bar{x} \text{ is supported}) \\ =& \Pr\left(\sum_{\mathbf{w} \in \mathrm{Dom}(W)} p_{\mathbf{u}x\mathbf{w}} > \sum_{\mathbf{w} \in \mathrm{Dom}(W)} p_{\mathbf{u}\bar{x}\mathbf{w}} \right). \end{aligned} \tag{4.6}$$

We say that a given CPT(X) is *supported* if all of the rules it contains are supported. However, as we cannot directly determine whether or not a CPT is supported, we define the score of a CPT to be the probability that it is supported.

Definition 4.4. Let N be a CP-net over variables V and let $U \subseteq V$ be the parent set of $X \in V$. Let us denote $W = V \backslash U \cup \{X\}$. We say that CPT$(X)$ is *supported* if each rule contained in CPT(X) is supported. That is, if

$$\bigwedge_{\mathbf{u}:x_1 \succ x_2 \in \mathrm{CPT}(X)} \left(\sum_{\mathbf{w} \in \mathrm{Dom}(W)} p_{\mathbf{u}x_1\mathbf{w}} > \sum_{\mathbf{w} \in \mathrm{Dom}(W)} p_{\mathbf{u}x_2\mathbf{w}} \right).$$

Definition 4.5. Let N be a CP-net over variables V and let $U \subseteq V$ be the parent set of $X \in V$. Let us denote $W = V \backslash U \cup \{X\}$. The *CPT score*, S_t, of CPT(X) is

$$\begin{aligned} S_t(\mathrm{CPT}(X)) =& \Pr(\mathrm{CPT}(X) \text{ is supported}) \\ =& \Pr\left(\bigwedge_{\mathbf{u}:x_1 \succ x_2 \in \mathrm{CPT}(X)} \left(\sum_{\mathbf{w} \in \mathrm{Dom}(W)} p_{\mathbf{u}x_1\mathbf{w}} > \sum_{\mathbf{w} \in \mathrm{Dom}(W)} p_{\mathbf{u}x_2\mathbf{w}} \right) \right). \end{aligned} \tag{4.7}$$

This probability cannot be calculated exactly from the Dirichlet distribution due to the form of the associated integrals. In practice, we estimate this probability using Monte Carlo estimation (Robert and Casella, 2004). A full description of this process and an evaluation of the estimation accuracy can be found in Appendix D.1.

If a CPT contains a degenerate parent, then removing this parent will increase the S_t score. This is an important property as it implies that the CP-net learned by our algorithm will not contain any degenerate parents, as we explain in §4.3.3. That is, our learning algorithm will produce the learned CP-net in its simplest form, which, as we have seen in Chapter 3, makes the CP-net easier to reason with.

Proposition 4.6. *Let N be a CP-net over variables V. Let $Pa(X) = U \cup \{Y\}$, where $X, Y \in V$, $U \subseteq V$, and $Y \notin U$. Suppose that Y is a degenerate parent of X. Let CPT_1 be the current $CPT(X)$ and let CPT_2 be the $CPT(X)$ obtained by removing Y as a parent, as we did in §3.2.1. Then we have*

$$S_t(CPT_2) \geq S_t(CPT_1).$$

Proof. See Appendix E.10.

Remark. When calculating the score of CPTs, we use the joint probability of all of the contained rules being supported. We use this joint probability as we cannot assume from CP-net semantics, or the Dirichlet distribution, that these preference rules are probabilistically independent. However, when scoring a whole CP-net, we consider each variable's CPT score independently (although we cannot assume that these probabilities are independent either).

The score of a CPT is the probability that all of its rules are supported. If r_1 and r_2 are rules within the same CPT(X), then their support conditions both dictate restrictions upon the same partition of the p_i variables. In particular, the partition of the p_i variables by their assignments to Pa(X) (the parent set associated with the CPT containing r_1 and r_2) and X. However, if r_1 and r_2 are rules in distinct CPTs, then their support conditions assert requirements over two different partitions of the p_i variables (corresponding to the appropriate parent sets and variables of their CPTs). Thus, rules within the same CPT are likely to have more directly dependent probabilities than rules in different CPTs. We take the former dependence into account by using the joint probability for CPT scores. The latter is what we are omitting by treating CPTs as independent from one another. In order to consider this dependency in our scoring function, we would need to use the joint probability of the whole CP-net. This would be a much more complex scoring function and would need to be recalculated from scratch for every new CP-net. By treating CPTs independently, we can work over the space of acyclic structures rather than the space of acyclic CP-nets, as we discuss below, which massively reduces the number of candidate models we need to consider. Further, we can update the score if the structure changes, by recalculating S_t for the variables that have lost or gained a parent only. Our learning algorithm repeatedly updates the CP-net score given minor structural changes. Thus, this simplification will reduce the complexity of our algorithm and improve its practical applicability.

Definition 4.7. Let N be a CP-net over variables V. The *CP-net score*, S_c, of N is

$$S_c(N) = \prod_{X \in V} S_t(\text{CPT}(X)). \tag{4.8}$$

Note that, for a given structure, this score is maximised by maximising the individual CPT scores. As we calculate S_t values through Monte Carlo estimation, finding (one of) the highest scoring CPT for X is almost as easy as determining the CPT score of a specific table (we discuss this in more detail in Appendix D.2). Thus, instead of considering specific CP-nets, we will consider only the structure from now on. Given a structure, we always assume that the optimal CPTs are utilised. We formally define the score of a given structure as follows.

Definition 4.8. Let G be an acyclic structure over variables V. Let $\text{Pa}(X)$ denote the set of variables that are parents of $X \in V$ in G. The *structure score*, S, of G is

$$S(G) = \prod_{X \in V} \max_{\text{CPT}(X) \in \mathcal{T}(\text{Pa}(X),X)} \{S_t(\text{CPT}(X))\}, \tag{4.9}$$

where $\mathcal{T}(U, X)$ denotes the set of all possible CPTs for variable X, given that it has parent set U.

Note that $|\mathcal{T}(U, X)| = 2^{2^{|U|}}$. Thus, by considering only the CP-net structures, we have massively reduced the number of candidate models we must consider in our learning process.

Our learning method aims to find a CP-net (structure) that maximises this score. Such a CP-net has the maximum probability that the preference rules it represents are supported by the user's true preference order. Note that it is unlikely that we will obtain a CP-net with a score of 1 for several reasons. Firstly, we do not require the user's preferences to be representable by a CP-net. Secondly, by representing our beliefs about the p_i values via a Dirichlet distribution, we have encoded a level of uncertainty and, thus, the relevant probabilities will not be equal to 1. They may get close if there is a lot of data or a strongly informed prior is used. Finally, even if we have a score of 0.9 for every CPT and there are five variables, then $S = 0.9^5 = 0.59$. That is, even small degrees of uncertainty are amplified by the multiplication process.

4.3.3 CP-Net Learning Algorithm

In this section, we describe our algorithm for CP-net learning. We use a score-based approach to learning, aiming to maximise the score given in Equation 4.9 over the space of acyclic structures. To do so, we utilise a greedy search algorithm, which guarantees that we obtain a locally optimal structure.

Note that we are searching over the space of all acyclic structures (DAGs) over our variable set V. This means that we are not imposing any restrictions upon the structures we can learn. Note that the existing works on CP-net learning often utilise assumptions such as the learned CP-net must be tree-structured or the maximum in-degree is bounded by some constant k.

The pseudocode for our learning algorithm is given by Algorithm 4. The learning process begins with a specified acyclic starting structure, A. In our experiments, we either start with the empty structure (with no edges) or a randomised structure. If one had prior knowledge, the structure could be populated with any known dependencies.

Let $\mathcal{G}$ be the space of all directed acyclic graphs over our variable set V. Let us first define the notion of neighbouring graphs within this space.

Definition 4.9. Let $G, H \in \mathcal{G}$. Then H is a *neighbour* of G if they differ on exactly one edge. That is, there exists some directed edge, e, such that $e \in G$ and $e \notin H$ or vice versa. Further, for any edge $e' \neq e$, we have that $e' \in G$ if and only if $e' \in H$. We denote the set of neighbours of G by $\mathrm{Ne}(G)$.

This notion of neighbours is used to move around the space $\mathcal{G}$. The general design of our learning algorithm is as follows; given the current structure, the algorithm identifies the best scoring neighbour and moves to this neighbouring structure. This continues until none of the neighbours of the current structure are an improvement upon the current score. Thus, the learned CP-net will be locally optimal according to this structuring of the space $\mathcal{G}$ (that is, it will have a better score than any of its neighbours).

Given the starting structure, A, we first calculate the score $S(A)$, which can be done using Algorithm 8. Algorithm 8 takes inputs $\mathrm{Pa}(X)$ and X and calculates both the maximum S_t score for X (given its parent set) and (one of) the corresponding optimal CPT:

$$\max_{\mathrm{CPT}(X) \in \mathcal{T}(\mathrm{Pa}(X), X)} \{S_t(\mathrm{CPT}(X))\},$$
$$\mathrm{argmax}_{\mathrm{CPT}(X) \in \mathcal{T}(\mathrm{Pa}(X), X)} \{S_t(\mathrm{CPT}(X))\}.$$

For ease of notation, we will simplify the above terms to $MaxS_t(X|\mathrm{Pa}(X))$ and $OptCPT(X|\mathrm{Pa}(X))$. Details of Algorithm 8 are given in Appendix D.2. If we apply Algorithm 8 with inputs $\mathrm{Pa}(X)$ (in structure A) and X for each $X \in V$, then $S(A)$ is the product of the returned $MaxS_t$ scores. This procedure will also return the optimal CPTs for structure A. Note that we record the individual $MaxS_t$ scores as well as $S(A)$ in order to calculate Δ values (defined below).

Now that we know the score of A, we want to evaluate the scores of all neighbours of A. For $X, Y \in V$, let the ordered pair (X, Y) denote the edge $X \to Y$. When we refer to changing an edge, e, in A, we mean adding the edge if $e \notin A$ and removing it if $e \in A$. We denote the resulting structure by $A \oplus e$. Every $B \in \mathrm{Ne}(A)$ can be written as $B = A \oplus e$ for some $e = (X, Y)$, where $X \neq Y$. We cannot have $X = Y$ as this would make e a loop edge, which cannot be present in A or B as both are acyclic. Thus, instead of calculating the scores of each $B \in \mathrm{Ne}(A)$, we evaluate how changing edge e affects the score of A, for each $e = (X, Y)$,

Algorithm 4: CP-Net Learning

```
Input   : $A$ – Starting structure
          $D$ – User choice data
          $\alpha$ – Change threshold
Output: $N$ – Learned CP-net

Initialise CP-net $N$ with structure $A$ and empty CPTs;
TableScores – Empty list of $S_t$ scores;
$S(A) = 1$;
// Calculate $S(A)$ and identify the optimal CPTs for $A$:
for $X_i \in V$ do
    Calculate $MaxS_t(X_i|\mathrm{Pa}(X_i))$;                  // Using Alg. 8
    Determine $OptCPT(X_i|\mathrm{Pa}(X_i))$;                  // Using Alg. 8
    TableScores$[i] = MaxS_t(X_i|\mathrm{Pa}(X_i))$;
    $S(A) = S(A) \cdot MaxS_t(X_i|\mathrm{Pa}(X_i))$;
    $\mathrm{CPT}(X_i) = OptCPT(X_i|\mathrm{Pa}(X_i))$;
end
$C$ – Cycles matrix for structure $A$;
Initialise $\Gamma$ and $P$ as empty $|V| \times |V|$ matrices;
// Calculate all $\Delta(e)$ values and determine the corresponding
   optimal CPTs:
for $X_i, X_j \in V$, $i \neq j$ do
    $e = (X_i, X_j)$;
    if $e \notin A$ then
        Calculate $MaxS_t(X_j|\mathrm{Pa}(X_j) \cup \{X_i\})$;          // Using Alg. 8
        Determine $OptCPT(X_j|\mathrm{Pa}(X_j) \cup \{X_i\})$;        // Using Alg. 8
        $\Delta(e) = MaxS_t(X_j|\mathrm{Pa}(X_j) \cup \{X_i\})/$TableScores$[j]$;
        $P_{i,j} = OptCPT(X_j|\mathrm{Pa}(X_j) \cup \{X_i\})$;
        $\Gamma_{i,j} = \Delta(e)$;
    end
    else
        Calculate $MaxS_t(X_j|\mathrm{Pa}(X_j) \setminus \{X_i\})$;          // Using Alg. 8
        Determine $OptCPT(X_j|\mathrm{Pa}(X_j) \setminus \{X_i\})$;        // Using Alg. 8
        $\Delta(e) = MaxS_t(X_j|\mathrm{Pa}(X_j) \setminus \{X_i\})/$TableScores$[j]$;
        $P_{i,j} = OptCPT(X_j|\mathrm{Pa}(X_j) \setminus \{X_i\})$;
        $\Gamma_{i,j} = \Delta(e)$;
    end
end
// Continued on the next page
// Continuation of Algorithm 4
ValidChanges= $\left\{ e = (X_i, X_j) \middle| C_{i,j} = 0 \wedge \left( \left( e \in A \wedge \Gamma_{i,j} > \frac{1}{1+\alpha} \right) \vee \left( e \notin A \wedge \Gamma_{i,j} > 1 + \alpha \right) \right) \right\}$;
while $|ValidChanges| > 0$ do
    // Identify the best valid edge change and update the
       structure, scores, and CPTs:
    $e^* = (X_{i^*}, X_{j^*}) = \text{argmax}_{(X_i, X_j) \in \text{ValidChanges}}(\Gamma_{i,j})$;
    $A = A \oplus e^*$;
    $\text{CPT}(X_{j^*}) = P_{i^*,j^*}$;
    $\Delta(e^*) = \Gamma_{i^*,j^*}$;
    $S(A) = S(A) \cdot \Delta(e^*)$;
    $\textbf{TableScores}[j^*] = \textbf{TableScores}[j^*] \cdot \Delta(e^*)$;
    Update $C$ according to the new structure of $A$;      // Using Alg. 9
    // Update the Δ values and corresponding optimal CPTs:
    for $X_k \in V,\ k \neq j$ do
        $e = (X_k, X_j)$;
        if $e \notin A$ then
            Calculate $MaxS_t(X_j|\text{Pa}(X_j) \cup \{X_k\})$;      // Using Alg. 8
            Determine $OptCPT(X_j|\text{Pa}(X_j) \cup \{X_k\})$;     // Using Alg. 8
            $\Delta(e) = MaxS_t(X_j|\text{Pa}(X_j) \cup \{X_k\}) / \textbf{TableScores}[j]$;
            $P_{k,j} = OptCPT(X_j|\text{Pa}(X_j) \cup \{X_k\})$;
            $\Gamma_{k,j} = \Delta(e)$;
        end
        else
            Calculate $MaxS_t(X_j|\text{Pa}(X_j) \backslash \{X_k\})$;      // Using Alg. 8
            Determine $OptCPT(X_j|\text{Pa}(X_j) \backslash \{X_k\})$;     // Using Alg. 8
            $\Delta(e) = MaxS_t(X_j|\text{Pa}(X_j) \backslash \{X_k\}) / \textbf{TableScores}[j]$;
            $P_{k,j} = OptCPT(X_j|\text{Pa}(X_j) \backslash \{X_k\})$;
            $\Gamma_{k,j} = \Delta(e)$;
        end
    end
    ValidChanges= $\left\{ e = (X_i, X_j) \middle| C_{i,j} = 0 \wedge \left( \left( e \in A \wedge \Gamma_{i,j} > \frac{1}{1+\alpha} \right) \vee \left( e \notin A \wedge \Gamma_{i,j} > 1 + \alpha \right) \right) \right\}$;
end
return $N$;
```

where $X \neq Y$. We record the multiplicative change in score and refer to this value as $\Delta(e)$:

$$\Delta(e) = \frac{S(A \oplus e)}{S(A)}.$$

If changing the edge e introduces cycles into the structure, then $A \oplus e \notin \mathcal{G}$. As this structure is not in our search space, this is not an edge change we need to consider. We use a *cycles matrix*, C, to record whether each edge, e, creates cycles when changed in A. Let us enumerate the variables such that $V = \{X_1, X_2, ..., X_n\}$.

$$C_{i,j} = \begin{cases} 1 & \text{if changing the edge } (X_i, X_j) \text{ creates cycles,} \\ 0 & \text{if changing the edge } (X_i, X_j) \text{ does not create cycles.} \end{cases}$$

This information is stored so that it can be updated (rather than recalculated, for efficiency) once changes to the structure are made. Details on how to calculate and update C are given in Appendix D.3.

If $e = X \rightarrow Y$, then Y is the only variable that has a different parent set in A and $A \oplus e$. Thus, all variables other than Y have the same maximum S_t score in both A and $A \oplus e$. This simplifies $\Delta(e)$ as follows, by the definition of the score given in Equation 4.9. Let $U = \text{Pa}(Y)$ in A. Note that, if $e \notin A$, then $\text{Pa}(Y) = U \cup \{X\}$ in $A \oplus e$. If $e \in A$, then $\text{Pa}(Y) = U \backslash \{X\}$ in $A \oplus e$.

$$\Delta(e) = \begin{cases} \frac{MaxS_t(Y|U \cup \{X\})}{MaxS_t(Y|U)} & \text{if } e \notin A, \\ \\ \frac{MaxS_t(Y|U \backslash \{X\})}{MaxS_t(Y|U)} & \text{if } e \in A. \end{cases}$$

Thus, to calculate $\Delta(e)$ and $S(A \oplus e)$, we only need to calculate the maximum S_t score for Y in $A \oplus e$. This can be done by one application of Algorithm 8. This also yields the optimal CPT(Y) for $A \oplus e$. As no other variable incurs a change of parents, their optimal CPTs remain unchanged from A.

Our learning algorithm goes through each possible edge change and calculates the associated Δ value and new optimal CPT. We record the Δ values and possible future CPTs in the *change matrix*, Γ, and *potential CPTs matrix*, P, respectively. That is, $\Gamma_{i,j} = \Delta(X_i \rightarrow X_j)$ and $P_{i,j}$ contains the optimal CPT(X_j) for $A \oplus \ (X_i, X_j)$.

Our aim is to maximise the structure score, so one might expect that we are only interested in edge changes with $\Delta > 1$ (that is, the changes that improve this score). However, if $\Delta > 1$ is enough to implement an edge change, then we are allowing edge changes that result in arbitrarily small improvements to the score. This may lead to adding edges due to noise in the data or estimation variability, rather than because of true improvements in the model fit. Further, requiring $\Delta > 1$ for edge removal may leave unnecessary edges in the structure, as we show below. Thus, we will require that edges improve the score by a sufficient margin in order to be present in the structure.

Definition 4.10. Let α be the *proportional change threshold* parameter. Then, in order to be in the learned structure, we require an edge to improve the score proportionally by at least α. Suppose we have structure A and we are considering changing the edge $e \notin A$ (that is, adding e to the structure), then this edge change is only *valid* if the following inequality holds:

$$\Delta(e) > 1 + \alpha. \tag{4.10}$$

As a consequence of this requirement, if $e \in A$, the removal of this edge is *valid* if it is not improving the score proportionally by α. That is, if $S(A) < (1+\alpha)S(A \oplus e)$. Thus, this edge change (removing e) is valid if the following inequality holds:

$$\Delta(e) > \frac{1}{1+\alpha}. \tag{4.11}$$

The change threshold α is a hyperparameter, which we set experimentally in §4.4.

As $\alpha > 0$, the bound given in Equation 4.11, for validity of edge removal, is actually less than 1. This ensures that any degenerate parents in the learned structure will be considered valid removals.

Proposition 4.11. *Let N be an acyclic CP-net over variables V that has optimal CPTs. Let $Pa(X) = U \cup \{Y\}$, where $X, Y \in V$, $U \subseteq V$, and $Y \notin U$. Suppose that Y is a degenerate parent of X according to CPT(X). Let $e = Y \to X$, then e is a valid edge for removal. That is,*

$$\Delta(e) > \frac{1}{1+\alpha},$$

where α is our proportional change threshold. In fact, $\Delta(e) \geq 1$.

Proof. By our assumptions

$$\text{CPT}(X) = \mathit{OptCPT}(X|U \cup \{Y\}),$$
$$S_t(\text{CPT}(X)) = \mathit{MaxS}_t(X|U \cup \{Y\}).$$

Let CPT_2 be the CPT(X) we obtain by removing Y as a parent (as we did in §3.2.1). By Proposition 4.6, we have that

$$S_t(CPT_2) \geq S_t(\text{CPT}(X)) = \mathit{MaxS}_t(X|U \cup \{Y\}).$$

As e is an edge for removal, we have

$$\Delta(e) = \frac{MaxS_t(X|U)}{MaxS_t(X|U \cup \{Y\})}.$$

As CPT_2 was formed by removing Y as a parent, this is a CPT for X with parents U. That is, $CPT_2 \in \mathfrak{T}(U, X)$. This implies that

$$\begin{aligned} MaxS_t(X|U) &= \max_{\mathrm{CPT}(X) \in \mathfrak{T}(U,X)}\{S_t(\mathrm{CPT}(X))\} \\ &\geq S_t(CPT_2) \\ &\geq MaxS_t(X|U \cup \{Y\}). \end{aligned}$$

Thus,

$$\Delta(e) = \frac{MaxS_t(X|U)}{MaxS_t(X|U \cup \{Y\})} \geq 1 > \frac{1}{1+\alpha}.$$

□

Corollary 4.12. *If Algorithm 4 returns the CP-net N, then N will have no degenerate parents.*

Proof. Suppose Algorithm 4 returns the CP-net N. By Proposition 4.11, if N has degenerate parent $Y \rightarrow X$, then this edge is valid for removal according to its Δ value. The removal of an edge cannot create cycles. Thus, removing e is a valid edge change. This is a contradiction as Algorithm 4 cannot terminate while valid edge changes remain. □

By Definition 4.10, removing edges is easier than adding them. Thus, our learning method is more likely to return sparser structures and is unlikely to result in overfitting (although these properties will depend on the chosen α parameter). Furthermore, by Corollary 4.12, the learned structure will have no degenerate parents (that is, the learned CP-net will have the simplest possible structure). Therefore, the returned CP-nets are likely to be easy to interpret and reason with.

Removing all degenerate parents simplifies the final learned structure and also increases the set of valid edge changes that the algorithm can consider. Suppose we remove an edge e from structure A. There may be some edge $e' \notin A$ such that adding e' to A results in cycles, but adding e' to $A \oplus e$ does not. Changing edge e' would not be considered from A, but it would be considered from $A \oplus e$. Removing degenerate parents makes the CP-net structure as simple as possible without changing the CP-net. By having the simplest structure, we maximise the number of edges that can be added without creating cycles and, thus, maximise the number of valid edge changes that the algorithm can consider before terminating.

Returning to our description of the learning method, we now have the score and CPTs for the starting structure, as well as the Δ values for all possible edge changes, as well as the corresponding CPT changes. Out of the edge changes that do not introduce cycles (that is, the neighbours of A in $\mathcal{G}$), the edges that satisfy the relevant bound from Definition 4.10 are valid. These are the structural changes we consider implementing. The remaining edge changes are considered invalid. These edges are not considered (even if they do not create cycles) as either they do not sufficiently improve the score or their removal would significantly decrease the score.

Out of the valid changes, we select the edge change with the largest Δ value – that is, the change that improves the score most – as we are performing a greedy search optimisation of the score. If more than one valid edge change has maximum Δ, then we choose one such edge change at random. The selected edge is then changed in the structure. The CP-net score can be updated as follows:

$$S(A \oplus e) = S(A) \cdot \Delta(e).$$

If $e = (X, Y)$, then we update the CPT(Y) from A to the optimal CPT(Y) for $A \oplus\ e$, which is stored in the relevant entry of P. The $MaxS_t$ value for Y can be updated similarly to the overall score. Let $\mathrm{Pa}_A(Y)$ be the parent set of Y in A and let $\mathrm{Pa}_{A\oplus e}(Y)$ be the parent set of Y in $A \oplus e$, then

$$MaxS_t(Y|\mathrm{Pa}_{A\oplus e}(Y)) = MaxS_t(Y|\mathrm{Pa}_A(Y)) \cdot \Delta(e).$$

Now that the structure has changed, C will need updating. The details of how to update C can be found in Appendix D.3.

For all variables $Z \in V$, $Z \neq Y$, the parent set of Z has not changed. Thus, for any edge $W \to Z$, the Δ value and associated CPT(Z) in P have not changed. Therefore, the Δ values and associated CPTs only need updating for edges of the form $W \to Y$. For each $W \in V$ such that $W \neq Y$, let $e' = W \to Y$. We can use Algorithm 8 to calculate the maximum S_t value for Y in $(A \oplus e) \oplus e'$ and the associated optimal CPT(Y). We then use these values, and $MaxS_t(Y|\mathrm{Pa}_{A\oplus e}(Y))$ (calculated above), to recalculate $\Delta(e')$ and update Γ. We also update the optimal CPT in P.

Now that all of the scores, CPTs, and matrices have been appropriately updated to the new structure, $A \oplus e$, we can assess whether there are any more valid changes. If so, the best valid change is implemented and the update procedure is repeated. This continues until there are no more valid changes. The CP-net N is then returned.

As our starting structure is acyclic and any edge changes that introduce cycles are invalid, the learned CP-net must also be acyclic. Furthermore, by the terminating condition, the learned CP-net must be locally optimal. That is, no acyclic neighbour of N has a significantly (according to change threshold α) greater score and every edge in N significantly contributes to the score.

Proposition 4.13. *Algorithm 4 will always terminate in finite time.*

Proof. The space over which Algorithm 4 searches is $\mathcal{G}$, the space of DAGs over nodes V. As $\mathcal{G}$ is finite, if we can prove that Algorithm 4 never considers the same structure in $\mathcal{G}$ twice, then the algorithm must terminate in finite time.

Suppose for the sake of contradiction that Algorithm 4 considers some structure $A \in \mathcal{G}$ twice. This means that at some point the algorithm considered A, then made a sequence of (valid) edge changes $(e_1, e_2, ..., e_m)$ that returned the same structure A. That is,

$$A = (\cdots(((A \oplus e_1) \oplus e_2) \oplus e_3) \cdots) \oplus e_m.$$

As the sequence of edge changes starts and ends at A, the number of edge removals and additions must be equal and m must be even. Let $m = 2k$ and assume, possibly after some reordering, that $e_1, e_2, ..., e_k$ are the edges that were added and $e_{k+1}, e_{k+2}, ..., e_{2k}$ are the edges that were removed. If we have a structure B_1 and $B_2 = B_1 \oplus e$, then by our definitions we have $S(B_2) = S(B_1) \cdot \Delta(e)$. Thus, we must have

$$S(A) = S(A)\Delta(e_1)\Delta(e_2)\cdots\Delta(e_m), \tag{4.12}$$

where each Δ is defined as appropriate for the context in which the edge change was implemented. This means that $e_i = e_j$ does not imply $\Delta(e_i) = \Delta(e_j)$ here, even if both edges were added/removed.

As only valid changes are made in our algorithm, we must have $\Delta(e_i) > 1 + \alpha$ for the edges that were added $(i = 1, ..., k)$ and $\Delta(e_i) > \frac{1}{1+\alpha}$ for the edges that were removed $(i = k + 1, k + 2..., 2k)$. Thus,

$$\begin{aligned}\prod_{i=1}^{m} \Delta(e_i) &= \prod_{i=1}^{k} \Delta(e_i)\Delta(e_{k+i}) \\ &> \prod_{i=1}^{k} (1 + \alpha)\frac{1}{1 + \alpha} \\ &= 1.\end{aligned}$$

Thus, Equation 4.12 cannot hold, contradicting our assumption. Hence, the algorithm will always terminate in finite time. □

Thus, Algorithm 4 always terminates in finite time and returns a locally optimal (as defined above) CP-net.

Algorithm 8 has complexity $O\Big(2^{|\mathrm{Pa}(X)|}2^nN + 2^{|\mathrm{Pa}(X)|}N^2\Big)$. The worst case scenario is $|\mathrm{Pa}(X)| = n-1$, in which case this complexity becomes $O(4^nN + 2^nN^2)$. The complexity of calculating the cycles matrix for a structure and the complexity of updating it after an edge is changed are given in Appendix D.3. From these results, we can calculate the complexity of Algorithm 4. Let E_1 be number of edges added by Algorithm 4 and let E_2 be the number of edges removed. Let $C = 4^nN + 2^nN^2$. Then the complexity of Algorithm 4 is $O(n^2C + E_1(n^3 + nC) + E_2(n^4+nC))$. This expression can be explained by the fact that Algorithm 4 must call an instance of Algorithm 8 for every possible edge when calculating the initial Δ values. Then, after each edge change, the cycles matrix must be updated and, for some variable X, every edge terminating at X must have its Δ value updated, which requires running Algorithm 8.

This complexity shows that our learning method is not theoretically tractable. In §4.4, we also provide an experimental evaluation of the efficiency of Algorithm 4. In §4.5, we discuss how our learning method might be made more efficient.

4.3.4 Random Starts Learning Variation

Our algorithm returns a CP-net that is locally optimal. That is, a CP-net that cannot be significantly improved upon by moving to a neighbouring CP-net. However, as we are using greedy search, there is no guarantee that this CP-net is globally optimal. In this section, we propose using multiple random starts in order to improve the score of the learned CP-net. This variation is also tested in our §4.4 experiments.

When utilising random starts, instead of running the learning algorithm once, we run it k times, each time using a randomised (acyclic) starting structure. Then, out of the k learned structures, we return the CP-net that has the greatest score. In our experiments, we use the empty structure as the starting structure when using Algorithm 4 only once. When using random starts, we used one empty starting structure and $k-1$ randomised starting structures. This allows us to determine whether using an empty or random starting structure performs better when there is no prior information. Using random starts simply multiplies the complexity of the learning method by k.

When using random starts, it may also be interesting to see which structural properties the k learned CP-nets agree upon. Such properties may be considered as more likely to be true. For example, if all k assign Y as a parent of X, we might consider this relation more certain than if it only happened in one of the learned structures.

The random starts variation must perform as well as, or better than, Algorithm 4 on its own, with respect to maximising the structure score. However, using random starts also increases learning complexity. In our experimental results section, we shall consider whether the benefit to performance outweighs the additional computational cost.

Another variation we may consider to improve our optimisation procedure is incorporating random walks into our learning algorithm. This variation is discussed further in §4.5, along with other alterations that may improve learning performance or efficiency.

4.4 Learning Performance Experiments

In this section, we provide an experimental evaluation of our learning method. We start by giving the details of the experiments and the performance measurements we use. Then we provide the experimental results and analysis.

4.4.1 Experiments

As we discussed in §4.3.4, we are testing two variations of our learning method:

1. A single application of Algorithm 4, using the empty graph as the starting structure.

2. Using Algorithm 4 with k random starts. We apply Algorithm 4 once using the empty graph as the starting structure. We then apply Algorithm 4 $k-1$ times using randomly generated acyclic starting structures. Out of the k learned CP-nets, the one with the highest score is returned as the learned CP-net. Any ties are broken at random.

Algorithm 4 calls Algorithm 8 in order to estimate the optimal CPT scores via Monte Carlo methods. This estimation uses a sample from the posterior Dirichlet distribution in Equation 4.4. As we mention in Appendix D.2, the same sample is used each time Algorithm 8 is called from Algorithm 4. This sample remains valid as our posterior distribution does not change over the course of learning. As we are using Monte Carlo estimation, the score estimates are dependent upon this sample. By re-using the same sample, we avoid the possibility of the algorithm moving in circles whilst appearing to consistently improve the score; if different samples are used at each estimation point, then it is possible for the algorithm to improve the score by returning to a CP-net it has visited previously, as two samples may return different score estimates for the same CP-net. It is possible that such loops may result in our learning method not terminating.

In the case of our second learning variation, each random start calls Algorithm 4 separately. Different samples are used for each of the k learning attempts. This was done for practicality, as the k learning attempts are performed independently of one another. Despite the fact that the scores of the k learned CP-nets come from different samples, they are still comparable as our estimation error is fairly low (see Appendix D.1). Thus, the CP-net with the largest estimated score will have the highest, or close to the highest, true score out of the k attempts.

Data Generation

In order to test both learning variations, we used simulated data that is consistent with a CP-net. In order to obtain such data, we first generated a random CP-net. We used outcome ranks (see Chapter 2) to find a consistent ordering for this CP-net. This gives a complete (not necessarily strict) preference ordering over the outcomes that is consistent with the generated CP-net. Technically, any consistent ordering is equally valid for ordering the outcomes here. For example, we could use the lexicographic consistent ordering defined by Boutilier et al. (2004a). Rank ordering (unlike lexicographic ordering) is not necessarily strict, meaning that outcomes can have equal preference. Such indifference is likely in a real person's preference order over many outcomes. Thus, rank order could be considered a more realistic choice in this regard.

Now that we have a preference ordering, we must specify the exact p_i values in order to generate our data. Technically the p_i can be any values in $[0, 1]$ that sum to 1, as long as they are consistent with the preference order. As we do not have real-world data, we cannot say what a typical distribution of the p_i values is like. However, if two p_i values are very close, it is likely that the difference in preference will be harder to detect from the data. For these reasons, we chose to make the p_i values increase in equal increments. This means that the values are evenly spaced and also as far apart as possible. These p_i values are obtained via the following procedure.

The outcome which is least preferred according to the preference ordering is assigned a weight of 1. The outcome(s) in the second least preferred position, according to the preference ordering, is (are) assigned a weight of 2. As we continue to move up the preference ordering to the most preferred outcome, the weights increase by 1 for each preference position. The p_i are then proportional to these weights. They can be calculated by dividing the outcome weights by the sum of all weights.

Example 4.14. Suppose we have six outcomes, $o_1, o_2, ..., o_6$. Below we denote a possible preference ordering and the outcome weights assigned by the above method. We also give the corresponding p_i values, which are obtained by dividing the weights by the total sum of weights, 16.

$$\begin{array}{rccccccccccc} \text{Preference order:} & o_1 & \succ & o_2 & \sim & o_3 & \sim & o_4 & \succ & o_5 & \succ & o_6 \\ \text{Weights:} & 4 & & 3 & & 3 & & 3 & & 2 & & 1 \\ p_i \text{ values:} & \frac{1}{4} & & \frac{3}{16} & & \frac{3}{16} & & \frac{3}{16} & & \frac{1}{8} & & \frac{1}{16} \end{array}$$

The outcomes with weight i are in the i^{th} least preferred position. We call this 'preference position i'. The p_i values defined above have the property that any outcome in preference position i is i/j times more likely to be chosen than an outcome in preference position j $(i \geq j)$. The weight of the worst outcome could instead be less than 1, which would increase the distance between consecutive p_i values. However, this would put the worst outcome at a disproportionate disadvantage in comparison to the other outcomes.

Now that we have the p_i values, we can generate our data by drawing a random sample from the multinomial distribution given in Equation 4.3.

Random CP-Net and Structure Generation

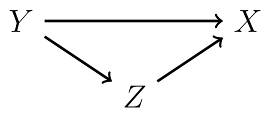
As mentioned in the previous section, we have chosen to use data that is consistent with a CP-net in these experiments. Intuitively, we are assuming our simulated data to be chosen by a user whose preferences can be modelled by a CP-net (or at least, whose preferences are consistent with a CP-net). Thus, as we explained above, in order to generate our simulated data, we first need a CP-net. These CP-nets are randomly generated by the same CP-net generator used in the experiments of Chapters 2 and 3 (see Appendix C.1 for details).

In our second learning variation, we consider learning from randomised starting structures as well as empty structures. Thus, we need to be able to randomly generate acyclic starting structures. We do this via the following procedure. All acyclic structures have a topological ordering of the variables. If there is an edge $X \to Y$ in the structure, then X must come before Y in this ordering. Thus, the adjacency matrix of an acyclic structure is upper triangular (possibly after some re-labelling of variables). We start by randomly generating an upper triangular adjacency matrix, A. This is done by randomly assigning each $A_{i,j}$ to be 0 or 1, for all $j > i$ (all other entries are 0). We then order the variables randomly, say $X_1, ..., X_n$. Then, by assuming that the adjacency matrix A corresponds to this variable order, we have generated an acyclic graph over our variable set. In practice, we re-arrange A according to this random order so that it corresponds to our standard enumeration of the variables.

Example 4.15. Suppose we have three variables, $\{X, Y, Z\}$. We first want to create a random, 3×3 upper triangular matrix, A. To do so we randomly allocate the three top right entries (those above the diagonal) to be 0 or 1. One possibility (where they are all 1) is given below:

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

We then order the variables randomly, say $\{Y, Z, X\}$. If we consider A to be the adjacency matrix with respect to this ordering, it gives us the following acyclic structure.



However, if all our working uses the natural ordering $\{X, Y, Z\}$, it is inconvenient to have A using a different ordering. Instead, we re-arrange the rows and columns of A so that it uses our usual ordering and is, thus, compatible with all other working. The re-ordered matrix is denoted A'.

$$A' = \begin{array}{c c} & \begin{array}{ccc} X & Y & Z \end{array} \\ \begin{array}{c} X \\ Y \\ Z \end{array} & \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \end{array}$$

Experiment Design

In this section, we give the details of the experiments we have carried out in order to evaluate the performance of our learning methods. These details are summarised in Tables 4.1 and 4.2.

We test our learning methods on CP-nets with 5 or 10 variables. In each case, we randomly generate 100 CP-nets and generate a simulated data sample of size 1000 for each. We provide our learning methods with increasing amounts of data, in order to see how much data is required for good performance. These increasingly large data sets are nested; for example, we learn from the first 50 data points, D_1, then from the first 100, D_2, so $D_2 \supsetneq D_1$. However, the way in which we select the training data depends on the performance measure we intend to use.

We use two types of performance measure to evaluate how well our learning methods perform, the details of which can be found in §4.4.2. The first type measures the agreement between the CP-net used to generate the data (the true CP-net), N_T, and the learned CP-net, N_L. These can be evaluated from N_L and N_T alone. The second type measures the agreement between N_L and unseen (future) user choice data. This requires a test set of data in addition to the training data we used to learn N_L.

If a performance measure evaluates similarity between N_T and N_L, we supply the algorithm with successively larger, nested subsets of the data. This is in order to see how much data we need to observe for our learning method to obtain a reasonably accurate CP-net model. To get a subset of size m, we simply use the first m data points in our sample. We use subsets of sizes 50, 100, 200, 300,..., 1000.

If we are measuring the agreement between N_L and a test set of unseen user choices, then performance depends upon the choice of test data used and so we implement cross validation. However, the size of the test data also impacts these performance scores. Thus, for effective comparison, the size of the test data set must remain constant as the training data size grows. We fix the size of all test sets to be 250. In order to evaluate the average agreement between the learned CP-net and a test set of 250 unseen user choices, we perform 5-fold cross validation. That is, we will randomly partition the data into two pieces (training data and test data) five times. Each time, we learn a CP-net from the training data and then evaluate agreement with the test set (unseen user choices). Agreement is then averaged over the five different attempts (and test sets). However, as we must keep the test set size at 250, but we want to use varying amounts of training data in our experiment, we cannot simply partition the whole data set. As each test set must have size 250, the training data can be up to size 750. We test training data sizes 50, 100, 200,...,700, 750. In order to perform the cross validation for m training data points, we take the first $m+250$ data points from our observed data, call this set D. We then partition D randomly into two parts of size m and 250. These make up the training and test data sets respectively. We apply learning using the training data and evaluate the performance measure with respect to the returned N_L and test data set. We repeat this with five random partitions of D and average the performance results over all five runs. Note that the increasingly large training sets are still nested to a degree.

Experiment	Learning Variation	Cross Validation – Yes/No	Number of Variables, n	Number of Random Starts
1a	1	N	$\{5, 10\}$	1
1b	1	Y	$\{5, 10\}$	1
2a	2	N	$\{5, 10\}$	$\{10, 20, 30\}$
2b	2	Y	$\{5\}$	$\{10, 20, 30\}$

Table 4.1: Experiment Details 1

Experiment	Training Data Size	Change Threshold (%)
1a	$\{50, 100, 200, ..., 1000\}$	$\{0.01, 0.05, 0.1, 0.5, 1, 2, 3, 4, 5, 10\}$
1b	$\{50, 100, 200, ..., 700, 750\}$	$\{0.01, 0.05, 0.1, 0.5, 1, 2, 3, 4, 5, 10\}$
2a	$\{50, 100, 200, ..., 1000\}$	$\{0.005, 0.01, 0.05\}$
2b	$\{50, 100, 200, ..., 700, 750\}$	$\{0.005, 0.01, 0.05\}$

Table 4.2: Experiment Details 2

As the training data is different depending on the performance measures, we perform two versions of each experiment; one which uses the cross validation method and one which does not. We can then evaluate each performance measure on the results of the appropriate version. This gives us four experiments, which we shall enumerate 1a, 1b, 2a, and 2b. Experiment 1a will test learning variation 1, without cross validation. Experiment 1b will test learning variation 1, using cross validation. Similarly for experiments 2a and 2b.

Note that, when using random starts and cross validation (experiment 2b), the order is as follows; we perform k random starts and pick the highest scoring out of the k learned CP-nets. The relevant performance measures are then evaluated for this CP-net with respect to the test set. This is repeated five times and the performance scores are averaged over the five CP-nets, which were each the best out of their k random starts.

In our experiments, we also vary the change threshold hyperparameter, α, in order to find which value optimises performance. The number of random starts used in experiment 2 is also varied. This is in order to identify the optimal number of random starts by considering the tradeoff between the complexity cost of additional random starts and the benefit to learning performance. Tables 4.1 and 4.2 give the full details of which parameter values are tested in each experiment.

Experiments 1a and 1b use a large range of α values, as there is no intuitive boundary between a real score improvement and the effects of overfitting. As we see in §4.4.3, these experiments show that learning performs better as α gets smaller. Thus, in the subsequent experiments, we use a smaller set of lower values for α.

Note that the same 100 CP-nets (of each size) and the same 1000 data points for each CP-net are used in all four experiments.

4.4.2 Performance Evaluation Measures

In this section, we refer to the CP-net used to generate the data as N_T and the learned CP-net as N_L.

In order to measure the complexity of our algorithm, we record the time it takes for our methods return N_L. For random starts, we record the time elapsed for each random start and use the sum to represent the total time elapsed. This omits the time taken to generate random starting structures and evaluate which returned structure scores highest. However, these tasks are fairly trivial and would collectively take only a few seconds. As we see in §4.4.3, the learning times are fairly large, so this omission will not significantly affect the results.

We also record how many edge changes each learning method performs. Combined with the theoretical complexities in §4.3.3, these results can give us an approximate complexity for our algorithm and its variations.

The score of each N_L is also recorded. Comparing these demonstrates how well our variations have improved the greedy search optimisation method.

In order to measure the similarity between N_T and N_L, we use the following two methods. These measures are evaluated only for the experiments which do not use cross validation (the 'a' experiments). The similarity between N_T and N_L illustrates whether our learning algorithm correctly identified the dependencies and preferences in the data that were implied by N_T, the true preference structure. Keep in mind, however, that this is not the whole story. Our aim was to learn a CP-net that was consistent with the user's true preference order – the p_i values. The p_i ordering is a linearisation of N_T, and so it generally contains more preferences than just those implied by N_T. Thus, while we want N_T and N_L to agree, we also want N_L to be consistent with the rest of the p_i order. It is possible for N_L to agree with most of the p_i ordering while differing from N_T.

First, we measure **preference graph (PG) similarity**. This evaluates how similar the preference graphs of N_T and N_L are. Let us denote these preference graphs by G_{N_T} and G_{N_L}. As N_T and N_L are CP-nets over the same variable set, they also have the same associated outcome set. Thus, their preference graphs use the same set of nodes. In any preference graph, two outcomes are connected by an edge if and only if they differ on the value of exactly one variable. Thus, any two outcomes, o_1 and o_2, are connected by an edge in G_{N_T} if and only if $\text{HD}(o_1, o_2) = 1$, which occurs if and only if o_1 and o_2, are connected by an edge in G_{N_L}. Therefore, if the orientation of the edges was removed, G_{N_T} and G_{N_L} would be the same graph. To measure the similarities between these graphs, we can simply measure the proportion of edges that are oriented the same way in both. Note that, as we work with binary CP-nets only, both G_{N_T} and G_{N_L} have $2^{n-1}n$ edges. The PG similarity of N_T and N_L is defined by the following metric:

$$\frac{|\{o_1 \to o_2 \in G_{N_L}\} \cap \{o_1 \to o_2 \in G_{N_T}\}|}{|\{o_1 \to o_2 \in G_{N_L}\}|} = \frac{|\{o_1 \to o_2 \in G_{N_L} | o_1 \to o_2 \in G_{N_T}\}|}{2^{n-1}n}.$$

By the above argument, this metric is symmetric – swapping N_T and N_L does not change the similarity score. Recall that CP-nets and their preference graphs are equivalent. Thus, PG similarity measures the proportion of all preferences encoded by N_L that agree with N_T. As they are swap preferences, all other preferences encoded by N_L must be contradicted by N_T. This measure has previously been used to evaluate the performance of some of the existing CP-net learning methods (Haqqani and Li, 2017; Liu et al., 2014, 2013).

Our second metric for measuring similarity between N_T and N_L is **entailment agreement/disagreement/incomparability**. To evaluate this, we first generate a set of distinct pairwise outcome preferences that are entailed by N_T, call this set $\mathcal{P}$. For every preference $p \in \mathcal{P}$, we then evaluate whether this preference, its reverse, or neither are entailed by N_L. The proportion of preferences in each (distinct) case are the entailment agreement, disagreement, and incomparability measures, respectively:

$$\text{Entailment Agreement} = \frac{|\{o_1 \succ o_2 \in \mathcal{P} | N_L \models o_1 \succ o_2\}|}{|\mathcal{P}|},$$

$$\text{Entailment Disagreement} = \frac{|\{o_1 \succ o_2 \in \mathcal{P} | N_L \models o_2 \succ o_1\}|}{|\mathcal{P}|},$$

$$\text{Entailment Incomparibility} = \frac{|\{o_1 \succ o_2 \in \mathcal{P} | N_L \nvDash o_1 \succ o_2 \wedge N_L \nvDash o_2 \succ o_1\}|}{|\mathcal{P}|}.$$

These metrics (or similar variations) are often used as a measure of CP-net learning performance (Allen, 2016; Allen et al., 2017b; Guerin et al., 2013; Haqqani and Li, 2017; Labernia et al., 2018, 2017; Liu et al., 2014, 2013; Michael and Papageorgiou, 2013; Siler, 2017), though Guerin et al. (2013) were the first to use them in this exact form. Some works also used these proportions to define their aim (or optimisation function) in learning (Allen, 2016; Allen et al., 2017b; Haqqani and Li, 2017; Labernia et al., 2017; Liu et al., 2014). Except for Siler (2017), these methods all use outcome preferences as training data and, in most cases, they used the training data as $\mathcal{P}$ when evaluating these proportions. This is clearly a biased set for testing learning performance. We will generate a test set of preferences, $\mathcal{P}$, that is unrelated to our training data.

In order to generate $\mathcal{P}$, we repeat the following procedure until the set is sufficiently large; first, generate a random outcome pair (o_1, o_2) and use dominance testing to determine whether either preference direction is entailed by N_T. If $N_T \vDash o_1 \succ o_2$ or $N_T \vDash o_2 \succ o_1$, then the entailed preference is added to $\mathcal{P}$. In order to set a size for $\mathcal{P}$, we must first determine how many distinct preferences a CP-net may entail.

Proposition 4.16. *Let N be a binary acyclic CP-net with n variables. Let E_N denote the number of distinct pairwise outcome preferences that are entailed by N. Let N_0 be the binary CP-net over n variables that has no edges in its structure. Then we must have*

$$E_N \geq E_{N_0} = 3^n - 2^n.$$

Proof. See Appendix E.11.

This gives us a tight lower bound on the number of preferences entailed by an acyclic binary CP-net over n variables (tight because this bound is achieved by the CP-net with no edges). For $n = 5$, this bound is 211 and, for $n = 10$, it is 58,025. This means that, for any CP-net with five variables, we can always construct a set of k distinct entailed preferences if $k \leq 211$. Similarly for CP-nets over ten variables. In our experiments, we use $|\mathcal{P}| = 211$. This set of entailments will be more representative of N_T in the $n = 5$ case than for $n = 10$. We can see from the entailments bound of 58,025 that, to achieve a similarly representative set for $n = 10$, we would need an impractically large set of preferences. Thus, we use the same $|\mathcal{P}|$ for both cases. When generating $\mathcal{P}$ and evaluating the entailment agreement/disagreement/incomparability proportions, we need to perform dominance testing. We use the rank and suffix fixing method of dominance testing that we introduced in Chapter 2.

Unlike PG similarity, these proportions illustrate the average agreement between N_L and N_T on general preferences, not just swap preferences. Further, we also consider the cases where N_L is consistent with N_T preferences, even if they are not entailed by N_L – this is the entailment incomparability case. Recall that we are aiming to find N_L that is weakly consistent with the p_i ordering. Thus, our aim is to have all N_T preferences be consistent with N_L, if not entailed. Note that every swap preference must be entailed or contradicted (its reverse is entailed), thus, we do not need to consider consistency separate to entailment in the PG similarity.

As the preferences in $\mathcal{P}$ are all entailed by N_T, this metric is not symmetric like PG similarity; if we swap N_T and N_L, these proportions may change.

As most of the existing methods for CP-net learning use outcome preferences as data, there are no existing metrics for measuring agreement between N_L and (unseen) user choice data. Thus, we define the following two metrics. We show that these metrics can also be used to evaluate the agreement between N_L and the true p_i ordering. In general, they can evaluate the agreement between any CP-net and (quantitative) preference ordering.

These new metrics can be evaluated from N_L and a test set of choice data. This enables us to evaluate learning performance without knowing the true CP-net, as required by the previous measures. This is important for real world applications, as we are unlikely to always know what the user's true CP-net is, if one even exists (which we do not assume in our learning).

The first metric is called **data flip agreement (DFA)**. This metric evaluates the sum of the data differences over each swap preference entailed by N_L. This is also the sum of all data differences over the edges of the preference graph, G_{N_L}. This sum is scaled by the size of the data and $|V|$, so that it lies on the $[-1, 1]$ scale. Suppose we want to evaluate the agreement between N_L and the choice data set D. Let $d(o)$ denote the number of times outcome o is chosen in D. We define DFA as follows:

$$\text{DFA} = \frac{\sum\limits_{(o,o')\in F(N_L)} d(o) - d(o')}{|V| \sum\limits_{o\in\Omega} d(o)} = \frac{\sum\limits_{o'\to o\in G_{N_L}} d(o) - d(o')}{|V| \sum\limits_{o\in\Omega} d(o)},$$

where $F(N_L) = \{(o, o') \in \Omega \times \Omega | N_L \vDash o \succ o' \wedge \text{HD}(o, o') = 1\}$.

If DFA is large, then the swap preferences, $o \succ o'$, entailed by N_L are strongly supported by the data. That is, $d(o) - d(o')$ is large, meaning that o is picked a lot more often than o'. We can interpret the value of DFA as follows; recall that CPT(X) consists of rules of the form $\mathbf{u} : x_1 \succ x_2$, for each $\mathbf{u} \in \text{Dom}(\text{Pa}(X))$. Any outcome, o, such that $o[\text{Pa}(X)] = \mathbf{u}$, is '$X$-preferred' if $o[X] = x_1$. Otherwise (if $o[X] = x_2$), o is 'X-undesirable'. These are distinct and exhaustive states, every outcome is exactly one of X-preferred and X-undesirable. Thus, every user choice is either X-preferred or X-undesirable. If a CPT is supported, we would expect there to be more X-preferred choices than X-undesirable choices. DFA is the average difference between the proportion of X-preferred choices and X-undesirable choices. Say DFA$= 0.2$, then, on average, for every variable, 60% of user-choices were preferred and the other 40% were undesirable. As we assume there is noise in the data (that is, the user does not always pick the optimal outcome), we do not expect to obtain DFA$= 1$. However, if DFA is larger, then the disparity between X-preferred and X-undesirable choices is greater, which suggests that CPT(X) is more strongly supported by the user's true preference order.

Despite this interpretation, it is still not clear what constitutes a 'good' DFA value, other than DFA> 0. Thus, we will use this measure mostly for comparison purposes.

We evaluate DFA for N_L with a test data set, in order to see how well N_L agrees with unseen (future) user choice data. To evaluate how well N_L agrees with the true preference order (p_i order), we use data that represents this ordering perfectly. In order to construct such a data set, we make $d(o)$ be the weight of o we used when calculating our p_i. Thus, the proportion of times o_i is chosen in this data set is exactly p_i. We call this *perfect data.*

Example 4.17. For the p_i values and weights in Example 4.14, the perfect data set must have

$$d(o_1) = 4,\ d(o_2) = 3,\ d(o_3) = 3,\ d(o_4) = 3,\ d(o_5) = 2,\ d(o_6) = 1.$$

Thus, the perfect data set is

$$\{o_1, o_1, o_1, o_1, o_2, o_2, o_2, o_3, o_3, o_3, o_4, o_4, o_4, o_5, o_5, o_6\}.$$

Our second measure of agreement between N_L and unseen choice data is **data order consistency (DOC)**. This metric estimates the proportion of a given ordering that is weakly consistent with N_L. Recall that our aim is to find N_L that is weakly consistent with the user's true preference order. In real world applications, we are unlikely to know the user's true preference order. However, user choice data gives us an ordering over the outcomes (according to how often they were chosen) that approximates user preference. We use DOC to estimate the proportion of this ordering that is weakly consistent with N_L. However, in our experiments, the true preference order is known. Thus, we will also evaluate DOC with respect to this ordering (this is done by using the perfect data instead of a set of new user choice data).

To evaluate DOC for N_L with a given data set, D, we perform the following procedure. Let $d(o)$ denote the number of times o is chosen in D – these values induce the data ordering we are considering. Randomly generate a set of distinct, unordered pairs, $\{o_1, o_2\}$, where $o_1 \neq o_2$. Evaluate $d(o_1)$ and $d(o_2)$ and determine whether N_L is consistent with the implied preference. If $d(o_1) > d(o_2)$, this suggests $o_1 \succ o_2$, which is consistent with (not contradicted by) N_L if $N_L \nvDash o_2 \succ o_1$. Similarly if $d(o_2) > d(o_1)$. If $d(o_1) = d(o_2)$, then this implies $o_1 \sim o_2$ (the user is indifferent), which is consistent with N_L if $N_L \vDash o_1 \bowtie o_2$. We return the proportion outcome pairs that are consistent with N_L. Note that, unlike DFA, DOC also considers the agreement between N_L and the data on non-swap preferences. Let $\mathcal{Q}$ denote the set of randomly generated unordered pairs. Then

$$\mathrm{DOC} = \frac{\left|\left\{\{o_1, o_2\} \in \mathcal{Q} \middle| \begin{array}{l} d(o_1) > d(o_2) \wedge N_L \nvDash o_2 \succ o_1 \vee \\ d(o_1) = d(o_2) \wedge N_L \vDash o_1 \bowtie o_2 \end{array}\right\}\right|}{|\mathcal{Q}|}. \tag{4.13}$$

As $\mathcal{Q}$ can be any set of unordered pairs, we must have $|\mathcal{Q}| \leq 2^{n-1}(2^n - 1)$. If $\mathcal{Q}$ is the set of all ordered pairs, $|\mathcal{Q}| = 2^{n-1}(2^n - 1)$, then DOC is the exact proportion of the ordering that is consistent with N_L, otherwise it is an approximation. For $n = 5$, we use the set of all ordered pairs, $|\mathcal{Q}| = 496$. For $n = 10$, this is impractical as $2^{n-1}(2^n - 1) = 523,776$. Thus, for $n = 10$ we use $|\mathcal{Q}| = 1000$.

Note that, if the entirety of a sub-ordering of the preference order is weakly consistent, then the sub-ordering is strongly consistent. That is, there is a consistent ordering of N_L that contains this sub-ordering.

4.4.3 Results

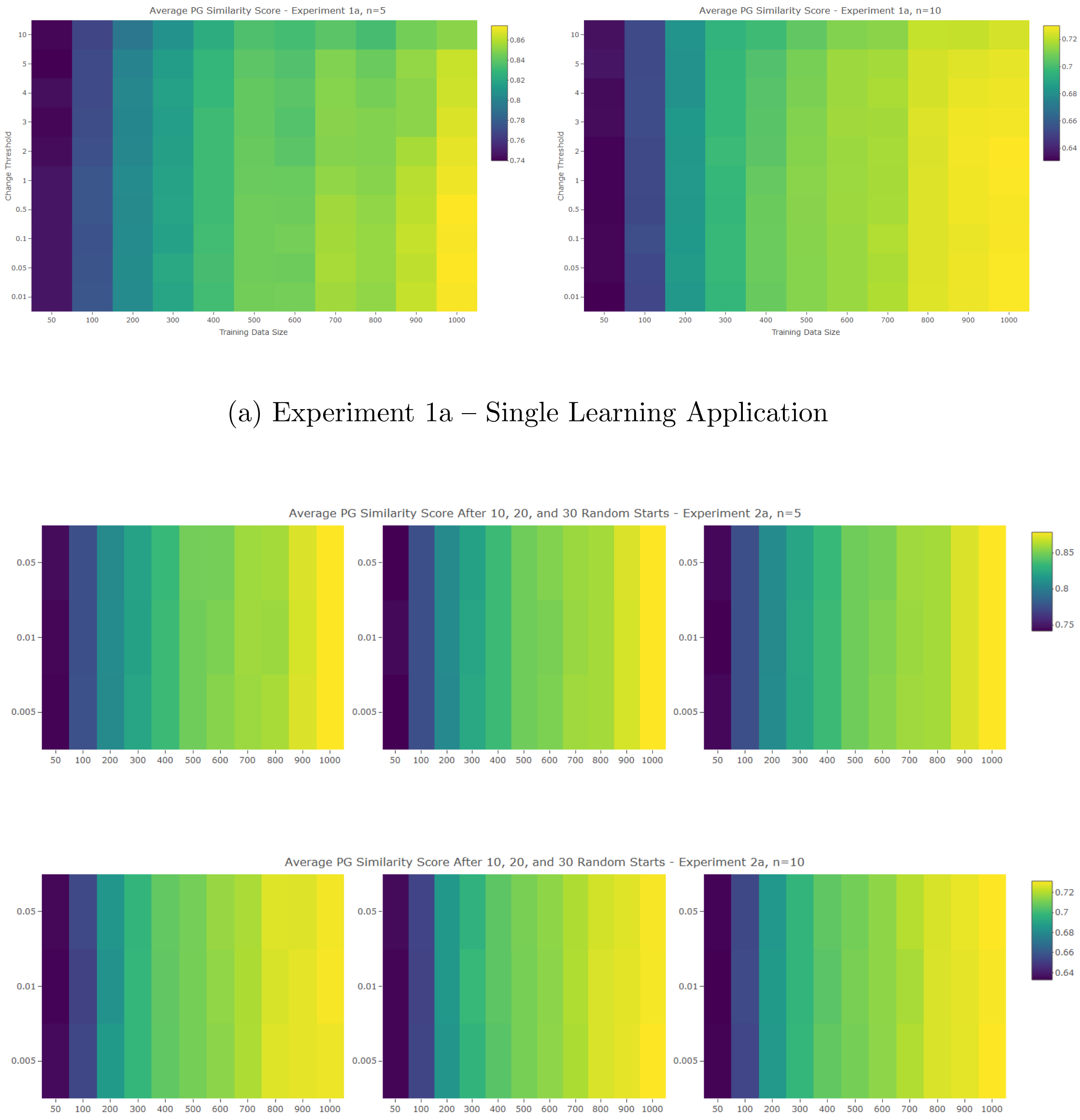
In this section, we analyse the results of the experiments described above. These results are summarised via heatmaps in Figures 4.1 – 4.7. These heatmaps show the average results for each combination of change threshold, α, and training data size that we tested (as detailed in Tables 4.1 and 4.2). Each data point is averaged over the 100 simulated CP-nets tested and, in the cross validation case (the 'b' experiments), over the five training (and test) sets. For the experiment 2 results, we have separated the results into the three cases of 10, 20, and 30 random starts. However, all three heatmaps use the same scale, making these results directly comparable and allowing us to evaluate which number of random starts is optimal.

Figure 4.1 shows the preference graph (PG) similarity scores for the experiments without cross validation (the 'a' experiments), separated by the number of variables, n. Recall that PG similarity is a measure on the $[0, 1]$ scale, showing how similar the learned CP-net, N_L, is to the true CP-net, N_T. A similarity score of 1 means that the learned CP-net is N_T. All plots show a clear improvement as the training data size increases. That is, the PG similarity increases with the training set size. This is as we would expect, as more training data means that the learning algorithm has more informed beliefs regarding the user's preferences and so we would expect the learned CP-net to be more accurate and, thus, closer to the true CP-net.

In experiment 1a, we see that performance also improves as the change threshold, α, gets smaller (though perhaps for $n = 10$ performance is beginning to level off or worsen for the smallest α values). When setting the change threshold, we want α to be large enough to filter out any score improvements caused by noise in the data or score estimation error, but small enough that all real improvements are considered valid. Thus, one might expect learning performance to improve as α decreases but eventually worsen once α becomes too small and begins allowing edge changes due to noise or estimation error. This change-point would be the optimal choice of α. Perhaps this is what we are seeing in the $n = 10$ data. However, one would expect the $n = 10$ change-point to be lower than $n = 5$ as more variables means larger possible parent sets and, thus, the relative improvement of adding or removing a single edge (parent) may be smaller. Thus, it is also possible that this levelling off for $n = 10$ is simply due to variation in the learning process due to score estimation (since the smaller values of α are fairly close together and, thus, may only marginally affect learning performance).

In experiment 2, we restricted our range of α values for practicality. As learning appeared to perform best for smaller values of α in experiment 1, we chose the smallest values of α as well as an additional, smaller value, $\alpha = 0.005\%$, in order to see whether performance would continue to improve as α decreases.

In experiment 1a, we obtained a maximum similarity score of 87.5% in the $n = 5$ case and 73.0% in the $n = 10$ case. For $n = 5$, this is fairly close to the original CP-net. We expect $n = 10$ learning to be at a disadvantage here as 10 variable CP-nets have 1,024 outcomes. Thus, even 1000 choice data points is nowhere near enough to reflect the full preference ordering over the outcomes. Hence, all learning attempts for $n = 10$ are using very approximate preference



(a) Experiment 1a – Single Learning Application

(b) Experiment 2a – Learning With Random Starts

Figure 4.1: Preference Graph Similarity With N_T (No Cross Validation)
Yellow – Higher degree of similarity
Blue – Lower degree of similarity

data. In light of this, a similarity score of 73% is quite impressive. Moving onto experiment 2a, we do not see much improvement from the addition of random starts. For experiment 2a, the $n = 5$ case has maximum similarity score of 87.9% and the $n = 10$ case has maximum similarity score of 73.1%. As these random starts require running the learning algorithm 10, 20, or 30 times instead of one, the improvement with respect to PG similarity does not seem worth the additional complexity cost. Furthermore, using additional random starts (moving from 10 to 20 to 30) shows little to no improvement in PG similarity. The change threshold does not appear to significantly affect performance in experiment 2a, though this could be because the range of values is too small to have much effect.

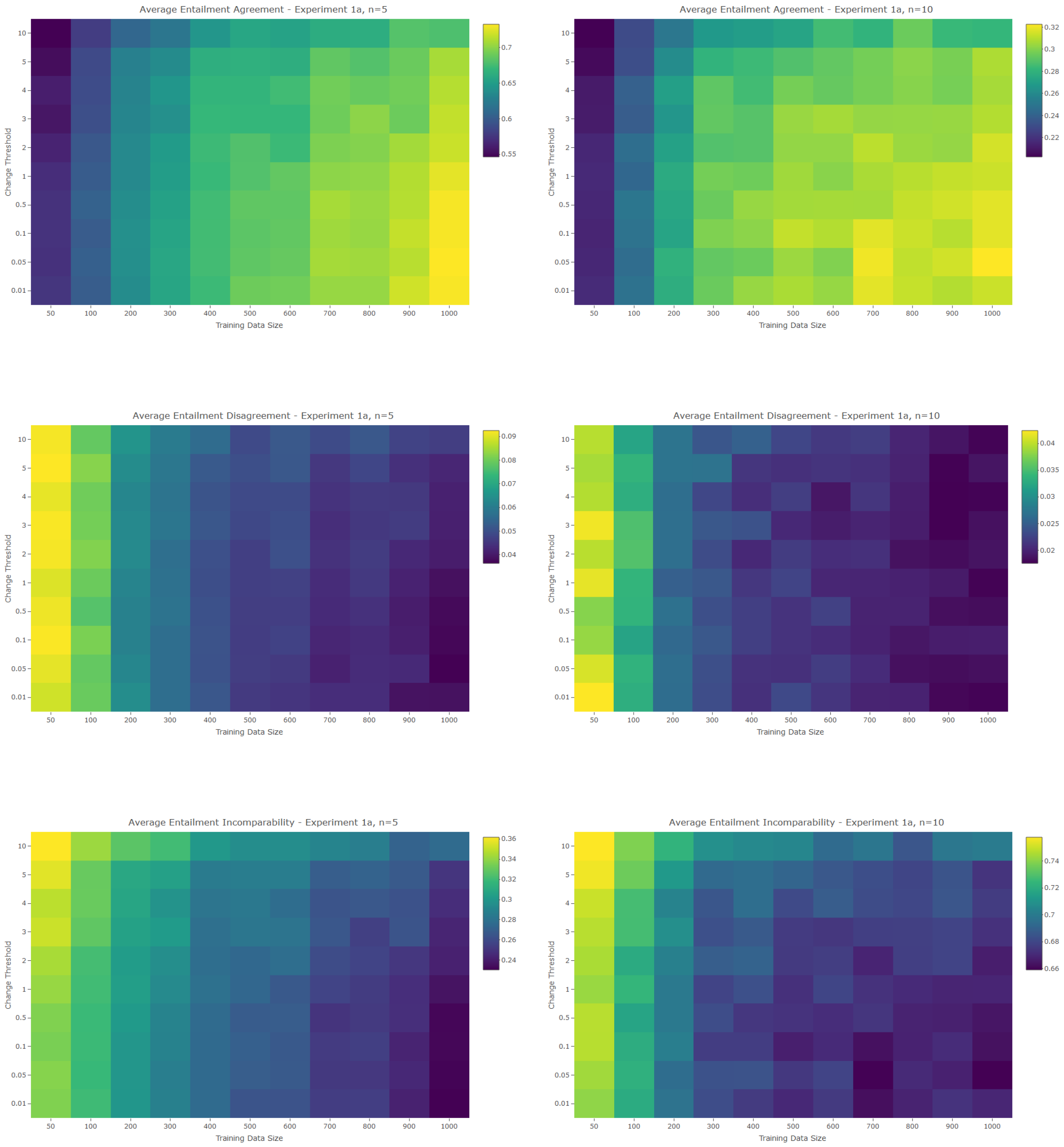
The standard errors of these results are fairly small, ranging between approximately $0.70 - 1.00\%$ for experiment 1a, $n = 5$, $0.72 - 1.00\%$ for experiment 2a, $n = 5$, $0.48 - 0.70\%$ for experiment 1a, $n = 10$, and $0.47 - 0.67\%$ for experiment 2a, $n = 10$. In general, these standard errors get smaller as the training data size increases. In experiment 1a, variability also decreases for smaller values of α. In experiment 2a, there is no clear effect of α (possibly because the tested values are all fairly close), nor any effect of the number of random starts on standard error.

Figure 4.1 has shown that we achieved a high level of similarity with the original CP-net (with respect to swap preferences) by using a single application of learning with smaller values of α. The most likely method of improving performance from these results is to use more training data, particularly in the $n = 10$ case. We might also consider more α values in the range $(0\%, 0.01\%]$ to see how performance behaves (if it continues to improve) and identify the optimal change threshold.

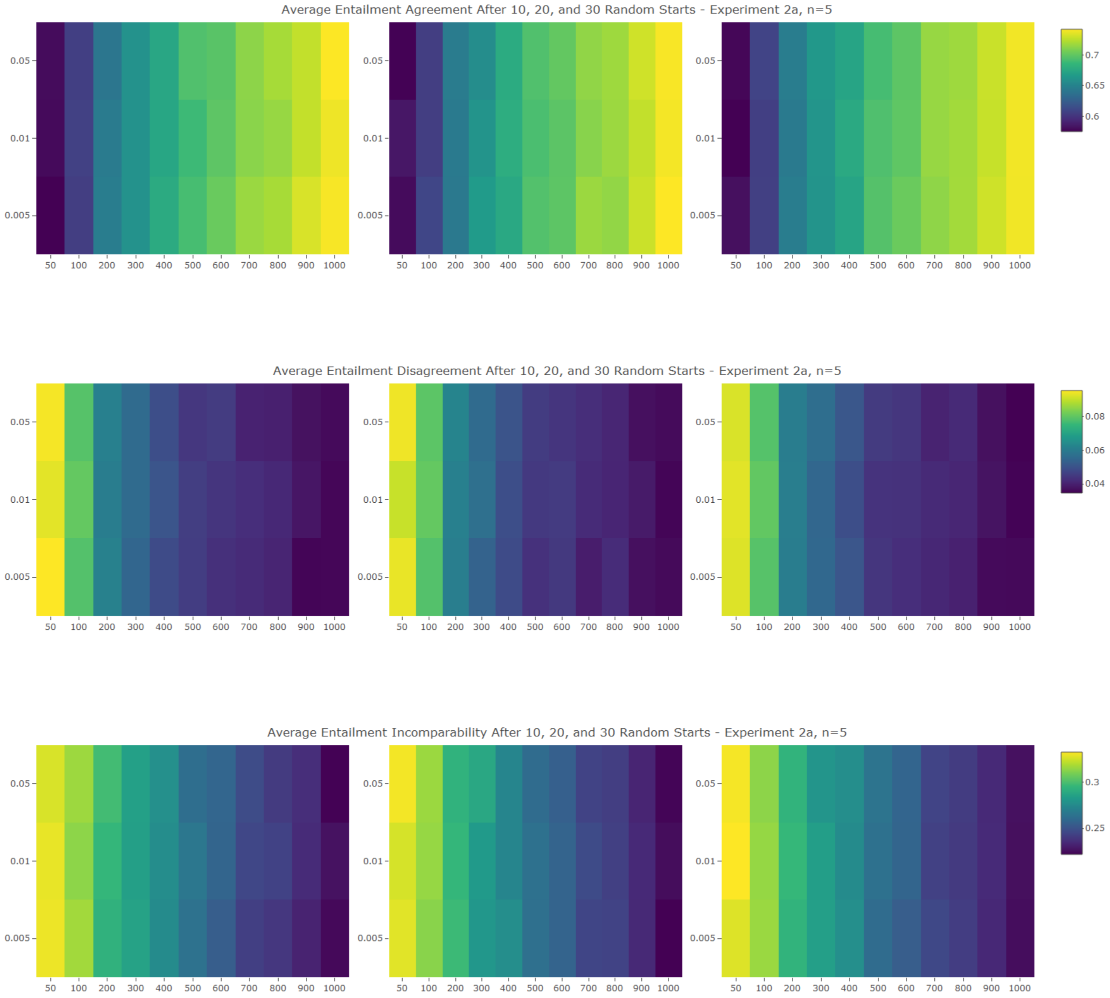
Figure 4.2 shows the entailment agreement, disagreement, and incomparability results for the experiments without cross validation (the ‘a’ experiments). Recall that these proportions show the agreement between N_L and N_T regarding general (not necessarily swap) preferences entailed by N_T. Our aim is to learn a CP-net that is consistent with the user’s true preference order, which is a consistent ordering for N_T. Thus, for entailed preferences of N_T, it is sufficient to have N_L be consistent with (not contradictory to) them. That is, our primary interest is that the entailment disagreement scores are low.

In both experiments 1a and 2a, we again see the general trend of improved performance as the amount of training data increases, as we would expect. The proportion of preferences entailed by N_L increases, whereas the proportion of preferences that are contradicted or incomparable for N_L both decrease. Thus, the level of agreement between N_L and the preferences encoded by N_T is increasing.

For experiment 1a, performance also appears to improve as the change threshold, α, decreases, though there is more variation in this trend for $n = 10$. This is



(a) Experiment 1a – Single Learning Application

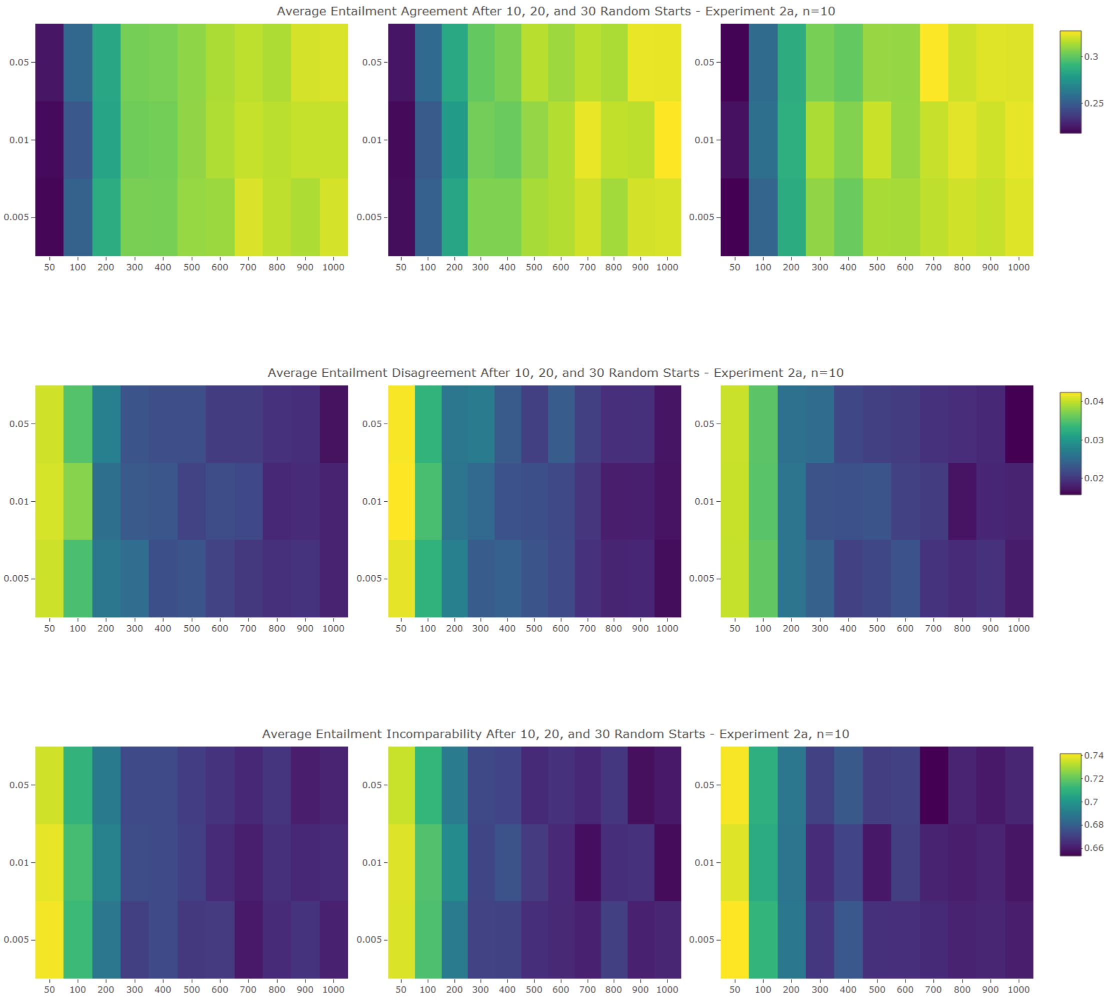


Average Entailment Agreement After 10, 20, and 30 Random Starts - Experiment 2a, n=5

Average Entailment Disagreement After 10, 20, and 30 Random Starts - Experiment 2a, n=5

Average Entailment Incomparability After 10, 20, and 30 Random Starts - Experiment 2a, n=5

(b) Experiment 2a, $n = 5$ – Learning With Random Starts



Average Entailment Agreement After 10, 20, and 30 Random Starts - Experiment 2a, n=10

Average Entailment Disagreement After 10, 20, and 30 Random Starts - Experiment 2a, n=10

Average Entailment Incomparability After 10, 20, and 30 Random Starts - Experiment 2a, n=10

(c) Experiment 2a, $n = 10$ – Learning With Random Starts

Figure 4.2: Proportions of Entailment Agreement, Disagreement, and Incomparability With N_T Entailed Preferences (No Cross Validation)
Yellow – Larger proportions
Blue – Smaller proportions

perhaps because there is more variation in how well the data represents the preference order for $n = 10$ as even 1000 data points is insufficient to reflect the full ordering, as we discussed previously. This overall trend again suggests that the optimal α value is around 0.01% or lower. For $n = 5$, the optimal results were 73.2% entailed, 3.6% contradicted, and 23.0% incomparable. The optimal 3.6% contradicted means that we can achieve a learned CP-net consistent with 96.4% of preferences entailed by N_T. For $n = 10$, the optimal results were 32.2% entailed, 1.8% contradicted, and 66.0% incomparable. That is, we can achieve a learned CP-net consistent with 98.2% of N_T preferences. Thus, our learned CP-nets are highly compatible with N_T preferences, even though we do not appear to recover N_T exactly through learning. The $n = 5$ case has a higher rate of agreement than $n = 10$, which could be because the data is more informative about the true preference order or because larger CP-nets generally have a higher rate of incomparability. The latter could also explain the lower disagreement proportion for $n = 10$.

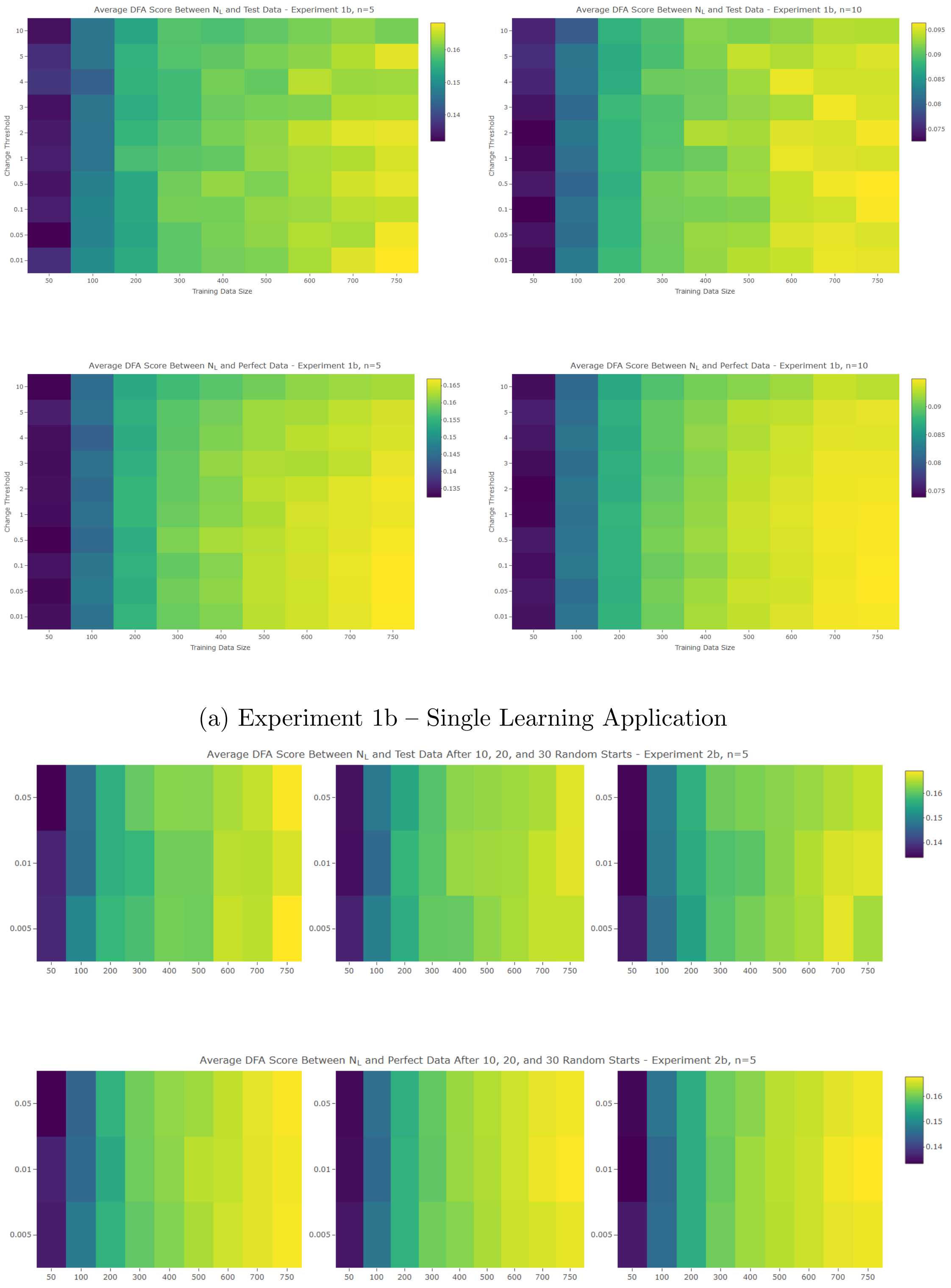
In experiment 2, a smaller, lower range of α values were tested as smaller values appeared to be the most successful in experiment 1. However, in experiment 2a we do not see any clear impact of the choice of α, perhaps because we tested such a small range of values. As in Figure 4.1, there appears to be little improvement in performance from adding random starts (that is, between the experiment 1a and 2a results). Improvement from random starts mostly occurs for smaller data sizes, where learning does not perform as well. Further, moving from 10 to 20 to 30 random starts has little effect on the results. In some cases, additional random starts has made the average results worse. The optimal scores for experiment 2a, $n = 5$, are 74.2% entailed, 3.4% contradicted and 22.2% incomparable. For $n = 10$, the optimal scores are 32.7% entailed, 1.6% contradicted and 65.3% incomparable. Note that not all of these optimal results were achieved by 30 random starts, some are achieved after only 20. These improvements are minimal and don't appear worth the extra computational cost of applying learning 10, 20, or 30 times. Furthermore, we see in later plots that non-empty starting structures appear to result in significantly longer learning times. This means that the computational cost of random starts is *more* than 10, 20, or 30 times a single application of learning from an empty start (as we had in experiment 1).

The standard errors of these results are also low. For $n = 5$, the standard errors are between approximately 1.20 – 1.62% for agreement, 0.22 – 0.50% for disagreement, and 0.96 – 1.30% for incomparability. For $n = 10$, the standard errors range between 0.74 – 1.09% for agreement, 0.09 – 0.27% for disagreement, and 0.69 – 1.05% for incomparability. For the disagreement proportions, variability is generally smaller for larger training set sizes. For $n = 10$, the agreement and incomparability standard errors increase for larger training sets. Otherwise, there is no clear effect of either training size or α on variability. The number of random starts also does not affect variability (in experiment 2a), though the experiment 2a standard errors are consistently slightly higher than experiment 1a.

These results again show that our learned CP-nets are highly compatible with the N_T preferences. The more training data and the smaller the change threshold, the better the learning performance. However, additional random starts do not appear to be worth applying. In future experiments, we would like to evaluate how performance improves with additional data (particularly in the $n = 10$ case) and for smaller values of α, as these results suggest reducing α improves performance – as mentioned above, however, we expect that at some point α will become too small and performance will worsen. The Figure 4.2 results are quite similar to the PG similarity results. This is unsurprising as they both measure the agreement between N_L and N_T on preferences entailed by N_T. The swap preferences considered by PG similarity are the building blocks of the general preferences considered here.

Figure 4.3 shows the average DFA scores for the cross validation learning experiments (the 'b' experiments). These results are averaged over the 100 tested CP-nets and the five training sets for each. For each combination (and for each choice of training data size and change threshold), we evaluated the DFA between the learned CP-net, N_L, and its corresponding test set as well as between N_L and the corresponding perfect data set. Recall that DFA measures the agreement between a data set and a CP-net, with respect to the swap preference. DFA scores lie in the range $[-1, 1]$, where a higher score means better agreement. However, as we mentioned previously, it is not clear what DFA score constitutes a 'good' level of support, other than the requirement that we have DFA> 0. Thus, we are largely considering these scores relative to one another.

However, we have also evaluated the DFA for the true CP-net, N_T, to put these scores in some context. The average DFA between N_T and the test sets for each experiment are as follows: in experiment 1b, N_T has average DFA of 0.1839 with the test data for $n = 5$ and 0.1060 for $n = 10$. In experiment 2b, N_T has average DFA of 0.1840 with the test data for $n = 5$ (no $n = 10$ experiment was performed in this case). We also evaluated the average DFA between N_T and the associated perfect data. As perfect data is CP-net specific and the same set of $n = 5$ CP-nets were used for both experiments, these averages are only dependent on n. The average DFA of N_T with perfect data is 0.1832 for $n = 5$ and 0.1051 for $n = 10$. It is unclear why the test data agreement scores are higher, other than it is perhaps easier to strongly agree with smaller data sets where it is likely that only the more



(a) Experiment 1b – Single Learning Application

(b) Experiment 2b – Learning With Random Starts

Figure 4.3: DFA Results for Test Data and Perfect Data (Cross Validation Experiments)
Yellow – Higher level of agreement
Blue – Lower level of agreement

strongly preferred outcomes appear (250 data points is insufficient to represent the full preference ordering for $n = 5$ or $n = 10$). A DFA score of approximately 0.18 (for $n = 5$) implies that, on average, for every variable, X, 59% of data points are X-preferred and 41% are X-undesirable, as explained in §4.4.2. A DFA score of approximately 0.11 (for $n = 10$) implies that, on average, 55.5% of data points are X-preferred and 44.5% are X-undesirable. Thus, the distribution of the data over improving flips is closer for $n = 10$.

All DFA plots show improved scores (improved agreement with both test and perfect data) as the training data size increases. This is as we would expect; as the training data increases, the learning algorithm has more informed beliefs about the user's true preference ordering and, thus, we would expect N_L to model the user's true preferences more accurately. Consequently, we expect the learned CP-net to agree more strongly with data that reflects the user's true preference order (perfect data) and data generated according to this ordering (test data). As we have seen in previous figures, performance also improves as α decreases in experiment 1b, with slightly more variation in this trend for $n = 10$. Whereas, in experiment 2b, there is no clear effect of change threshold on performance.

In experiment 1b, for $n = 5$, the optimal DFA between N_L and test data is 0.1683 and for perfect data it is 0.1670. For $n = 10$, the optimal DFA between N_L and test data is 0.0964 and for perfect data it is 0.0950. These scores are all reasonably close to the agreement scores for N_T. Thus, our learned CP-net agrees with unseen data and the true preference ordering (on swap preferences) almost as well as the true CP-net (which is consistent with the true preference ordering). Recall that our learning aim was to obtain a CP-net, N_L, consistent with the true preference order. These results are particularly good for $n = 10$, where the training data (of size up to 750) is less informative.

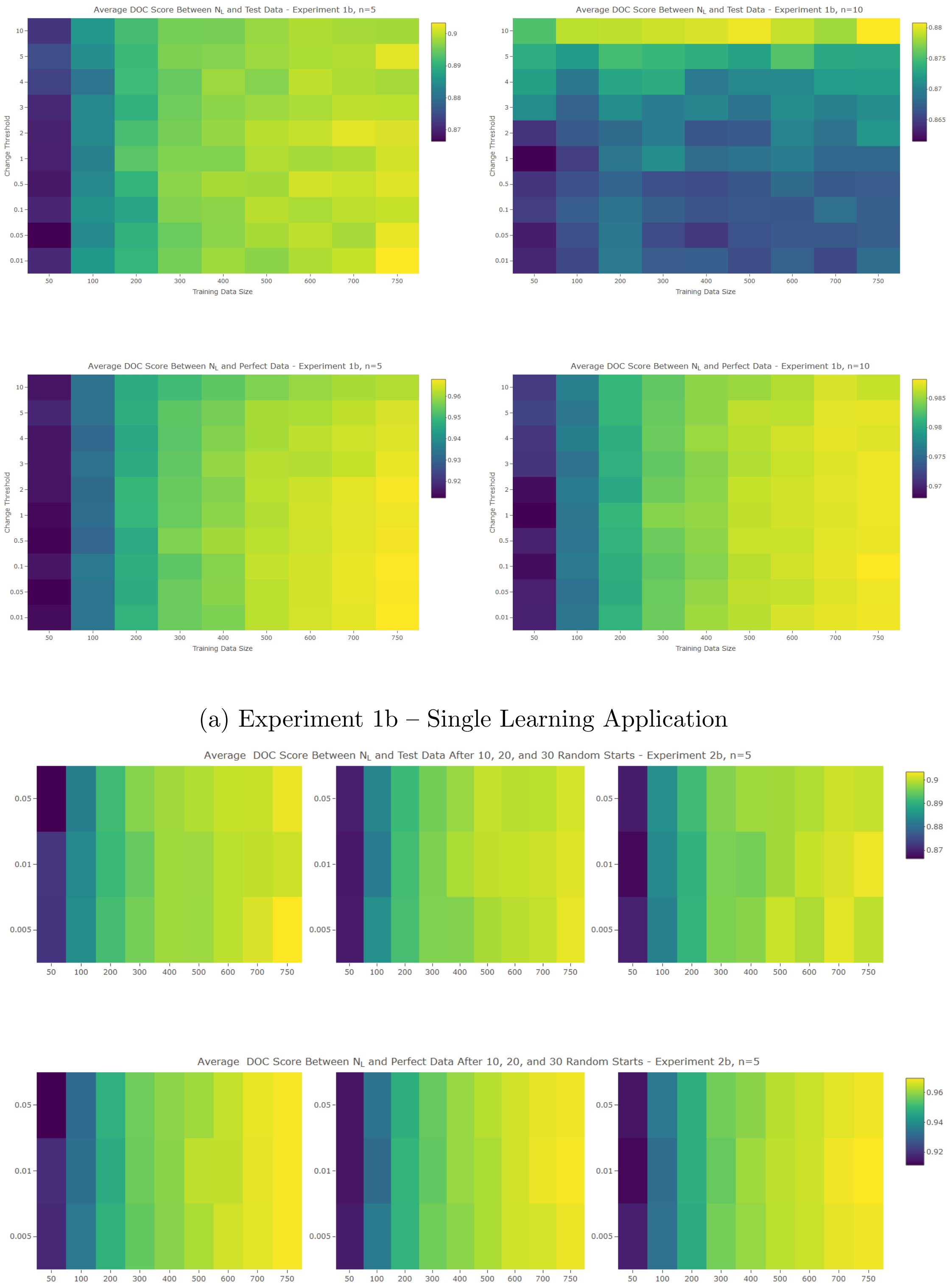
In experiment 2b ($n = 5$), the optimal DFA between N_L and test data is 0.1692 and for perfect data it is 0.1677. Note that the optimal test data DFA is actually achieved after only 10 random starts. In general, there is little improvement as a consequence of adding additional random starts (moving from 10, to 20, to 30). In fact, in some cases, average performance worsens. Thus, it does not appear worth using more than 10 random starts. Perhaps even fewer random starts are required to achieve this improvement in DFA score. In general, the experiment 2b results show that random starts do improve DFA to some degree – the optimal DFA results are higher and closer to the N_T DFA scores. However, as DFA values are difficult to interpret as 'good' or 'bad', it is unclear whether this improvement is worth the additional complexity cost.

All DFA standard errors are reasonably small. Note that these are the standard errors of the cross validation averages. That is, for each of the 100 tested CP-nets, we take the average DFA score over the five training and test sets considered (for each set of parameters tested) and then take the standard error of these 100 averaged scores. For $n = 5$, the standard errors of DFA with test data lie between approximately 0.0017 – 0.0028 for experiment 1b and between 0.0019 – 0.0030 for experiment 2b. For DFA with perfect data, they lie between 0.00090 – 0.0020 for experiment 1b and between 0.00089 – 0.0019 for experiment 2b. For $n = 10$ (experiment 1b), the standard errors for DFA with test data lie between 0.0012 – 0.0019 and for DFA with perfect data, they lie between 0.0006 – 0.0011. The variability of these scores decreases as the training data size increases. For experiment 1b, variability of DFA with perfect data also decreases for smaller values of α, but this is not the case for experiment 2b (possibly because the tested values of α in experiment 2b are all fairly similar). Otherwise, there is no clear effect of α, or the number of random starts (in experiment 2b), on the variability of the DFA scores.

From these results, we have seen that our learned CP-nets (for both $n = 5$ and 10) can achieve DFA scores close to the N_T scores (the optimal DFA) both with unseen data and the true preference order. This suggests that the learned CP-nets agree with (swap preferences of) the true preference order almost as well as N_T (which is to say 100%). Learning again performs best with more training data and smaller α values. Random starts do improve performance, though no more than 10 are required (possibly less) and it is not clear whether this improvement is worth the associated cost.

In order to interpret DFA values more accurately (not just relative to one another), we should evaluate DFA between the true CP-net and data for a range of CP-nets and data. Possibly also considering how the score changes with small alterations to the true CP-net. We have already seen that the optimal DFA score depends on the number of variables. Understanding what constitutes a good or optimal DFA score would be helpful in applications where the true CP-net is unknown (or does not exist). In such situations, we cannot compare our values to the optimal as we have done here.

Figure 4.4 shows the average DOC scores for the cross validation experiments (the 'b' experiments). For each learned CP-net in these experiments, we evaluated the DOC between N_L and the associated test data and perfect data. Recall that DOC gives the proportion of the data-induced ordering that is consistent with N_L. For perfect data, this ordering is the true preference order. For $n = 5$, DOC is the exact proportion, but for $n = 10$, it is an approximation. Recall that our aim was



(a) Experiment 1b – Single Learning Application

(b) Experiment 2b – Learning With Random Starts

Figure 4.4: DOC Results for Test Data and Perfect Data (Cross Validation Experiments)
Yellow – Higher level of consistency
Blue – Lower level of consistency

to learn a CP-net that is weakly consistent with the true preference ordering, so our primary aim is for the DOC between N_L and perfect data to be as close to 1 as possible.

For comparison, we evaluated the DOC scores for N_T, the true CP-net. The average DOC scores for N_T over all the test data sets used in each experiment are as follows. For experiment 1b, the average DOC between N_T and the test data is 0.9029 for $n = 5$ and 0.7179 for $n = 10$. For experiment 2b, the average DOC for $n = 5$ is 0.9030. As the true preference order is consistent with N_T, the DOC between N_T and perfect data is always 1. The DOC is not always 1 for test data as it consists of 250 choices randomly drawn according to the true outcome preferences. This data size is insufficient to reflect the true ordering over all outcomes for $n = 5$ or $n = 10$ (for $n = 10$, most outcomes will not appear at all). Further, outcomes with similar preferences (probabilities) may have reversed or equal preferences in the test set due to its size. These issues are likely to be worse in the $n = 10$ case, explaining the smaller average DOC score in this case.

For $n = 5$ we see a similar trend to the previous scores. For experiment 1b, the DOC scores improve as the training data size increases and α gets smaller. The optimal average scores are 90.34% consistency with test data and 96.80% consistency with the true preference order. Thus, our learned CP-nets agree with new data just as well as N_T and they are consistent with almost all of the true preference ordering. Furthermore, all training data sizes and change thresholds give fairly good average DOC scores, with all test DOC scores $> 86\%$ and all perfect data DOC scores $> 91\%$. We can obtain scores close to optimal for all data sizes ≥ 300 for test data and ≥ 500 for perfect data. Thus, our learning algorithm is very successful at achieving preference order consistency (our primary aim), even when approximate training data or overly harsh change thresholds are used. This is good because it means that learning should perform well even if there is insufficient data and it is not overly sensitive to the α parameter setting.

For experiment 2b ($n = 5$), performance improves as the amount of training data increases, as expected, but there is no clear effect of changing α or additional random starts (increasing from 10 to 20 or 30) in this experiment. In fact, in some cases, additional random starts result in worse average scores. Thus, we do not expect further random starts to improve performance. The optimal DOC scores for experiment 2b are 90.34% DOC between N_L and test data and 96.95% for perfect data. Thus, while utilising random starts does increase the average DOC score, it is a very minor improvement, not worth the additional complexity. Furthermore, while the optimal scores have improved, using random starts does not improve the average DOC for all choices of α and training data size, so random starts does not reliably improve DOC.

For $n = 10$ (experiment 1b), we see the usual patterns for DOC with the perfect data (our primary aim) but not with test data. For the perfect data, we again see that performance improves as training data increases and α gets smaller. The optimal DOC score is 98.82% consistency with the true ordering. In fact, all data size and α combinations give a good DOC score, with all scores $> 96\%$. We can obtain near-optimal DOC for any data size ≥ 500. Thus, we again find our algorithm to be very successful at obtaining high levels of consistency with the true ordering (our primary aim). In fact, these results are even better than the $n = 5$ case, despite the fact that the training data is more reflective of the true preference ordering in the $n = 5$ case. This is perhaps because larger CP-nets tend to have more cases of incomparability. If two outcomes are incomparable, then the CP-net is always consistent with the data with respect to these outcomes.

However, for the test data, the DOC scores for $n = 10$ appear to depend primarily on the value of α. While there is some improvement as the training data size increases, the performance drastically improves when $\alpha = 10\%$ is used. This is much larger than the other values of α tested, perhaps explaining the significant jump in DOC score. We have seen similar (though less extreme) jumps between $\alpha = 5\%$ and $\alpha = 10\%$ in other plots. This increase in DOC suggests that the consistency with test data improves as α gets larger. The values of α that are less than 1% are all fairly close, explaining why little change is visible between these values.

Using a larger threshold means that only edge changes that improve the score by a larger amount are implemented. That is, only those edges representing stronger preferences are implemented – those with large data (and, hence, probability) differences. Generally, we do not want to make α too large as it means that finer details of the preference order from the training data are ignored and, hence, the learned CP-net is less accurate as a model for user preference.

However, a test data set is 250 random choices (drawn according to user preference). As CP-nets with $n = 10$ have 1,024 outcomes, 250 data points cannot represent the whole ordering. In fact, most outcomes will not appear at all – meaning they all are of 'equal preference' in the data ordering. As the test data set is small, we would expect that it is mostly the more preferred outcomes that appear and even these may not be in the correct relative frequency. Thus, the test data induced ordering is likely to only include the stronger preferences – that the most preferred outcomes are preferred to the least preferred outcomes. A possible explanation for why DOC with the test data increases with α here is that, by using a higher change threshold, learning ensures consistency with the strong preferences (those present in the test data) but does not force the entailment of finer grained preferences. If the latter are then largely incomparable, they will be consistent with the test data, regardless of whether it considers them equally preferred or has the wrong ordering. On the other hand, all incorrect orderings or (incorrectly) equal preferences in the test data will be contradicted if N_L entails these finer preferences, as we expect for smaller α values. This is less likely to be an issue for $n = 5$ as 250 test data points will be more reflective of a preference ordering over only 32 outcomes.

The optimal DOC between N_L and test data for $n = 10$ is 88.08%. This is for $\alpha = 10\%$ and 750 training data points. However, in practice, we would utilise the smaller α values as we can see from the perfect data results that this is where we achieve optimal consistency with the true preference order. Also, as explained above, we believe that these high values of DOC are an artefact of the small test sets rather than better learning. When smaller α values are used, we can still get DOC scores up to around 86.8% as training size increases. Thus, we still get a high level of agreement with the test data when we learn the preference order more accurately. In fact, all α and data size combinations give a good DOC score, with all average values $> 86.1\%$. Notice that these scores are higher than the DOC between N_T and test data, which was 71.79%. This could be due to the same reasoning as above, because the training data is ≤ 750 data points which, again, is not sufficient to represent the whole preference ordering. Thus, even when low thresholds are used, the algorithm only attempts to enforce the preferences that occur in the training data. This will again be the ‘stronger’ preferences that are more likely to appear in the test set. On the other hand, N_T is consistent with the whole ordering, entailing much of it, including preferences unlikely to be reflected in such a small test set. This may also occur for $n = 5$, boosting the DOC scores for test data, but this will be to a smaller degree as, especially for 750 data points, the training data more accurately represents the full preference ordering.

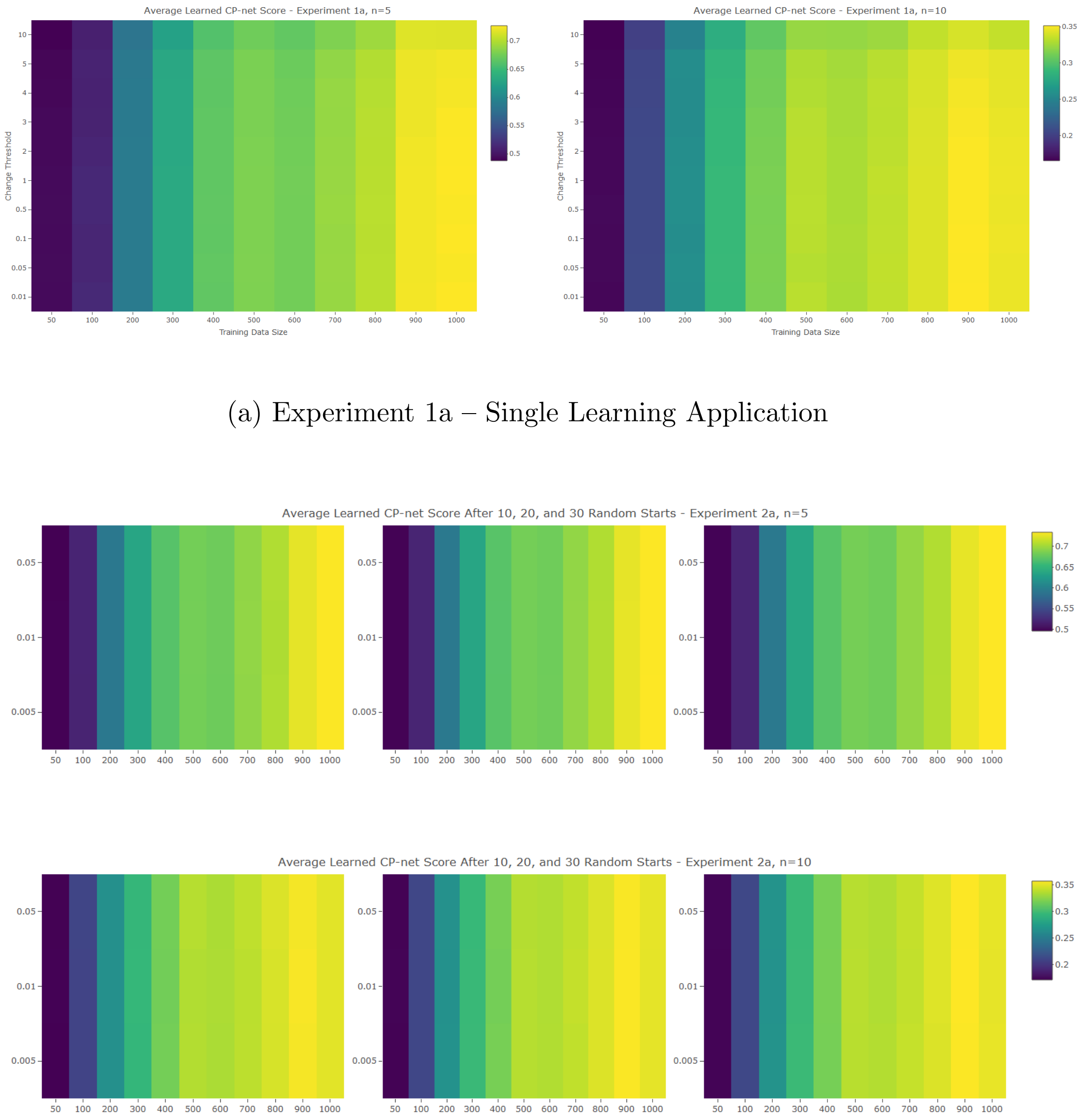
The standard errors of these DOC scores are all reasonably small. Note that these are, again, the standard errors of the cross validation averages. For $n =$ 5, the standard errors of DOC with test data (both experiments) lie between approximately 0.20 – 0.31% and for DOC with perfect data, they lie between 0.17 – 0.34%. For $n = 10$ (experiment 1b), the standard errors of DOC with test data lie between approximately 0.13 – 0.27% and for DOC with perfect data, they lie between 0.048 – 0.11%. In general, the variability of the DOC scores decreases for larger training data sizes and shows no clear dependence upon α or the number of random starts used (in experiment 2b). One exception to this is the DOC with test data results for $n = 10$, where variability is higher for larger training sets and also increases for smaller values of α.

From the Figure 4.4 results, we have seen that our learning is able to obtain high levels of consistency with the true preference ordering (which was our aim in learning), even for $n = 10$ where the training data is more approximate. This optimal consistency occurs for larger training sets and smaller change thresholds, agreeing with the performance results we have seen previously. However, all combinations gave fairly high DOC scores, showing that learning can perform similarly well with even less data. Adding random starts gives minimal improvement, which is not worth the additional complexity. In future, we would like to evaluate how performance behaves for smaller values of α, as these results suggest the scores will continue to improve.

We have also seen that learning can obtain high levels of consistency with previously unseen data. In fact, in this case, the learned CP-nets perform just as well or better than N_T. However, these results suggest that consistency with test data is not always best served by a completely accurate preference model (particularly for small scale test sets) and that learned CP-nets are at an advantage from being learned from the same type of data as the testing sets use. If one is using test data DOC to measure order consistency of N_L when the true ordering is unknown, then a test set large enough to reflect the full ordering should be used. If it is being used to evaluate predictive power on unseen data, then perhaps incomparability should not be considered a success in cases other than indifference as the true preference has not necessarily been correctly predicted – in practice, incomparable outcomes are likely to be ordered arbitrarily.

One might expect DOC to behave similarly to DFA and PG similarity as they are all measures of agreement between N_L and the test or perfect data sets. DFA and PG similarity consider flip agreements, which are the building blocks of the general orderings considered by DOC. In the perfect data case, DOC does show similar performance trends. The differences in behaviour (between DOC and DFA) in the test data cases are due to the above issues with 'weaker' preferences not being represented correctly in the small test set. For DOC, all orderings are consistent or not. However, in DFA, we look at weighted agreement. These 'weaker' preferences have either zero or little data difference in the test set and, thus, do not impact the DFA scores much.

Figure 4.5 shows the average score of the learned CP-nets for experiments 1a and 2a. The learned CP-net scores were also recorded for the 'b' experiments, averaging over the multiple cross validation iterations as well. These results were similar to those for the 'a' experiments. As the 'b' experiments are similar, but do



(a) Experiment 1a – Single Learning Application

(b) Experiment 2a – Learning With Random Starts

Figure 4.5: Learned CP-net Scores (No Cross Validation)
Yellow – Higher score
Blue – Lower score

not have $n = 10$ results for experiment 2 and use a smaller range of training data sizes, we have chosen to give the more comprehensive 'a' experiment results only here.

In general, edge changes performed by learning improve the CP-net score. The exception is when an edge is removed that improves the score but not sufficiently. The change threshold stops the learning process when the improvements to the score are getting 'too small'. A lower change threshold means that more edge additions (which improve the score) are possible and the threshold for edge removal increases (getting closer to 1), meaning fewer edge removals that worsen the score are possible. Thus, we expect the final score to be higher in general when a lower change threshold is used. Note that this improvement is not guaranteed. If we have the same starting structure and training set, it is possible that a lower change threshold can still give a worse score as the lower threshold can cause the search to go in a different direction.

In our results, we find that the CP-net scores generally increase as α gets smaller, as we expected. These score improvements are fairly small and, thus, do not show up well apart from the large jump between $\alpha = 10\%$ and $\alpha = 5\%$. More specifically, the range in CP-net scores (for a given training data size and ignoring $\alpha = 10\%$) is generally less than 0.004 for experiment 1a and less than 0.001 for experiment 2a. There is some variability in this trend of increasing scores. This could be caused by the variability in the effect of lowering α, as discussed above, or because of score estimation error as each learning attempt uses a distinct Dirichlet sample, or, in the case of experiment 2, variation due to the randomised starting structures. The increase in CP-net scores is particularly inconsistent for the smaller values of α in experiment 1a and in experiment 2a. This is likely to be because these values of α are all fairly close together, meaning that the associated score improvements are likely to be smaller and more easily outweighed by such variation.

The learned CP-net scores also increase with the size of the training data. This is perhaps because, as the data size increases, the learning algorithm has more information about the true preference order. Thus, with larger training sets, more of the learned CP-net preferences will have associated data and, thus have larger potential support (and we expect learning to maximise this support). When we have very little data, most of the preference rules associated with a CP-net will have little to no data and, thus, the support score for these rules (and thus the CPTs and the CP-net) will be low as the believed preferences are likely to be equal or very close. This would also explain why the $n = 10$ scores are lower than $n = 5$

as the training data will be missing a lot more information about the preference order for $n = 10$, due to the larger outcome set.

There are some exceptions to CP-net scores increasing with training data size. For both experiments, we see a dip at 600 training data points (more extreme in the $n = 5$ case) and for $n = 10$ we see another dip at 1000 data points. As these dips do not occur at 600 data points in the cross validation experiments (where different training data is used), they must be due to the exact training data sets. The cross validation experiments did not use training data of size greater than 750 but we suspect that the $n = 10$ dip at 1000 data points is also due to the specific training sets, rather than the size of the training data. Such dips could be caused by one or several of the size 600 (or 1000) training sets being more difficult to fit to a CP-net than the preceding size 500 (or 900) training set, causing a significant decrease in score and lowering the average. This could be because the additional 100 data points result in conflicting preferences that cannot be satisfied by one CP-net simultaneously, meaning that any learned CP-net must contradict the data to some degree. Alternatively, it could be that the additional 100 data points create preference equality between certain outcomes, lowering the potential support scores (small or no data differences cannot add much support to any preference). Note that, although these issues have caused a dip in average CP-net score, we do not see similar dips in the other heatmaps, meaning that the quality of N_L is not affected even though it is not as well supported by the training data.

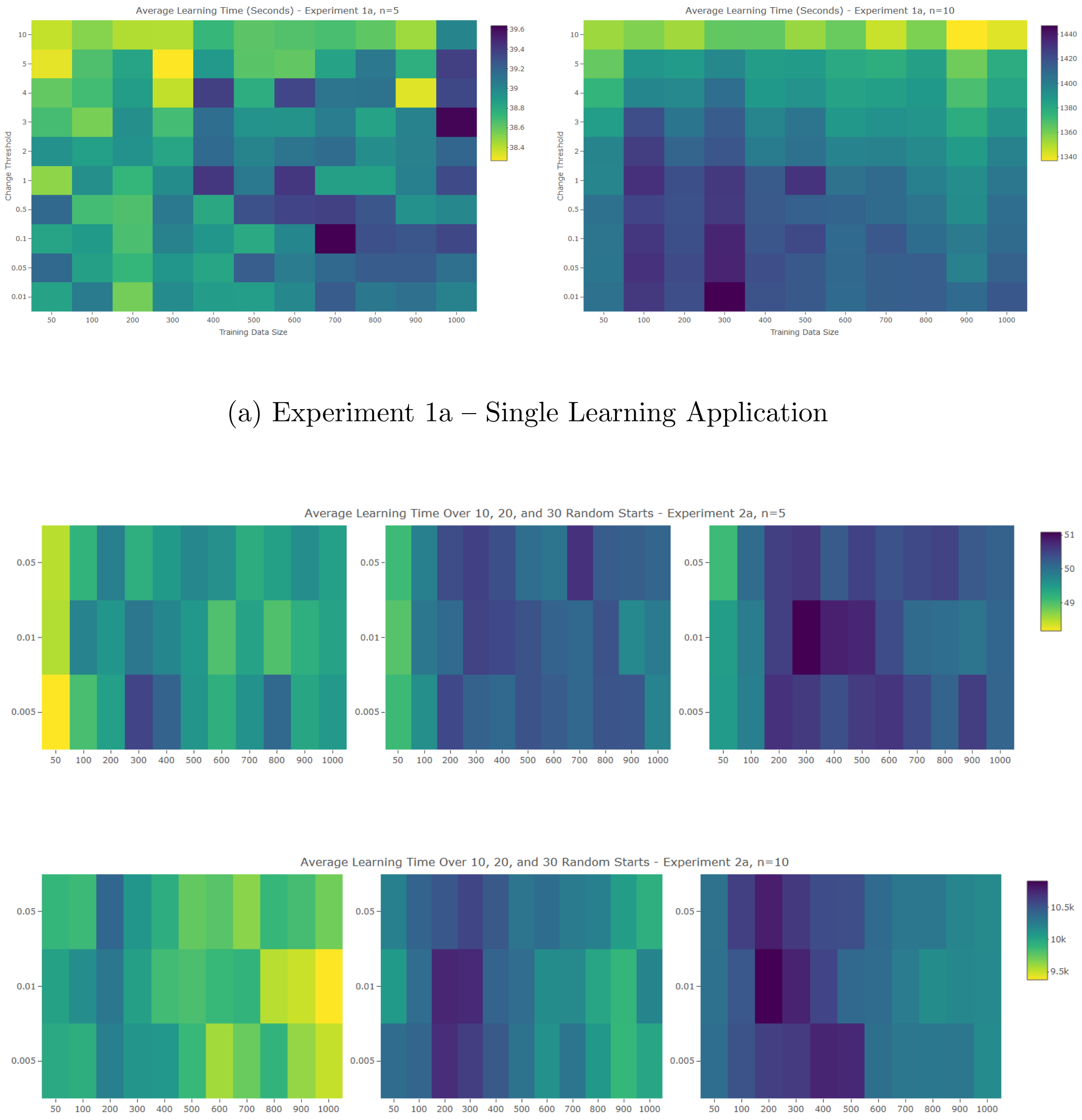
In experiment 1a, the best learned CP-net score we achieve is 72.68% probability of being supported for $n = 5$ and 35.16% for $n = 10$. Recall that these scores are the product of the CPT probabilities of support. Thus, these suggest a typical CPT support score (obtained by taking the n^{th} root) of 93.82% for $n = 5$ and 90.07% for $n = 10$. Thus, learning can obtain a CP-net that is strongly supported by the data. The minimum scores for these experiments suggest typical CPT scores of 86.63% for $n = 5$ and 83.52% for $n = 10$ so, while learning can fit any data set reasonably well, we do see significant improvement as the data more accurately represents a full preference ordering.

Adding random starts should always improve the learned CP-net score obtained as we pick the best score over multiple attempts (including the single empty start considered in experiment 1). The random start scores (experiment 2a) are between 0.001 and 0.009 higher than the scores obtained by a single attempt (experiment 1a). Increasing the number of random starts (from 10 to 20 to 30) also generally improves the average score, as we would expect, by around 0.001 or less, though in some cases additional random starts made the score worse. The latter is possible because the minimal score improvement may be outweighed by variation due to randomised starting structures and Dirichlet score approximations. The maximum score we can obtain from using random starts is 73.44% for $n = 5$ and 35.69% for $n = 10$. However, while random starts do improve our learning optimisation, previous results have shown that this corresponds to little to no improvement in the quality of the learned CP-net, suggesting that random starts are not worth their additional computation complexity.

The standard errors of the learned CP-net scores are also small. For $n = 5$, the standard errors of the CP-net scores range between approximately 1.31 – 1.58% for experiment 1a and between 1.33 – 1.54% for experiment 2a. For $n = 10$, the standard errors of the CP-net scores lie between approximately 0.65 – 1.25% for experiment 1a and between 0.66 – 1.24% for experiment 2a. CP-net score variability is generally lower when smaller training data sets are used. The value of α and the number of random starts (in experiment 2a) have no apparent effect on score variability.

From these results, we have seen that, in general, the behaviour of the learned CP-net scores mimics the performance of our other measures. This is encouraging as it suggests that our CP-net score is a sensible choice to optimise as this results in accurate learned CP-nets. These results have also shown that our learning algorithm can obtain CP-nets with very high support probabilities (meaning that our greedy optimisation works well), even when using only one learning attempt. The random starts also improve optimisation, showing that we can, at least sometimes, obtain a higher score by considering non-empty starting structures. However, the improvement to scores is fairly conservative and, from what we have seen from other measures, not worth the additional complexity.

Figure 4.6 shows the average learning times for the 'a' experiments. In the case of experiment 2, we recorded the total time it took to perform all k learning attempts. However, we have divided these times by k here for comparability. We have again omitted the results from the 'b' experiments as they are similar but do not include training data sizes above 750 or the $n = 10$ case in experiment 2. The time taken to perform learning depends on n, as this determines the size of the CP-net and number of scores we are considering, and the number of edge changes performed. However, not every score calculation takes the same amount of time. For example, calculating the score of a variable will take longer if it has more parents. Therefore, initial score and Δ calculation times will depend on the starting structure and Δ calculation times after each edge change will depend on the parent set size of the variable that lost or gained a parent. Similarly, calculating and updating the cycles matrix takes longer for denser structures. These additional



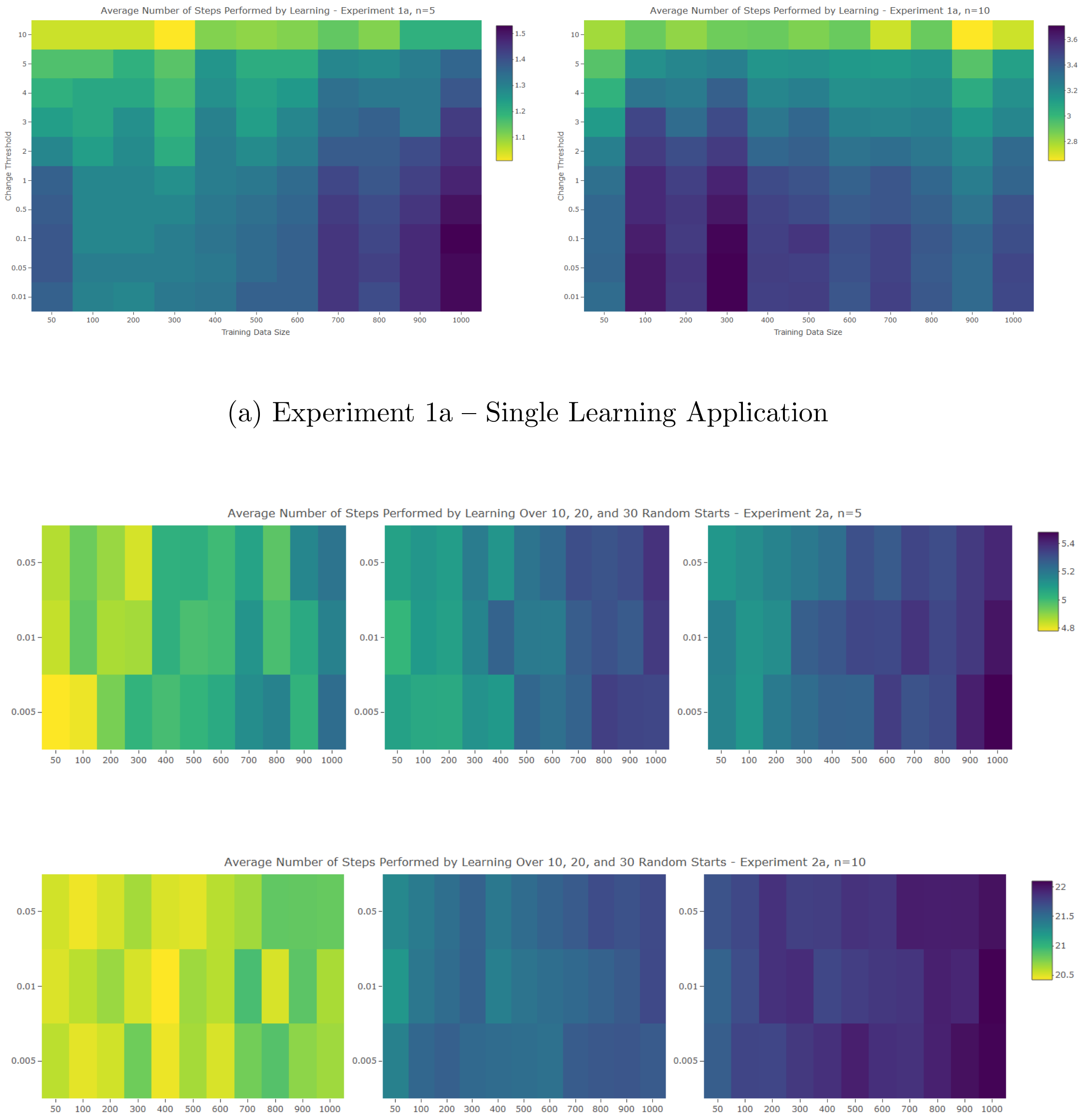
(a) Experiment 1a – Single Learning Application

(b) Experiment 2a – Learning With Random Starts

Figure 4.6: Learning Time Elapsed Results (No Cross Validation)

Yellow – Faster learning times

Blue – Slower learning times



(a) Experiment 1a – Single Learning Application

(b) Experiment 2a – Learning With Random Starts

Figure 4.7: Number of Edge Changes in Learning (No Cross Validation)
Yellow – Fewer edge changes
Blue – More edge changes

effects on learning time may explain the differences we see between the learning time and edge change results.

We give the average number of edge changes (the number of steps in the learning procedure) for the 'a' experiments in Figure 4.7. For experiment 2, we have similarly divided the total steps over k random starts by k, to give the average number of edge changes per learning attempt. The 'b' experiment results are omitted as they are, again, similar to the 'a' results but over a more restricted set of configurations.

As α gets smaller, more edge additions and fewer edge removals are considered 'valid'. As experiment 1 has an empty starting structure, we would expect learning to perform more edge changes in general for smaller values of α and we do observe this trend in the experiment 1a results. In experiment 2a, we do not see any clear effect of α on average steps. This may be because the net effect of more possible edge additions and less removals is minimal or non-existent in the random start case. Or it could be because the α values are quite close together and, thus, changing between them would not often affect whether an edge change is valid. If the effect is minor, then it could have been outweighed by variation due to the random starting structures and variation in Dirichlet score approximation.

We can also see a trend of longer learning times as α gets smaller for experiment 1a, though this behaviour is less consistent for $n = 5$. In experiment 1, we use empty starting structures and, thus, the only reason for deviation between the behaviour of steps and time is variation in the time required for Δ and cyclicity calculations after each edge change. The increase in average edge changes is fairly small for $n = 5$, with a range of 1 to 1.5 learning steps on average. This suggests that the increasing learning times are less consistent for $n = 5$ because the increase in average steps is small and, thus, more easily outweighed by the above learning time variations. We can see that the (small) increase in edge changes does not have a large effect on average learning times as they are all within 1.4 seconds of one another. On the other hand, the learning times for $n = 10$ closely follow the behaviour of the corresponding edge change results. This is likely to be because the increase in average steps (as α decreases) is larger for $n = 10$ and the effect of increasing edge changes is larger for $n = 10$ as edge changes take longer for larger values of n.

In experiment 2a, we see no clear effect of α on the average learning times. This is unsurprising as we saw no impact of the change threshold on the average number of steps either. The average learning times also do not mimic the average edge change behaviour as α changes, even for $n = 10$. This is likely to be because varying α results in only minimal change in the average learning steps and learning times have additional variation in experiment 2 due to the randomly selected starting structures.

Unlike in previous results, the size of the training data does not appear to have such a strong effect on the length of the learning process. This makes sense as larger training sets makes learning more accurate but, in practice, all training sets give a Dirichlet distribution of the same order to which we then try to fit a CP-net (from an empty or random start). There is, however, a general trend of increasing edge changes as the data size increases. This is unsurprising as larger training sets encode more preferences, which we must ensure are reflected by N_L. If a preference has support in the data, then edge changes to support this preference have higher Δ values. Therefore, if more preferences have support in the data, then more edge changes are made. On the other hand, if there is little data, we only have to ensure a few preferences are satisfied to have maximal agreement, all other preferences have little to no support (in either direction) from the data. The learning algorithm will not make edge changes for these preferences as it will not sufficiently improve the score.

The exception to this trend is the experiment 1a results for $n = 10$. Here, there is no clear effect of training data size, though there is perhaps some suggestion that the average edge changes get smaller for large training sets. It is not clear why the latter would be the case. It is possible that there is no clear effect for $n = 10$ because all data sizes ≤ 1000 are similarly 'small' when considering and fitting a preference order over 1,024 outcomes. Comparatively, this would be like considering only training sets up to size 32 for $n = 5$. Thus, the improvement (with respect to representing preference) in data size is minimal and, thus, could be being overshadowed by variability in training sets (which varies more for $n = 10$) and score approximation (which determines whether edge changes are valid or not). We do not see the same behaviour in experiment 2a for $n = 10$, which could be because the small increases in data have more of an effect when utilising non-empty starting structures.

In experiment 1a, we see that the learning times generally follow the same patterns as the average edge changes with respect to data size, with the $n = 10$ results mimicking the edge changes more closely, as we had before. For experiment 2a, $n = 5$, we see that training data of size 50 or 100 appear to be quicker but other than that, the data size has no clear effect on learning time. This matches up with what we see in the average edge change results, to a degree, as the increase in average learning steps slows after the smallest data sizes. The differences in behaviour between learning times and steps can be attributed to the variation in starting structures and cyclicity and score calculation times after each edge change, which cause variation in learning times even when the same number of steps occur. As the increase in average learning steps is small and results in little increase in learning times (particularly for the larger data sizes), it is unsurprising that the effect of increasing average steps is outweighed by this variation.

As for the $n = 10$ learning time results for experiment 2a, we generally see learning times decrease as data size increases (with the exception of the smallest training data of size 50). It is unclear why this happens, as the average edge change results appear to increase with data size. Learning times also depend upon the random starting structures as well as cyclicity and score calculation times after each edge change (which all vary more for $n = 10$ as the CP-net structure is larger). However, there is no apparent reason why either of these factors would cause learning times to decrease for larger training sets.

Recall that previous results have suggested optimal performance occurs when using the smallest values of α and maximal training data. From the experiment 1a results in Figures 4.6 and 4.7, we can see that learning with these configurations takes approximately 39.1 seconds and performs 1.5 edge changes on average for $n = 5$. For $n = 10$, learning under these configurations takes approximately 1,410 seconds (23.5 minutes) and performs around 3.5 edge changes on average. Both average times are impractical if we need to apply learning a large number of times. Furthermore, it is likely that real applications will require more than ten variables, making even one learning attempt potentially impractical. Thus, while our learning performs well, we must find a way of making it more efficient. This is our primary interest for our future work, as we discuss in more detail in §4.5.

The average number of edge changes are fairly low, meaning that our learning algorithm is not exploring much of the search space. Furthermore, as experiment 1 used empty starting structures, these edge changes suggest that, on average, N_L has 1 or 2 edges for $n = 5$ and up to 4 edges for $n = 10$. Thus, learning produces fairly sparse CP-nets, which is as we predicted due to the fact that removing edges is easier than adding them (because of their respective Δ thresholds). Sparse N_L structures may also mean that some of the N_T preferential dependencies are not captured. However, our PG similarity scores were high, suggesting no major relationships were missed. Also, recall that our aim is not to recreate N_T but to be consistent with the preference order. Nevertheless, in future experiments, we may want to make it easier to add edges, both to enable further exploration of the model space and to allow denser models to be considered. It is possible that learning more complex structures may increase the likelihood of learning incorrect preferences from noisy data, whereas sparse structures are more likely to encode correct information but perhaps miss out some of the finer details of the model. This is something to consider in our future work.

Recall that, for experiment 2, we gave the average time and steps for each learning instance (each random start). For the larger data sizes (where learning has previously been seen to perform better, α has little to no effect), the average learning time is between 49 and 50.5 seconds for $n = 5$ and between 9,500 and 10,300 seconds (between 158.3 and 171.7 minutes) for $n = 10$. This shows that learning from a non-empty starting structure takes significantly longer and is even less practical. These longer learning times are due to both the larger number of average learning steps and the fact that dealing with denser structures will make score and cyclicity calculations take longer. This increase in time is more significant for $n = 10$ as there is a larger increase in average edge changes, each edge change takes longer for larger n, and the randomised structures will usually contain more edges than the randomised starts for $n = 5$. The average learning times also grow as we increase the number of random starts (from 10 to 20 to 30). This is probably because the proportion of (faster) learning instances using an empty starting structure is decreasing (from 1/10 to 1/20 to 1/30), meaning the average learning time increases, getting closer to the average learning time when using a randomised start.

The total learning times for experiment 2a (over all random starts), for the larger training sets, were approximately 495 seconds (8.25 minutes) for 10 random starts, 1,000 seconds (16.7 minutes) for 20 random starts, and 1,515 seconds (25.25 minutes) for 30 random starts when $n = 5$. For $n = 10$, the total learning times were approximately 95,000 seconds (26.4 hours) for 10 random starts, 200,000 seconds (55.6 hours) for 20 random starts, and 306,000 seconds (85 hours) for 30 random starts. As expected, using random starts takes significantly longer. However, as randomised starting structures make learning slower, using k random starts takes longer than k times as long as experiment 1 learning instances. From these values, we can see that using multiple random starts is impractical, particularly if there are more than 5 variables. Furthermore, previous results have shown that using random starts results in minimal to no improvement to the quality of the learned CP-net and is, therefore, not worth the drastic increase in computation time.

The average number of edge changes performed by each random start in experiment 2a (again, for larger training sets) ranges between 5 and 5.5 for $n = 5$ and between 20.5 and 22.1 for $n = 10$. This is significantly larger than the average number of learning steps in experiment 1, meaning that more edge changes occur when using a randomised starting structure. Furthermore, as we use additional random starts (increasing from 10 to 20 to 30), the average edge changes increase, due to the same reasoning as the time elapsed results. Randomised starting structures enables the search to make more edge changes and explore a larger area of the search space. However, the larger number of edge changes could be because most edges in the starting structure are incorrect or irrelevant and are, thus, removed – this has been observed in several examples of learning with randomised starts. The average number of edge changes are consistent with this possibility and it is further supported by the fact that we see little performance improvement from random starts. This would imply that learning from a random start takes longer, in part, due to the time taken to remove most of this random structure.

The standard errors of the learning times and the number of edge changes are slightly high (in comparison to their respective data values), but not excessive. This is particularly true for the time elapsed results and for the experiment 1 results. The following figures give the standard error of the learning times for experiment 1a and, for experiment 2a, the standard error of the average learning times for a single random start (obtained from the total time by dividing by the number of random starts, k, as we did for Figure 4.6). For $n = 5$, the standard errors lie between approximately 0.17 – 0.43 seconds for experiment 1a and between 0.18 – 0.45 seconds for experiment 2a. For $n = 10$, the standard errors lie between 11.46 – 17.04 seconds for experiment 1a and between 67.74 – 159.69 seconds for experiment 2a. The variability of learning time does not appear to be dependent upon either α or the size of the training data. However, in experiment 2a, the variability of an average single learning attempt time decreases as the number of random starts increases. This may be because the effect of having a single empty start among k random starts (which will add to learning time variation) diminishes as k increases.

The following figures give the standard error of the number of edge changes performed by learning for experiment 1a and, for experiment 2a, the standard error of the number of edge changes performed (on average) by a single random start (obtained by dividing the total number of edge changes by k, as we did for Figure 4.7). For $n = 5$, the standard errors lie between approximately 0.077 – 0.10 edge changes for experiment 1a and between 0.043 – 0.79 edge changes for experiment 2a. For $n = 10$, the standard errors lie between 0.12 – 0.15 edge changes for experiment 1a and between 0.075 – 0.14 edge changes for experiment 2a. For experiment 1a, the variability of the number of edge changes is higher for smaller values of α. Experiment 2a shows no effect of α on variability, which may be because the tested values are all fairly similar. For $n = 5$ (both experiments), variability also increases for larger data sizes. For $n = 10$, there is no clear effect of the training size on variability. For experiment 2a, the standard error of the average edge changes in a random start decreases as k increases (which may be caused by the same reasoning as above).

From the results in Figures 4.6 and 4.7, we have seen that the learning algorithm generally performs more steps and takes longer for smaller values of α (though there is no noticeable effect in experiment 2). As previous results suggest learning performs better for smaller α, this means that better performance comes at the cost of longer learning times. For $n = 5$, we also see a general increase in the number of steps and learning times as the training set grows and there are more preferences for the algorithm to model (which, again, means that as performance improves, learning times get longer). The effect of data size is less clear for $n = 10$, perhaps because all training data up to 1000 data points is similarly small as representative data over 1,024 outcomes. We will need to run more experiments with larger training sets in our future work to evaluate the relation between data size and learning time and steps in this case (though we expect it to be similar to $n = 5$).

From the experiment 1 results, we find that even one learning attempt is impractically long. Despite these long learning times, few edge changes are performed by learning. Thus, the algorithm does not explore much of the model space and, as we start with an empty structure, the learned CP-nets are fairly sparse. In our future experiments, we may consider making it easier to add edges. This is supported by previous results which suggest performance would improve for smaller values of α. The experiment 2 results show that learning from randomised starts performs significantly more steps and takes much longer. In the $n = 10$ case, a single learning attempt with a randomised starts takes hours. As randomised starting structures increase average learning times, using k random starts takes significantly longer than k times the single learning times from experiment 1. The total learning times for k random starts ($k = 10, 20, 30$) are completely impractical and confirm that the minor performance improvements from random starts are not worth the additional complexity costs. Our primary aim in our future work will be to address the long running times of our learning algorithm. We will also consider alternative ways of improving the greedy search optimisation, such as random walks or modifications to our random starts. This is discussed in more detail in §4.5.

We also observe from the experiment 2 results that the random start utilising an empty starting structure is more likely to be the optimal random start (obtaining the highest score) than a randomised starting structure. In particular, for $n = 5$, the empty start is between 1.25 and 1.38 times more likely to be the optimal random start than any of the other random starts, which use randomly generated starting structures. For $n = 10$, the empty start is between 4.32 and 4.63 times as likely to be optimal. These multiples decrease as the number of random starts attempted increases. This shows that, when learning is only attempted once (as in experiment 1), using an empty starting structure is likely to perform better than a randomly generated starting structure. This could be because the edges in a random structure are likely to be incorrect or irrelevant. Such edges may be a hindrance to learning progress, either because they must first be removed or by making useful edge changes impossible to perform due to cyclicity problems. In contrast, the empty structure makes no assumptions and only relations supported by the data will be added. This could explain why the empty structures are even more likely to be optimal in the $n = 10$ case, as the randomised structures are likely to have more edges than for $n = 5$. The fact that the empty starting structure is the most likely to be chosen also helps to explain, particularly for $n = 10$, why we see little improvement in learning performance from using random starts.

In this section, we have seen that our learning algorithm performs well according to all four of our performance evaluation measures. More specifically, this means that our learned CP-net, N_L, has a high level of agreement with the original CP-net, N_T, on swap preference, as well as a high level of consistency with respect to general preferences. Furthermore, we found that N_L is consistent with unseen data, performing almost as well as N_T with respect to both swap and general preferences. Finally, we also found N_L to be consistent with a large proportion of the true preference ordering, which was our primary aim in learning. This means that N_L will be strongly consistent with large sub-orderings of the true preference order. In fact, in some cases, learning achieved strong consistency with the total ordering (meaning N_T was recovered). The quality of N_L according to these measures also matches up with the N_L score results, implying that we constructed a sensible score to optimise. Optimal performance generally occurs when larger training sets are used and for smaller values of α. We also usually see better performance for $n = 5$ as training sets of size ≤ 1000 cannot reflect the full preference order for $n = 10$. However, in general, the performance results for $n = 10$ are not far behind $n = 5$. This is encouraging as it suggests that the algorithm can learn effectively even with only partial information (even 1000 data points is a small set for $n = 10$, to fully reflect a preference order over 1,024 outcomes, we would need around 500,000 data points). This is important as full information is impractical for larger values of n and it is likely that real world applications will not always have such complete data sets to learn from.

Our initial set of α values (tested in experiment 1) was fairly wide as it is not obvious what is an appropriate percentage change such that valid improvements are above this threshold and score improvements due to noise in the data or score estimation error fall below. As learning results improve as α decreases in experiment 1, we used a smaller set (for practicality) of lower values of α in experiment 2. We expected that, as α continued to decrease, learning performance would improve until α became too small and started allowing edge changes due to noise or estimation error. However, in general, we saw no clear effect of α on performance in experiment 2, possibly because the small set of values tested were all fairly close together. Thus, in future experiments, we would like to consider smaller values of α, in order to find the optimal change threshold.

We find, over all performance measures, that the random starts utilised in experiment 2 result in minimal or no improvement to N_L. Furthermore, when using these random starts, we find that it is most likely to be the empty starting structure (as we used in experiment 1) that is found optimal and, thus, returned as N_L. The large increase in learning time when using random starts is not feasible for practical use; it takes longer than k times a single learning attempt (which alone are not quick to perform) when applying k random starts as randomised starting structures result in longer learning times. Thus, the minimal improvements to learning performance resulting from random starts are not worth their increased time costs. In our future work, we would like to consider how else we might improve our greedy optimisation method, perhaps by modifying random starts or utilising random walks. This is discussed in more detail in §4.5.

For a single learning attempt, as we use in experiment 1, the time costs are not unreasonable. However, they are impractical for applications that require many learning attempts or a larger number of variables. Furthermore, these times are generally longer for the learning configurations we find most successful (more training data and smaller α values). Our primary goal in our future work is to improve learning efficiency whilst preserving the good performance results we have seen here, as we discuss in §4.5.

4.5 Discussion

4.5.1 Summary

In this chapter, we have have introduced our new CP-net learning method. This method is more widely applicable than existing techniques due to several properties. First, it uses user choice data rather than pairwise outcome preferences.

Second, we allow this data to contain noise and we do not assume that the user's true preferences are representable by CP-nets. Finally, our method may return CP-nets with any acyclic structure, whereas other methods often assume restrictions upon the in-degree of the learned structures.

We also provided an experimental evaluation of the performance of our learning algorithm. These results found that maximising our constructed CP-net score produces learned CP-nets that agree strongly with both the user's true CP-net and preference order, as well as previously unseen data sets. In particular, our learning algorithm can achieve over 95% consistency with the user's true preference order (our primary aim in learning), even when the data reflects only partial preference information. However, a single run of our learning algorithm took up to 40 seconds for $n = 5$ and 24 minutes for $n = 10$. Thus, a single attempt will quickly become impractical for larger n and any application that requires many learning attempts will be infeasible even for small values of n. Thus, while our learning algorithm performs well, further work is required to make it efficient enough to be usable in practice. We discuss some possibilities for how to improve this efficiency below.

In these experiments, we also tested a second variation of our learning algorithm in which we utilised several randomised starts in order to improve the CP-net score optimisation. In general, we found that considering randomised starts did not improve the learning performance by much. Furthermore, randomised starting structures greatly increase the already long learning times. Thus, considering k random starts took even longer than k times a single learning run. Consequently, the total learning times for randomised starts were completely impractical and far outweighed any minor benefit to learning performance. Thus, in this form, randomised starts are not worth considering. In our future work, we intend to consider how we can improve the performance of random starts as well as other methods of improving the CP-net score optimisation, as we discuss below.

4.5.2 Potential Improvements to Our Learning Algorithm and Further Experiments

There are several possible directions for future work on our learning algorithm. Perhaps the most important is addressing the current limitations on its practical application. As we have seen in our §4.4 experiments, the run-time of our algorithm becomes impractical for $n = 10$. Further, our complexity evaluations in §4.3.3 showed our learning method to be intractable. This complexity only gets worse if the random starts variation discussed in §4.3.4 is used. This means that our learning algorithm is unlikely to be efficient enough for practical purposes.

Particularly if the data updates regularly, meaning the learned structure would need routinely updating also. These long run-times are primarily caused by the calculation of scores (and Δ values). In Appendix D, we discuss how the score estimation imposes requirements upon system storage and accuracy that limit the size of n for which we can learn. Reducing the size of the Dirichlet sample used for estimation would improve both efficiency and the storage requirements, but at a cost to the accuracy of our estimations. In the future, it may be worth evaluating what effect lowering this accuracy has on learning performance and whether the improvement to efficiency is worth it. However, an entirely new method for score estimation is likely to be required in order to make our algorithm applicable to n values much larger than 15.

We would also like to consider different methods of optimising our score over the space of acyclic structures, other than greedy search. In particular, one might consider using a genetic algorithm. Haqqani and Li (2017) used a genetic algorithm for optimisation and their learning returned reasonable CP-nets in less than 25 minutes for $n = 100$. As CP-nets scale exponentially with n, this is far beyond what our method can manage currently. Thus, it is possible that, in addition to improving performance, an alternative optimisation method may also improve the running time of our algorithm. However, a different score estimation method will still be required for larger values of n.

Alternatively, we could consider variations to the greedy search method other than randomised starts. As we mentioned in §4.3.4, another way we may improve our learned CP-net score (and, thus, learning performance) is by implementing random walks within our greedy search. Suppose our learning algorithm has starting structure A_1 and makes k edge changes to reach a locally optimal structure, A_2. If we picture the space of acyclic structures as a surface, with score dictating height, then we have successively made the k steepest (valid) climbs from A_1 and all possible steps from A_2 are not sufficiently steep to be 'worth' moving. However, it is possible that, if we moved away from A_2 slightly, we may be able to keep climbing even higher. For example, it is possible that the presence of one edge, e_1, makes the addition of some other edge, e_2, impossible due to the creation of cycles. As our learning algorithm considers only one edge change at a time and allows only valid changes, it cannot consider adding e_2 (possibly by first removing e_1), even if it would greatly improve the score. Such changes may be explored by implementing random walks.

Returning to our scenario, we want to see whether the peak we climbed can go any higher. To do so, we want to move away from A_2, but we do not want to destroy all of the progress we have made. Thus, we perform a random walk from A_2

that changes $c < k$ edges and attempt to continue climbing (that is, we perform our search algorithm again from this new starting structure). However, if we only change one edge, it is likely that the search will go straight back to A_2, so we ensure that $c \geq 2$. Performing c edge changes creates a new acyclic structure, A_2', that is not too far from A_2. Learning is then performed using A_2' as the starting structure, returning a locally optimal structure, A_3. If we have improved upon A_2 (that is, $S(A_3) > S(A_2)$), then we repeat this process of searching for a higher peak by randomly walking from A_3. However, if $A_3 = A_2$ or $S(A_3) \leq S(A_2)$, then we do not move to A_3 and this is counted as a strike. In this case, we randomly select c again and perform another random walk from A_2, then perform learning from this new starting structure. If enough strikes occur, then we stop random walking and return the CP-net with structure A_2. This random walk variation could also be combined with random starts. In this case, we would perform learning with random walks from a set of randomised starting structures, then return the highest scoring of the learned structures.

Another variation we may consider, as we alluded to above, is enabling the learning algorithm to consider the effect of more than one edge change in advance. By allowing the search to consider more complex moves over the model space, we may be able to improve our optimisation performance.

There are also two possible improvements to random starts that we might consider in our future work. Firstly, as we discussed in §4.4.1, we should use the same Dirichlet sample for estimation over all k learning attempts – this makes the comparison of the k learned structures more exact. Secondly, we could aggregate the k learned CP-nets in order to form an even better structure. For example, suppose the k learned structures are $N_L^1, ..., N_L^k$. Each structure has a score, but this score is simply the product of the CPT scores. For this example, let us consider the scores to be $|V|$-vectors of the CPT scores. For example, $S(N_L^1) = (S_{X_1}^1, S_{X_2}^1, ..., S_{X_n}^1)$, where $S_{X_i}^1$ is the score of CPT(X_i) in N_L^1. The structure score of N_L^1 is then simply the product of this vector. Now, suppose that N_L^1 has the highest score of all the learned CP-nets. It is possible that, while N_L^1 has the highest score, we may also have $S_{X_i}^2 > S_{X_i}^1$. That is, CPT(X_i) has a higher score in N_L^2 than in N_L^1. If it is possible to change the parents of X_i in N_L^1 to the parent set of X_i in N_L^2 (without affecting acyclicity), then we can obtain a CP-net with a score that is even higher than N_L^1. Let N_L^* be the structure obtained from N_L^1 by changing the parents of X_i to be the parent set of X_i in N_L^2. Then $S(N_L^*) = (S_{X_1}^1, ..., S_{X_{i-1}}^1, S_{X_i}^2, S_{X_{i+1}}^1, ..., S_{X_n}^1)$. Thus, N_L^* has a greater score than N_L^1 (note that this assumes the same Dirichlet sample is used in all k learning attempts, as discussed above).

One might start with the highest scoring CP-net, and try to successively change any parent set that can be improved. This would be done in order of the magnitude of the score improvement, and only implemented in the cases where acyclicity is preserved. Alternatively, one might try and construct the best scoring (acyclic) combination of parent sets from the learned structures via some optimisation procedure. As we mentioned previously, relations that appear in more of the N_L^i may be considered more likely to be part of the user's true preference structure. Thus, one might allocate such relations more weight in these aggregations.

Further experiments on our current learning method is another important direction for future work. Firstly, our previous experiments suggest that performance improves as the change threshold, α, gets smaller. We predict that when α becomes too small, performance will decline as it will allow edge changes due to noise in the data or score estimation error. We would like to evaluate the performance of a wider range of values of α in future experiments, in order to identify this change point and, thus, the optimal threshold.

Secondly, we would like to conduct experiments varying the uninformed prior parameters, in order to see how this affects learning performance. We would also like to consider how informed priors might be constructed and evaluate learning performance in these cases as well. In particular, whether accurate informed priors improve learning performance. A further discussion of prior possibilities was given in §4.3.2.

Thirdly, experiments that use data not (necessarily) representable by a CP-net, or with different p_i distributions, would illustrate how our learning procedure may perform on real data. Finally, we would like to conduct a direct experimental comparison between our learning method and some of the other existing methods.

Another aspect we would like to consider in any future experiments is examining directly the structural similarity of the true and learned CP-nets. In the existing experiments, we primarily looked at the similarity of the implied preferences, which is related but not directly linked to structural similarity. Structural similarity will give us an indication as to whether the relations we are extracting from the data are true or not. As our experiments showed little movement within the search space, few edges are added in the empty start case. Such an analysis might reveal whether learning can only find the 'most important' of the true relationships, or whether it is approximating the truth with the closest sparse structure (whose edges may be distinct from the original CP-net). We might also look at the effects of graph properties such as connectivity or density (of the true CP-net structure) on the learning algorithm performance. Perhaps the algorithm is more likely to identify the true relationships if the true CP-net is sparser. Equally, it would be interesting to see the effect of these properties on the other measures of performance that we considered previously.

4.5.3 Other Future Work

Applying our algorithm to real world data is another interesting direction. One possible application is to the user data available on Steam. Steam is a gaming platform with a public web API, through which one can obtain information about users: their activity on the platform, their friends, and about the games they play. O'Neill et al. (2016) has already used the Steam API to compile a comprehensive database. We suggest that game features such as multi-player or not, age rating, price, whether it is owned by the user's friends, and game genre could be used as the CP-net variables. The number of hours a user has played a game can be used as the user choice data. From this data, we can learn a CP-net that dictates the user's preferences over different game categories. This model can then be used for personalised advertisement, for example.

Our learning method can be extended to non-binary CP-net learning. In order to do so, we need to generalise our CPT score to non-binary CP-nets. Once this is defined, learning can proceed as before; the score of a structure is the product of the maximum CPT score for each variable. Note that we can use the same method as in Appendix D.2 to estimate the optimal CPT scores for a given structure. We optimise this new structure score over the space of acyclic structures as we did before. The Δ values are defined as before and the same update procedures can be performed after each structural change. Thus, it only remains to define our CPT score for non-binary CP-nets.

Let Ω denote our set of outcomes and $|\Omega| = \mathcal{O}$. Suppose $\Omega = \{o_1, o_2, ..., o_{\mathcal{O}}\}$. Let p_i denote the probability that the user chooses outcome o_i. We can again utilise an uninformed Dirichlet prior for the p_i values. We then observe some user choice data. Let $d(o_i)$ denote the number of times the user chose outcome o_i. This gives us the same posterior distribution over the p_i as before:

$$p_1, ..., p_{\mathcal{O}} \sim Dir(\beta_1 + d(o_1), ..., \beta_{\mathcal{O}} + d(o_{\mathcal{O}})).$$

Now consider CPT(X), for some $X \in V$. Let $|\text{Dom}(X)| = m$. A typical row of CPT(X) has the form $\mathbf{u} : x_1 \succ x_2 \succ \cdots \succ x_m$, where $\mathbf{u} \in \text{Dom}(\text{Pa}(X))$. Let $W = V \backslash \{X\} \cup \text{Pa}(X)$. This rule represents $|\text{Dom}(W)| m(m-1)/2$ pairwise outcome preferences (note that $|\text{Dom}(W)| \geq 2^{|W|}$, with equality only if all variables in W are binary). In particular, for each $\mathbf{w} \in \text{Dom}(W)$ and $1 \leq i < j \leq m$, this rule dictates that $\mathbf{u}x_i\mathbf{w} \succ \mathbf{u}x_j\mathbf{w}$. This set of preferences is the transitive closure of $m-1$

preferences: $\mathbf{u}x_1\mathbf{w} \succ \mathbf{u}x_2\mathbf{w}$, $\mathbf{u}x_2\mathbf{w} \succ \mathbf{u}x_3\mathbf{w}$, ..., $\mathbf{u}x_{m-1}\mathbf{w} \succ \mathbf{u}x_m\mathbf{w}$. Thus, the CPT rule represents the transitive closure of $|\text{Dom}(W)|(m-1)$ pairwise outcome preferences. An outcome preference, $o_i \succ o_j$, is supported if $p_i > p_j$. However, as each CPT rule represents an exponential number of pairwise preferences, we again simplify the condition for support. The rule $\mathbf{u} : x_1 \succ x_2 \succ \cdots \succ x_m$ is *supported* if

$$\bigwedge_{1 \leq i < m} \sum_{\mathbf{w} \in \text{Dom}(W)} p_{\mathbf{u}x_i\mathbf{w}} > \sum_{\mathbf{w} \in \text{Dom}(W)} p_{\mathbf{u}x_{i+1}\mathbf{w}}.$$

We are abusing notation here, using p_{o_i} to denote p_i for practicality. A CPT is supported if all of its rules are supported. The score of a CPT is the probability that it is supported.

$$\begin{aligned}
&S_t(\text{CPT}(X))\\
=&\Pr(\text{CPT}(X) \text{ is supported})\\
=&\Pr(\text{All rules in CPT}(X) \text{ are supported})\\
=&\Pr\left(\bigwedge_{\mathbf{u}:x_1 \succ x_2 \succ \cdots \succ x_m \in \text{CPT}(X)} \left(\bigwedge_{1 \leq i < m} \sum_{\mathbf{w} \in \text{Dom}(W)} p_{\mathbf{u}x_i\mathbf{w}} > \sum_{\mathbf{w} \in \text{Dom}(W)} p_{\mathbf{u}x_{i+1}\mathbf{w}}\right)\right).
\end{aligned}$$

This support condition for CPTs is more complex to check than the condition in the binary case. Further, non-binary CP-nets have more outcomes and larger CPTs. Due to these properties, the complexity of calculating scores will be greater in the non-binary case and the corresponding storage requirements will also be greater. Thus, non-binary learning will have more restrictions upon possible n values. Thus, finding a more efficient score estimation method is even more important in this case. Perhaps further simplification of the above definitions could make the score calculation complexity closer to the binary case.

We would like to perform similar experiments in this non-binary case and determine what size of non-binary CP-net we can learn currently. Further, we would like to make the non-binary method more widely applicable (with respect to n and domain sizes) and efficient, similarly to the binary case.

Another modification we would like to explore is learning CP-nets with indifference. As CP-nets encode preferences over a large number of outcomes, indifference between outcomes is likely to occur in real world applications. Suppose we have $x, \bar{x} \in \text{Dom}(X)$. We might say that the user is (conditionally) indifferent between x and $\bar{x}$ if the support probabilities for $x \succ \bar{x}$ and $\bar{x} \succ x$ are sufficiently close. Alternatively, one might say that they are indifferent if there is a high probability that the relevant p_i sums are sufficiently close. Either definition would require the CP-net score to be redefined.

It would be useful to make our learning applicable in the online case also. In this scenario, we receive small amounts of data over time. The aim is to obtain, at every time point, a CP-net that is consistent with all data observed so far. This requires updating our learned CP-net when a small amount of new data is observed. In order to be practical, this update procedure should be fairly efficient. However, due to the construction of our score function, a quick update procedure does not appear possible. Suppose we observe some data, D_1, and learn a CP-net, N_1. Suppose we then observe further data, giving us a larger data set, $D_2 \supseteq D_1$. We now want to update N_1 so that it is consistent with D_2. Even if D_2 has only one more data point than D_1, all scores will need re-calculating from scratch. Thus, the best we can do is to use N_1 as an informed starting point for learning with data set D_2. This informed starting point may mean the learning performs fewer steps than if an empty or random start is used, but this is still not an efficient update process. As we mentioned above, this may be made quicker by a different estimation process. However, this still falls under repeated batch learning, rather than updating as is usual in online learning.

We would like to look into whether our learning technique can be simplified or transformed into one that can be applied in online learning contexts – perhaps by simplifying our score function. If this is not possible, we would like to define a different learning method for online learning from user choice data. Note that there is currently no such existing method in the literature.

We used two simplifications when defining our CP-net score in the binary case. The first was simplifying the support condition for a preference rule into a single inequality. The second was defining the score of a CP-net to be the product of its CPT scores. The latter treats CPTs independently, ignoring the possible probabilistic dependence between rules in distinct CPTs. Both simplifications were made for the sake of practicality. If these simplifications were not made, then our search space would be the space of all acyclic CP-nets over n variables. Further, every CP-net must have its score calculated from scratch – it cannot simply be updated if a change is made. Note that the score of a CP-net is now the probability that all of the pairwise preferences it encodes are supported. However, these probabilities will not sum to one as there are p_i orderings that are not consistent with any CP-net. Thus, it may be that all CP-net scores are very small which can cause problems in estimation and comparison. Optimising the score without these simplifications will be far more complex. However, for completeness, one might evaluate the learning performance when one or both of these simplifications are not utilised. These results would demonstrate whether these simplifications result in any significant decrease in performance. Further, this would show whether the practicality of using these simplifications is worth any associated cost to performance.

A more distinct direction we may consider in our future work is how user choice data might be used in constraint based learning (instead of score based). This has been considered for CP-net learning from preference data but not yet for choice data. The constraint based approach attempts to detect the significant relationships in the data to build the CP-net structure. Such approaches have been developed for learning Bayesian networks, but they are unlikely to be appropriate for CP-nets due to the differences in symmetry.

Chapter 5

Conclusions

In this thesis, we addressed two of the key problems with using CP-nets as models for user preference in practice: being able to answer dominance queries efficiently and learning a user's CP-net from choice data. We introduced two methods of improving dominance testing efficiency (rank pruning and CP-net preprocessing), which have both been shown to significantly improve dominance testing time. Furthermore, they can be used in combination and alongside existing methods to improve dominance testing efficiency even further. We have also designed a novel learning procedure for CP-nets using choice data. We have shown that the learned CP-nets both successfully model the user's true preferences and agree with previously unseen (future) data. The problem of learning from choice data has received little previous attention, despite the fact that choice data is a more realistic format for user preference data than the more commonly used pairwise preferences.

Dominance testing is an important requirement for reasoning with user preferences modelled by a CP-net. However, unlike other reasoning tasks, answering dominance queries is complex when using a CP-net model (in fact, it is PSPACE-complete for CP-nets in general). In Chapters 2 and 3, we developed two distinct methods of improving the efficiency of answering dominance queries, which can be combined for further efficiency. In Chapter 2, we constructed outcome ranks, a quantitative representation of user preference for a given outcome. These outcome ranks can be used to prune the dominance testing search tree in order to improve efficiency. We showed via experimental comparison that rank pruning results in significantly faster dominance testing times than other existing pruning methods. Furthermore, when considering combinations of pruning methods, we found that rank pruning is a critical component in order to have an efficient pruning schema.

As outcomes ranks provide a quantitative measure of preference, they can also be used to obtain consistent preference orderings over the outcomes or any subset of outcomes. They can also be used to obtain consistent outcome orderings under plausibility constraints. In some cases, using outcome ranks allows us to obtain such orderings more efficiently than existing methods. We can also use properties of outcome ranks to answer ordering queries for CP-nets in a new way. In Chapter 2, we also constructed a more generalised outcome rank that is defined for CP-nets that may have indifference statements in their CPTs and simplifies to the original ranks in the special case of no indifferences. As these generalised ranks also reflect all preferences encoded by the CP-net, they have the same properties as the original ranks. Thus, all of the above results, including dominance query pruning, also apply to our generalised ranks. In many cases, this is the first time these results have been achieved in the case of indifference as previous authors have not considered this generalisation. Indifference between outcomes is a likely occurrence in real world applications and so these generalised ranks increase the applicability of our results.

In Chapter 3, we approached the problem of efficient dominance testing from a different perspective – CP-net preprocessing. We introduced a new method of preprocessing a CP-net by identifying and iteratively removing variables that are unimportant to the relevant dominance query. The resulting, reduced CP-net and query are then partitioned into mutually independent sub-queries. The result is a set of queries over much smaller CP-nets that can answered separately (and simultaneously, if possible, to further improve efficiency). We refer to this method as UVRS preprocessing. The reduced queries are equivalent to the original; the original query is true if and only if all reduced sub-queries are true. This means that finding any sub-query false is sufficient to answer the original problem. The space of outcomes we must search over to answer the reduced set of queries is exponentially smaller than the search space of the original query. Thus, preprocessing significantly reduces the size of the original dominance testing problem. We can combine this preprocessing with our work from Chapter 2 for an even more efficient dominance testing process by utilising rank pruning (or an efficient pruning combination) to answer the reduced sub-queries.

We have also shown how UVRS can be applied in combination with the existing CP-net preprocessing method, forward pruning (Boutilier et al., 2004a). This combination is more effective than both methods used individually as forward pruning enables UVRS to remove more of the CP-net than it does when used in isolation. The method of combining the two also makes it reasonably efficient to apply, in fact, we found that applying the combination is faster than forward pruning alone in our experiments. We provided an experimental evaluation and comparison of the performances of the three preprocessing methods – UVRS, forward pruning, and their combination – on binary CP-nets. In these experiments, we utilised the most effective pruning schema from Chapter 2 to answer the queries and compared the efficiency of answering the unprocessed query to the efficiency of applying preprocessing and then answering the reduced query (or queries). These experiments found that UVRS improves dominance testing efficiency significantly more than forward pruning and the combination of methods performs even better. Furthermore, preprocessing appears to perform better for CP-nets with more variables. This is advantageous as dominance queries are generally harder for larger CP-nets and, thus, reducing query efficiency is of more interest in these cases. For larger binary CP-nets, we found that using UVRS can halve dominance testing times and using the combination can reduce times by up to 60%, even when using an already efficient pruning method to answer queries. Thus, by combining the methods from Chapters 2 and 3, we achieve significantly more efficient dominance testing.

In order to use CP-nets in practice to model and reason with user preferences, we first need to determine the user's CP-net. However, eliciting the CP-net directly from the user is not always possible or practical and may lead to inaccurate models due to human error or change in preferences over time. Thus, we want to be able determine a user's CP-net from observed data (passively). In Chapter 4, we introduced a new method of learning a user's CP-net from observed choice data. Most existing work on CP-net learning uses a collection of pairwise preferences as data. However, in many contexts it is not possible to observe pairwise preferences. Rather, we can only observe which outcome was successful (the item the user chose). Our learning procedure uses such choice data, as it is more realistically observable. We also relaxed other common assumptions such as the consistency of the data, the requirement that the user's true preferences are representable by a CP-net, and conditions upon the learned CP-net structure (other than acyclicity). We constructed a CP-net score that measures the agreement between a set of choice data (also allowing prior beliefs about user preference to be taken into account) and the preference rules represented by a given CP-net. Given a CP-net structure, it is simple to evaluate the CPTs that maximise this score. Thus, our learning procedure explores the space of acyclic CP-net structures, attempting to maximise the agreement score between the learned CP-net and observed choice data. Currently, the CP-net score (and, hence, the learning procedure) is only defined for binary CP-nets. The only other existing method for CP-net learning from choice data, Khoshkangini et al. (2018), bases their learning on probabilistic dependence in the data, rather than preferential dependence. This can lead to incorrectly oriented structures due to differences in symmetry. Consequently, their learned structure is closer to a Bayesian network than a CP-net model of the data.

We evaluated the performance of our learning procedure experimentally, using simulated data where the user's true preference order was known and representable by a CP-net. The learned CP-nets were over 95% consistent with the user's true preferences and their agreement scores with previously unseen (future) data sets were similar to the true CP-net's. This suggests that the learned CP-nets are a good model for the user's true preference structure. Unfortunately, the learning procedure is not yet efficient enough for practical use and cannot handle a large number of variables due to computational limitations. Thus, further work is required to address these limitations whilst preserving learning performance.

A discussion of our proposals for improvements and directions for future work on each of these techniques can be found in the discussion section of the relevant chapters (§2.6, §3.4, and §4.5).

Appendix A

Iteratively Updating Consistent Orderings

In general, CP-nets do not specify a total ordering over the outcomes. That is, there exist outcomes o_1 and o_2 such that $N \not\models o_1 \succ o_2$ and $N \not\models o_2 \succ o_1$ ($N \models o_1 \bowtie o_2$). Let $\succsim^C$ be a complete consistent ordering for N. As we do not know the user's preference between o_1 and o_2, $\succsim^C$ can order them in any manner without contradicting N. Thus, we can have $o_1 \succ^C o_2$, $o_2 \succ^C o_1$, or $o_1 \sim^C o_2$ – these incomparable outcomes have been forced into an arbitrary order by $\succsim^C$. Suppose we have $o_1 \succ^C o_2$ (or $o_1 \sim^C o_2$) and we learn that o_2 is preferred to o_1 by the user. Our ordering is no longer consistent with all of the known user preference information and, thus, needs updating to become consistent with $o_2 \succ\ o_1$. In this appendix, we present a method for updating any consistent ordering as new (consistent) preference information is learned. We also demonstrate how this process can be applied iteratively. That is, as more preference information is learned, the ordering can be repeatedly updated in order to be consistent with all known user preferences. These methods can be used on any consistent ordering (in particular, they can be used to update the rank ordering we introduced in §2.3.3 if new preferences are learned). These methods can also be used to update consistent orderings for any transitive preference structure (it is not restricted to CP-nets, as we shall show).

Being able to update a preference order given new information is important for any system that is continuously learning the user's preferences. For example, consider a news app. When a user first downloads the app, it may ask some general preference queries in order to present the most relevant articles. However, over time, the system can observe which specific articles the user reads. From this data, the system learns more specific preference information, which it can use to curate the news feed to suit the user's preferences more accurately. Alternatively, systems (such as Amazon, perhaps) may start by assuming some global preferences (for example, that a user prefers cheaper, more highly rated products) and then use the user's data over time to update this preference model to be more personalised.

In this appendix, we assume that the preference model we are updating is a consistent ordering and that the new preference information comes in the form of consistent pairwise preferences. Recall that consistent orderings contain all preference information encoded by CP-nets (by Theorem 2.12). However, consistent orderings are more directly applicable as they provide an explicit preference ordering of any set of outcomes. They also have many preference reasoning applications as we have seen in §2.3 and §2.4. Thus, it is more useful to be able to directly update the consistent ordering given new information, rather than updating the CP-net and then re-generating the ordering, particularly as producing a consistent ordering is intractable (see the remark in §2.3.5). Note that single pairwise preferences ($o_1 \succ o_2$) cannot be incorporated into a CP-net directly, they are simply added separately. They can be added into the preference graph by adding the relevant edge, $o_2 \rightarrow o_1$, but there is no CP-net that can express this new preference structure.

The assumption that the new preference information is a pairwise preference is not unreasonable; many general preference statements can be decomposed into a set of pairwise outcome preferences. However, the assumption that every new preference statement is consistent with the existing preferences (this is formally defined below) is unlikely to hold in real-world applications. This could be because a user's preferences change over time (and so contradict previous information) or because the user makes contradictory decisions, which is particularly likely if there are a large number of outcomes or the user does not have strong preferences.

How one updates a preference model given new, inconsistent information is likely to be context-dependent. For example, how quickly are user preferences likely to change? Should we immediately prioritise new preferences or do we wait for sufficient evidence before adjusting the model? If compromises are to be made, do we prioritise historic or new preference information? As systems are likely to receive such inconsistent information, we would like to create an update method that could be tuned as appropriate for different contexts. This is a direction for our future work. In this appendix, we address the problem of updating given new, consistent preference information.

Let us first formalise what we mean by consistent information. Let N be our CP-net and $\succsim^C$ be an ordering over the associated outcomes. Let o_1 and o_2 be associated outcomes. We say $\succsim^C$ is a consistent ordering if $N \vDash o \succ o' \implies$

$o \succ^C \ o'$. We say $o_1 \succ o_2$ is consistent with N if $N \nvDash o_2 \succ o_1$. Let G_N be the preference graph of N. The above definitions can also be formulated with regards to G_N. We say $\succsim^C$ is a consistent ordering if the following condition holds; if there is a directed path $o' \rightsquigarrow o$ in G_N, then $o \succ^C o'$. We say $o_1 \succ o_2$ is consistent with N if there is no directed path $o_1 \rightsquigarrow o_2$ in G_N. We shall now generalise these notions of consistency to an arbitrary graph, G, over the outcomes, representing user preference.

Definition A.1. Let G be a graph over the outcomes that represents user preference such that $o \succ o'$ if and only if there is a directed path $o' \rightsquigarrow o$ in G. Let $\succsim^C$ be any total preorder over the outcomes. The ordering $\succsim^C$ is *consistent* with G if the following condition holds; if there is a directed path $o' \rightsquigarrow o$ in G, then $o \succ^C o'$.

Let o_1 and o_2 be any two outcomes. The preference $o_1 \succ o_2$ is *consistent* with G if there is no path $o_1 \rightsquigarrow o_2$ in G.

In general, consistent here means 'does not contradict'. If G is G_N, then these definitions become the usual notions of consistency with a CP-net.

Note that for G to have a consistent ordering, $\succsim^C$, G must be acyclic. Total preorders cannot have $o \succ^C o$. If G is cyclic, then it contains a directed path $o \rightsquigarrow o$ for some outcome, o. As $\succsim^C$ is consistent, we have $o \succ^C o$, a contradiction. Thus, if G has a consistent ordering, then it is acyclic.

Let N be a CP-net, let $\succsim^{C_0}$ be a consistent ordering (an ordering consistent with G_N), and suppose we learn the preference $o_1 \succ o_2$, which is consistent with N (G_N). We want to update $\succsim^{C_0}$ to be consistent with all of the current preference information – N and $o_1 \succ o_2$. We know that $o_1 \succ o_2$ is consistent with N so $N \nvDash \ o_2 \succ o_1$. Thus, $N \vDash o_1 \succ o_2$ or $N \vDash o_1 \bowtie o_2$. If it is the former, then $o_1 \succ o_2$ is not new information and $\succsim^{C_0}$ is already consistent with all known preference information. If $N \vDash o_1 \bowtie o_2$, then there is no directed path between o_1 and o_2 in G_N. By adding $o_1 \succ o_2$, we add more than one pairwise preference to N. Suppose we have $N \vDash a \succ o_1$ and $N \vDash o_2 \succ b$, but $N \vDash a \bowtie b$. Thus, we have paths $b \rightsquigarrow o_2$ and $o_1 \rightsquigarrow a$ in G_N, but a and b are not connected. By adding $o_1 \succ o_2$, we get an edge $o_2 \rightarrow o_1$ and so there is now a path $b \rightsquigarrow a$ in G_N. So by adding $o_1 \succ \ o_2$ we also got $a \succ b$. Thus, being consistent with all known preferences is not as simple as being consistent with N and $o_1 \succ o_2$ separately, we need an order consistent with their combination. This new preference structure can be obtained from G_N by adding the edge $o_2 \rightarrow o_1$. We call this new graph, which represents all current preference information, G_1. In the case where $N \vDash o_1 \succ o_2$, we have $G_1 = G_N$ as no new information is gained. We want to update $\succsim^{C_0}$ to an ordering that is consistent with G_1.

Note that, as for CP-nets, orderings consistent with G_1 (or any G in general) are all equally 'good'. From the given information, we cannot say that one ordering is more likely to be the user's true preference than another. Thus, we cannot do better than obtaining any ordering consistent with G_1 (or G in general).

Proposition A.2. *Let G be any acyclic graph representing preference. Let $o_1 \succ o_2$ be any preference consistent with G. Let G' be the graph obtained from G by adding the edge $o_2 \rightarrow o_1$. Let $\succsim^C$ be an ordering over the outcomes that is consistent with G and such that $o_1 \succ^C o_2$. Then $\succsim^C$ is consistent with G' also.*

Proof. In order to show $\succsim^C$ is consistent with G', we need to show that, for any directed path $o' \rightsquigarrow o$ in G', we have $o \succ^C o'$. Let $o' \rightsquigarrow o$ be any directed path in G'. If $o' \rightsquigarrow o$ does not contain the edge $o_2 \rightarrow o_1$, then, by definition of G', this path is also in G. As $\succsim^C$ is consistent with G, we must therefore have $o \succ^C o'$.

Now suppose $o' \rightsquigarrow o$ does contain the edge $o_2 \rightarrow o_1$. Recall that G is acyclic. As $o_1 \succ o_2$ is consistent with G, there is no directed path $o_1 \rightsquigarrow o_2$. Thus, adding the edge $o_2 \rightarrow o_1$ to G does not create cycles. Hence, G' is also acyclic. Thus, $o' \rightsquigarrow o$ in G' can only contain the edge $o_2 \rightarrow o_1$ at most once. Let us decompose $o' \rightsquigarrow o$ into $o' \rightsquigarrow o_2 \rightarrow o_1 \rightsquigarrow o$. Assuming $o' \rightsquigarrow o_2$ and $o_1 \rightsquigarrow o$ are non-trivial paths, they cannot contain the edge $o_2 \rightarrow o_1$ and, thus, (by definition of G') they are also in G. As $\succsim^C$ is consistent with G, we have $o_2 \succ^C o'$ and $o \succ^C o_1$. We also have $o_1 \succ^C o_2$ and so, by transitivity, we have $o \succ^C o'$, as we wanted. If $o' \rightsquigarrow o_2$ is a trivial path, then $o' = o_2$ and so $o' \rightsquigarrow o$ is $o_2 \rightarrow o_1 \rightsquigarrow o$. By the same argument as above, $o_1 \rightsquigarrow o$ is in G and so $o \succ^C o_1$. Thus, as $o_1 \succ^C o_2$, we have $o \succ^C o_2$ by transitivity, which here means $o \succ^C o'$. A similar argument is used if either $o_1 \rightsquigarrow o$ is a trivial path or both $o' \rightsquigarrow o_2$ and $o_1 \rightsquigarrow o$ are trivial paths. Thus, we have shown $o \succ^C o'$ and so $\succsim^C$ is consistent with G'. □

Thus, in order to update $\succsim^{C_0}$ to be consistent with G_1, it is sufficient to obtain $\succsim^{C_1}$ consistent with G_N such that $o_1 \succ^{C_1} o_2$. If $N \vDash o_1 \succ o_2$, then we already have $o_1 \succ^{C_0} o_2$ and so $\succsim^{C_1} = \succsim^{C_0}$ as no new information was learned. Now suppose $N \vDash o_1 \bowtie o_2$. If $o_1 \succ^{C_0} o_2$, then it is already consistent with G_1 and, again, no update is required. If $o_2 \succ^{C_0} o_1$ or $o_1 \sim^{C_0} o_2$, then $\succsim^{C_0}$ is no longer adequate as it does not reflect the preference $o_1 \succ o_2$. We update $\succsim^{C_0}$ to an ordering, $\succsim^{C_1}$, that is consistent with G_1 as follows.

If $o_1 \sim^{C_0} o_2$, then we change $\succsim^{C_0}$ by making o_1 preferred to all outcomes it is equivalent to in $\succsim^{C_0}$. Let $E = \{o \in \Omega | o \neq o_1 \wedge o_1 \sim^{C_0} o\}$. We obtain $\succsim^{C_1}$ from $\succsim^{C_0}$ by changing the relation $o_1 \sim^{C_0} o$ to $o_1 \succ^{C_1} o$ for every $o \in E$. This produces a new total preorder, $\succsim^{C_1}$, and preserves all strict $\succsim^{C_0}$ preferences. We cannot have $N \vDash o \succ o_1$ or $N \vDash o_1 \succ o$ for $o \in E$ as $o \sim^{C_0} o_1$ and $\succsim^{C_0}$ is consistent with N. Thus, if $N \models a \succ b$, then we know $\{a, b\} \neq \{o_1, o\}$ for any $o \in E$. As the relative positions of o_1 and E are the only changes, the preference between a and b must be the same in both $\succsim^{C_0}$ and $\succsim^{C_1}$. As $\succsim^{C_0}$ is consistent with N, we must have $a \succ^{C_0} b$ and so we also have $a \succ^{C_1} b$. Thus, as $N \models a \succ b \implies a \succ^{C_1} b$, we know $\succsim^{C_1}$ is consistent with N. As $o_2 \in E$, we also have $o_1 \succ^{C_1} o_2$, by definition. Thus, by Proposition A.2, $\succsim^{C_1}$ is consistent with G_1.

Let us now formalise how we visualise consistent orderings. We consider consistent orderings to be vertical lists of *levels*, where the top level contains the outcomes most preferred by $\succsim^C$. If $a \succ^C b$, then b is on a lower level than a. If several outcomes are equivalent ($\sim^C$), then they are all on the same level.

Example A.3. Suppose we have outcomes $\Omega = \{a, b, c, d, e, f, g, h\}$ and a consistent ordering $\succsim^C$:

$$a \succ^C b \sim^C c \succ^C d \succ^C e \sim^C f \sim^C g \succ^C h.$$

Then $\succsim^C$ is made up of five levels:

$$\begin{array}{c} a \\ \hline b \;\; c \\ \hline d \\ \hline e \;\; f \;\; g \\ \hline h \end{array}$$

We define level k to be the level that is k^{th} from the top – in the above example, we have levels one to five. By this definition, if o is on level k and o' is on level ℓ, with $k < \ell$, then $o \succ^C o'$. If o and o' are both on level k, then $o \sim^C o'$. The outcomes on each level have no specific order, but the levels themselves are uniquely defined. As orderings and levels uniquely define one another, we shall discuss them interchangeably.

Suppose o_1 is on level k. In the above case, $o_1 \sim^{C_0} o_2$, we removed o_1 from level k and moved it to a new level between levels $k - 1$ and k.

Suppose instead that $o_2 \succ^{C_0} o_1$. We now show that, under certain conditions that are simple to check, outcomes can swap levels without affecting consistency.

Proposition A.4. *Let $\succsim^{C_0}$ be any ordering consistent with acyclic graph G. Suppose $\succsim^{C_0}$ has ℓ levels. Let O_k denote the outcomes on level k of $\succsim^{C_0}$, for any $k \leq \ell$. Let $S = \{o \in O_{k+1} | \forall o' \in O_k,\ o \to o' \notin G\}$. Let $\succsim^{C_1}$ be the ordering obtained from $\succsim^{C_0}$ by moving some $o \in O_{k+1}$ up to level k. If level $k+1$ is now empty, the level is removed. Then $\succsim^{C_1}$ is also consistent with G if and only if $o \in S$.*

Proof. See Appendix E.12.

Definition A.5. Let $\succsim^C$ be an ordering consistent with graph G. Say $\succsim^C$ has ℓ levels. Let O_k denote the outcomes on level k of $\succsim^C$, for any $k \leq \ell$. We say that outcome $o' \in O_{k+1}$ is *improvable* if there are no outcomes, $o \in O_k$, such that the edge $o' \to o$ is in G.

Proposition A.6. *Let $\succsim^C$ be a consistent ordering for CP-net N. Say $\succsim^C$ has ℓ levels. Let O_k denote the outcomes on level k of $\succsim^C$, for any $k \leq \ell$. Then $o' \in O_{k+1}$ is improvable if and only if there are no outcomes, $o \in O_k$, such that $HD(o, o') = 1$. HD is Hamming distance, $HD(o, o') = |\{X \in V | o[X] \neq o'[X]\}|$.*

Proof. To show this, we prove that for $o' \in O_{k+1}$ and $o \in O_k$, the edge $o' \to o$ is in G_N if and only if $\text{HD}(o, o') = 1$.

If $\text{HD}(o, o') = 1$, then o and o' differ on the value of exactly one variable. They constitute a variable flip. Thus, by definition of the preference graph, there is an edge between o and o' in G_N. If $o \to o'$ is in G_N, then $N \vDash o' \succ o$ as this edge constitutes an IFS. Thus, as $\succsim^C$ is consistent with N, we must have $o' \succ^C o$. This is a contradiction as o' is on a lower level of $\succsim^C$ than o. Thus, the edge must be $o' \to o$. Thus, $\text{HD}(o, o') = 1$ implies that the edge $o' \to o$ is in G_N.

If the edge $o' \to o$ is in G_N, then, by definition of G_N, this edge constitutes a variable flip. That is, one variable change transforms o into o' (and vice versa). Thus, $\text{HD}(o, o') = 1$. □

By Proposition A.4, if $\succsim^C$ is consistent with G_N, then moving improvable outcomes up a level produces another consistent ordering of N. Note that, by Proposition A.6, we can check whether an outcome is improvable in this case in $O(n|O_k|)$ time. This proposition makes it possible to check whether an outcome is improvable from the consistent ordering directly. This will allow us to update any consistent ordering, given a consistent new preference, without consulting the CP-net.

Let us return to our ordering, $\succsim^{C_0}$, with $o_2 \succ^{C_0} o_1$. Let o_2 be on level k of $\succsim^{C_0}$. If level k of $\succsim^{C_0}$ has multiple outcomes, we start by moving o_2 to its own level above. This is now level k and the levels below all shift down by one. This is equivalent to what we did in the $o_1 \sim^{C_0} o_2$ case and the resulting ordering is consistent with G_N by the same argument. Let o_1 be on level ℓ now, $k < \ell$. In order to update $\succsim^{C_0}$ to be consistent with G_1, we perform the following procedure: for $i \in \{k+1, ..., \ell\}$, in increasing order, we perform the following outcome movements; all improvable outcomes on level i are moved up to level $i-1$. If any of these outcomes are still improvable, then they are moved up to level $i-2$. This continues until none of the outcomes are improvable or until the outcomes are moved into the original level k (the level containing o_2, which may not be in position k any more). If outcomes reach level k in this manner, they are then moved into their own level above level k. The ordering produced is $\succsim^{C_1}$. Intuitively, for each level i in order ($k < i \leq \ell$), we move all improvable outcomes in this level as far up the ordering as possible until they pass level k. An example of this process is illustrated below.

Example A.7. Consider a CP-net, N, with three variables, A, B, C, and no edges. Variable A is binary and variables B and C are tertiary. Let CPT(A) be $a_1 \succ a_2$. Let CPT(B) be $b_1 \succ b_2 \succ b_3$. Let CPT(C) be $c_1 \succ c_2 \succ c_3$. We will represent $o = a_i b_j c_k$ by the triple ijk.

The following levels correspond to a valid consistent ordering of N:

111
112 121 211
113 122 131 212 221
123 132 222
213 231
133 223 232
233

Suppose we now learn the user preference 213 ≻ 121. This is consistent with N as $N \models a_2b_1c_3 \bowtie a_1b_2c_1$. However, 121 is above 213 in our consistent ordering. Thus, our consistent ordering needs to be updated in order to be consistent with this new preference. To do this, we use the procedure described above. Outcome 121 is on level 2. As level 2 has multiple outcomes, we start by moving 121 to its own level directly above level 2. This means outcome 213 is now on level 6 (see the first level diagram in Figure A.1). Our update procedure dictates we run through levels 3 to 6 in order and move the improvable outcomes up the levels as far as possible until they pass level 2.

We start with level 3. Both of 112 and 211 have Hamming distance 2 from 121. Thus, by Proposition A.6, they are both improvable. We therefore move them up to level 2 (see the second level diagram in Figure A.1). As this removes all outcomes from level 3, the level is removed entirely. As outcomes 112 and 211 have made it to level 2, they are then moved up to their own level above level 2 (see the third level diagram in Figure A.1). Note that the level numbers have been been altered by this process; for example, 121 is no longer on level 2. However, to keep the procedure implementation clear, we will continue to use the level enumeration from the first level diagram below – 121 is on level 2 and we next move the improvable outcomes of levels 4 to 6 as far up as possible (until they pass level 2).

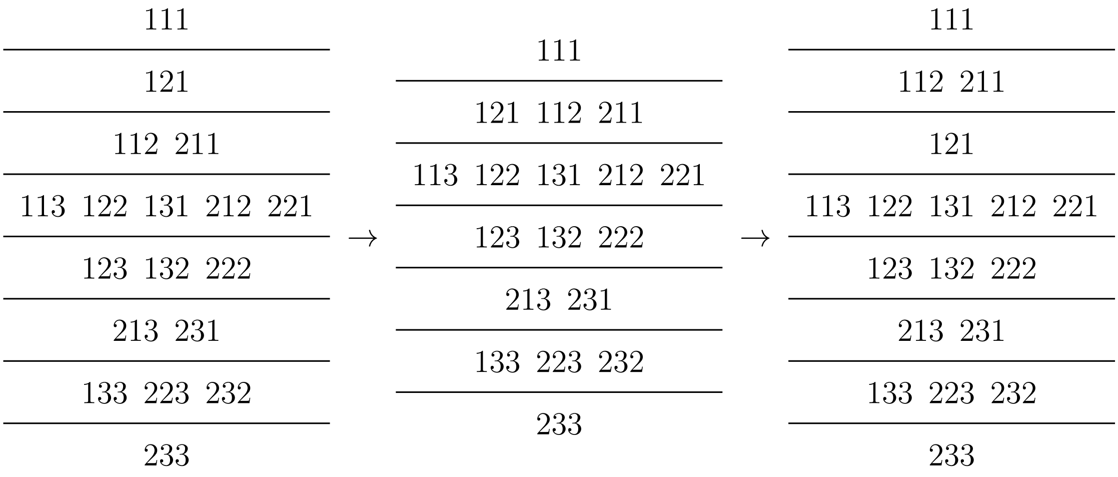


Figure A.1: Consistent Ordering Update Example Part 1

Let us now consider level 4. Outcomes 122, 131, and 221 all have Hamming distance 1 from 121 and, thus, are not improvable by Proposition A.6. Outcomes 113 and 212 have Hamming distance greater than 1 and are therefore improvable. These outcomes are thus moved up to level 2 (actually level 3). By our procedure, these outcomes then moved up to their own level above. The resulting levels are shown in the first diagram in Figure A.2.

Next consider level 5 (now actually level 6). Each outcome in this level has a Hamming distance of 1 from some outcome in the above level. We have $\mathrm{HD}(123, 122) = 1$, $\mathrm{HD}(132, 131) = 1$, and $\mathrm{HD}(222, 221) = 1$. Thus, none of these outcomes are improvable and so nothing happens. Finally, we consider level 6 (now level 7), which contains 213. Both 213 and 231 have a Hamming distance of greater than 1 from each of 123, 132, and 222. Thus, they are both improvable and move up to level 6. Level 7 is removed as it is now empty (see the second level diagram in Figure A.2). As $\mathrm{HD}(231, 221) = 1$, the outcome 231 is no longer improvable. However, 213 has a Hamming distance of more than 1 from each of 122, 131, 221, and 121 (all of levels 4 and 5). Thus, 213 is still improvable and moves up to level 5, then level 4. Level 4 is the 121 level, the previously named level 2. Thus, by our procedure, 213 is then moved to its own level above (see the third level diagram in Figure A.2) and the process terminates as we have now considered each of levels 3 – 6.

The resulting ordering has 213 above 121 and is therefore consistent with the new preference $213 \succ 121$. The resulting ordering is also a consistent ordering for N (as we shall prove below). Thus, by Proposition A.2, this ordering is consistent with the combination of N and the new preference, $213 \succ 121$, as we wanted.

Note that by using Proposition A.6, we did not need to consult N, we updated the consistent ordering directly.

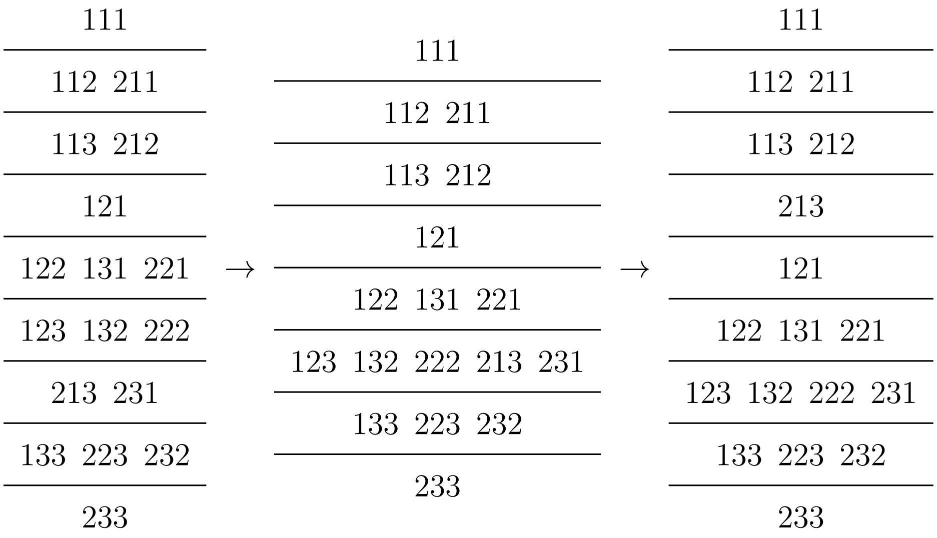


Figure A.2: Consistent Ordering Update Example Part 2

By the general procedure described above, the ordering $\succsim^{C_1}$ is obtained from $\succsim^{C_0}$ by using two types of action. First, moving improvable outcomes up a level (and removing empty levels if necessary). Proposition A.4 shows that this action preserves the consistency of an ordering. Second, moving a (proper) subset of the level k outcomes up to their own level between levels $k-1$ and k. The following lemma proves that this also preserves consistency. Thus, if we start with $\succsim^{C_0}$ that is consistent with G_N, then the resulting ordering, $\succsim^{C_1}$, must also be consistent with G_N. To prove that $\succsim^{C_1}$ is consistent with G_1, it remains only to prove that $o_1 \succ^{C_1} o_2$ (by Proposition A.2). We prove that $o_1 \succ^{C_1} o_2$ with the following theorem.

Lemma A.8. *Let $\succsim^{C_0}$ be an ordering consistent with graph G. Let $\succsim^{C_0}$ have ℓ levels. Suppose there is some k, $1 \leq k \leq \ell$, such that level k has more than one outcome. Let O_k denote the outcomes on level k. Let $R \subsetneq O_k$, $R \neq \varnothing$. The ordering $\succsim^{C_1}$ is obtained from $\succsim^{C_0}$ by moving the outcomes in R from level k up to a new level between levels $k-1$ and k. If $k=1$, then R is removed up from level 1 and becomes the new top level. All original levels $\geq k$ have now had their level index increased by one. The resulting ordering, $\succsim^{C_1}$, is also consistent with G.*

Proof. In constructing $\succsim^{C_1}$ from $\succsim^{C_0}$, the only outcomes that have moved are those in R. In particular, the only relative positions that have changed are between the outcomes in R and $O_k \backslash R$. The outcomes in R remain below outcomes in levels $< k$ and above outcomes in levels $> k$ (using the $\succsim^{C_0}$ level numbers). Suppose we have an outcome pair, (a, b), such that either $a \notin R$ or $b \notin O_k \backslash R$ (and vice versa). Then, as their relative positions have not changed, $a \succ^{C_0} b$ implies that $a \succ^{C_1} b$.

Consider any two outcomes, $o, o' \in O_k$. If there is a path $o \rightsquigarrow o'$ in G, then, as $\succsim^{C_0}$ is consistent with G, we have $o' \succ^{C_0} o$. This is a contradiction as o and o' are on the same level of $\succsim^{C_0}$. Thus, no two outcomes in O_k are connected by a directed path in G.

Let $o \rightsquigarrow o'$ be a directed path in G. As $\succsim^{C_0}$ is consistent with G, we have $o' \succ^{C_0} o$. As there are no directed paths between outcomes in O_k, we cannot have $o \in R$ and $o' \in O_k \backslash R$ (or vice versa). Thus, $o' \succ^{C_0} o$ implies $o' \succ^{C_1} o$ by the above argument. This shows that, for any directed path $o \rightsquigarrow o'$ in G, we have $o' \succ^{C_1} o$. That is, $\succsim^{C_1}$ is consistent with G, as we wanted to show. □

This result also proves (again) that our update procedure in the $o_1 \sim^{C_0} o_2$ case preserves consistency with N.

Theorem A.9. *Let G be a graph representing user preference and let $o_1 \succ o_2$ be a preference consistent with G. Suppose $\succsim^{C_0}$ is an ordering consistent with G such that $o_2 \succ^{C_0} o_1$. Let G_1 be obtained from G by adding the edge $o_2 \to o_1$ and let $\succsim^{C_1}$ be the ordering obtained from $\succsim^{C_0}$ by applying Algorithm 5. Then $\succsim^{C_1}$ is consistent with G_1.*

Proof. See Appendix E.13.

We now have a method in all possible cases for directly updating $\succsim^{C_0}$ to $\succsim^{C_1}$, which is consistent with G_1. Though in some cases this is a trivial update, $\succsim^{C_1} = \succsim^{C_0}$. In general, this update procedure is faster the closer o_1 and o_2 are in $\succsim^{C_0}$ (in the $o_2 \succ^{C_0} o_1$ case).

Suppose now that we learn i successively consistent preference statements. We start with N_G and learn the consistent preference $o_1 \succ o_1'$. This adds the edge $o_1' \to o_1$ to the preference graph, G_N, to give G_1. We then learn the second preference, $o_2 \succ o_2'$, which is consistent with G_1. This adds the edge $o_2' \to o_2$ to the preference structure, G_1, to give G_2. This continues until we learn the i^{th} preference, $o_i \succ o_i'$, which is consistent with G_{i-1}. This adds the edge $o_i' \to o_i$ to the preference structure, G_{i-1}, to give G_i. This graph, G_i, represents all current preference information (N and the i learned preferences combined) by the same argument used for G_1.

As we kept our updating procedure generalised, we can use it to iteratively update consistent orderings to be consistent with all current preference information. In the above scenario, we want an ordering consistent with G_i. We have already demonstrated how to update a consistent ordering for N, $\succsim^{C_0}$, into a consistent

Algorithm 5: Consistent Ordering Update

```
Input  : $G$ – Graph representing user preference
         $o_1 \succ o_2$ – New preference consistent with $G$
         $\succsim^C$ – Ordering consistent with $G$ such that $o_2 \succ^C o_1$
Output: $\succsim^{C^*}$ – Ordering consistent with $G$ such that $o_1 \succ^{C^*} o_2$

$k$ – The level of $\succsim^C$ containing $o_2$;
if Level $k$ contains $> 1$ outcome then
    $\succsim^{C^*}$ is obtained from $\succsim^C$ by moving $o_2$ to its own level between levels
      $k - 1$ and $k$;
end
else
    $\succsim^{C^*} = \succsim^C$;
end
For the remainder of the algorithm, let $i_j$ denote the level number of the
  current level $j$ of $\succsim^{C^*}$;
// $o_2$ is on level $k$ in $\succsim^{C^*}$
$\ell$ – The level of $\succsim^{C^*}$ containing $o_1$, $\ell > k$;
for $i \in \{i_{k+1}, i_{k+2}, ..., i_\ell\}$ do
    $I$ – The set of outcomes in level $i$ that are improvable;
    $J = I$;
    while $J \neq \varnothing$ and $J$ is on level $> i_k$ do
        Move $J$ up to the next highest level;
        // Any empty levels created are removed
        $J$ – The set of outcomes in $J$ that are still improvable;
    end
    $I' \subseteq I$ – Outcomes in $I$ that are now on level $i_k$;
    if $I' \neq \varnothing$ then
        Remove $I'$ from level $i_k$;
        Create a new level directly above level $i_k$ and populate it with $I'$;
    end
end
return $\succsim^{C^*}$;
```

ordering for G_1. Now suppose we have a consistent ordering for G_{i-1}, say $\succsim^{C_{i-1}}$, and we want to update this to an ordering, $\succsim^{C_i}$, consistent with G_i (that is, with the new preference $o_i \succ o'_i$).

If $o_i \succ^{C_{i-1}} o'_i$, then $\succsim^{C_{i-1}}$ is consistent with G_i by Proposition A.2. In this case, we set $\succsim^{C_i} = \succsim^{C_{i-1}}$. If $o_i \sim^{C_{i-1}} o'_i$, then o_i and o'_i are both on level k of $\succsim^{C_{i-1}}$, for some k. We obtain $\succsim^{C_i}$ from $\succsim^{C_{i-1}}$ by moving o_i from level k up to its own new level directly above. By Lemma A.8, $\succsim^{C_i}$ is still consistent with G_{i-1}. Clearly, $o_i \succ^{C_i} o'_i$ by construction. Thus, by Proposition A.2, $\succsim^{C_i}$ is consistent with G_i. Finally, if $o'_i \succ^{C_{i-1}} o_i$, then we can obtain $\succsim^{C_i}$ consistent with G_i by using the update procedure in Theorem A.9.

The procedure for updating $\succsim^{C_{i-1}}$ to $\succsim^{C_i}$ is the same as the procedure for updating $\succsim^{C_0}$ to $\succsim^{C_1}$, only now we use G_{i-1} rather than G_N. Thus, iteratively applying this general process, using the appropriate G, can update a consistent ordering given any number of newly learned preferences.

Recall that G_{i-1} is obtained from G_N by adding the $i-1$ edges, $o'_j \to o_j$ for $1 \leq j < i$. Therefore, we can simplify the definition of improvable for G_{i-1}, similarly to Proposition A.6 as follows. Suppose $\succsim^C$ is any ordering consistent with G_{i-1}. Let outcome o' be on level $k+1$ of $\succsim^C$. Then o' is improvable if and only if the edge $o' \to o$ is not in G_{i-1} for any outcome, o, on level k. That is, for any outcome, o, on level k, the edge $o' \to o$ cannot be in G_N or be the edge $o'_j \to o_j$ for any $1 \leq j < i$. Note that, as G_N is a sub-graph of G_{i-1}, $\succsim^C$ is also consistent with N. Thus, by Proposition A.6, for any o on level k, the edge $o' \to o$ is in G_N if and only if $\text{HD}(o, o') = 1$. Thus, o' is improvable if, for every o on level k, we have $\text{HD}(o, o') > 1$ and if $o' = o'_j$ (for some $1 \leq j < i$), then o_j is not on level k. Thus, we can again check improvability for G_{i-1} without consulting N. Thus, in all cases, $\succsim^{C_{i-1}}$ can be updated to $\succsim^{C_i}$ directly, without consulting N or G_N. However, the list of learned preferences (that were not already entailed when learned) may need to be consulted in the update procedure. Note that the complexity of checking whether o' is improvable is now $O(n|O_k| + ni)$.

We can now iteratively update any consistent ordering directly as additional (consistent) information about user preference is learned. An illustrative example of this procedure is given below.

Example A.10. Let us go back to Example A.7. We started with a CP-net, N, and a consistent ordering $\succsim^{C_0}$. We then learned the preference $213 \succ 121$. We updated $\succsim^{C_0}$ to $\succsim^{C_1}$ such that $213 \succ^{C_1} 121$. The current preference information is representable by G_1, which is obtained from N_G by adding the edge $121 \to 213$ (that is, $a_1b_2c_1 \to a_2b_1c_3$). By Theorem A.9, $\succsim^{C_1}$ is consistent with G_1.

Suppose we now learn the preference $122 \succ 231$. There is no directed path between 231 and 122 in G_1, so this preference is consistent with our current preference information, but it is new information (it is not already encoded by G_1). Let G_2 be obtained from G_1 by adding the edge $231 \rightarrow 122$. This graph now represents all current preference information. As $\succsim^{C_1}$ is consistent with G_1 and $122 \succ^{C_1} 231$, $\succsim^{C_1}$ is already consistent with G_2 (by Proposition A.2) and does not need updating. However, while $\succsim^{C_1}$ has not been changed, there are sections of this ordering that are now fixed that were previously changeable. For example, $122 \succ^{C_1} 231$ is now fixed as we have the preference $112 \succ 231$ in G_2. However, this order could have been reversed when only G_1 consistency was required. Note that this is unlikely to be the only ordering which became fixed by this update. For consistency we now refer to $\succsim^{C_1}$ as $\succsim^{C_2}$, even though the ordering has not changed.

Now suppose we learn the preference $133 \succ 221$. Again, there are no edges between 113 and 221 in G_2. Thus, this is a consistent preference and it adds to our known preferences. We update G_2 to G_3 by adding the edge $221 \rightarrow 133$. Currently, $221 \succ^{C_2} 133$, so we use the Theorem A.9 procedure to update $\succsim^{C_2}$ to be consistent with G_3. We start with $\succsim^{C_2}$, which is the same ordering we ended with in Example A.7. Outcome 221 is on level 6, but so are outcomes 122 and 131. Thus, our first step is to move outcome 211 to a new level above level 6. The resulting ordering is given by the first level diagram in Figure A.3. Outcome 221 is now on level 6 and outcome 133 is on level 9. The Theorem A.9 procedure moves the improvable outcomes of levels $7-9$ (in order) as far up the ordering as possible (until they pass level 6).

The ordering has 10 levels. Let O_k denote the outcomes on level k and let o' be on level $k+1$ (for some $k \leq 9$). By definition, o' is improvable if, for every $o \in O_k$, the edge $o' \rightarrow o$ is not in G_2. As we explained above, this is equivalent to the following condition. Our previously learned preferences are $213 \succ 121$ and $122 \succ 231$. Thus, o' is improvable if $\mathrm{HD}(o, o') > 1$ for all $o \in O_k$ and $o' = 121 \implies 213 \notin O_k$, $o' = 231 \implies 122 \notin O_k$.

We start by moving the improvable outcomes of level 7 as far up as possible. As $\mathrm{HD}(122, 221) = 2$ and $\mathrm{HD}(131, 221) = 2$, both 122 and 131 are improvable. Thus, we move them up to level 6 (level 7 is now empty and thus removed). As they have reached level 6, they are then moved up to their own level directly above. This new order is the second level diagram in Figure A.3. The levels now have different numbers (in particular, levels 6 and 7 have swapped), but we will continue to refer to the original level numbers for clarity of the procedure.

We now consider level 8. As $\mathrm{HD}(221, 222) = 1$ and $\mathrm{HD}(221, 231) = 1$, 222 and 231 are not improvable. Note that another condition for 231 to not be improvable would be if 122 had been in the above level. However, $\mathrm{HD}(221, 123) = 2$

and $\text{HD}(221, 132) = 3$, so 123 and 132 are improvable and are moved up a level. As they have reached (the original) level 6, they are then moved to their own level directly above. The resulting ordering is given by the third level diagram in Figure A.3.

Finally, we consider level 9, the level containing 133. As $\text{HD}(223, 222) = 1$ and $\text{HD}(232, 231) = 1$, outcomes 223 and 232 are not improvable. However, 133 has a Hamming distance of more than 1 from both 222 and 231 and is, thus, improvable. Therefore, 133 is moved up a level. As $\text{HD}(133, 221) = 3$, 133 remains improvable and is moved up again. The outcome 133 is now on the original level 6 (the 221 level). Thus, 133 is then moved up into its own level above and the process terminates. The resulting ordering, $\succsim^{C_3}$, is the fourth level diagram in Figure A.3.

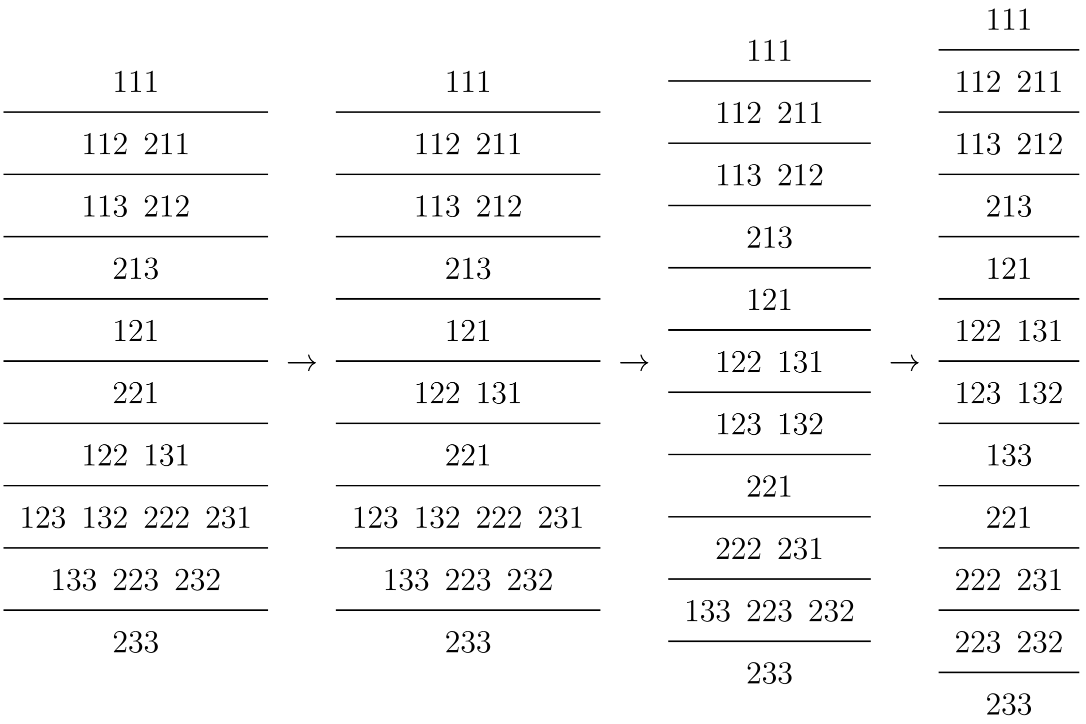


Figure A.3: Consistent Ordering Update Example Part 3

The resulting ordering, $\succsim^{C_3}$, is consistent with G_3 by Theorem A.9. However, consistency with N is also simple to check by hand as it is a simple CP-net. Notice that $213 \succ^{C_3} 121$, $122 \succ^{C_3} 231$, and $133 \succ^{C_3} 221$. Thus, we can see that $\succsim^{C_3}$ is also consistent with all of the learned preference information. Thus, $\succsim^{C_3}$ is consistent with G_3 by Proposition A.2 (applied repeatedly). That is, it is consistent with N, the learned preferences, and their transitive closure.

Note that the preference $112 \succ 121$ is encoded by G_3 as it contains a directed path $121 \rightsquigarrow 112$. Thus, if we went on to 'learn' this preference, no update is necessary. We do not need to add an edge to G_3 or update $\succsim^{C_3}$ and no additional parts of the ordering become fixed. Further, when we next update $\succsim^{C_3}$, we do not need to take $112 \succ 121$ into account when determining whether an outcome is improvable as it is not additional information. However, checking wither G_3 contains such a path is harder than dominance testing the original CP-net, N. Thus, unless we are already checking each learned preference for consistency, it is likely to be more efficient to treat such a preference as new information that $\succsim^{C_3}$ is already consistent with (as we did for $122 \succ 231$), rather than determining whether such a path exists.

Notice that the update procedure did not require us to consult the CP-net or graphs G_i. We only need the ordering and the list of previously learned preferences to perform the update. This is under the assumption that learned preferences are consistent and so we do not need to first check their consistency with G_i.

Note that, as more information is learned and we update our ordering correspondingly, more aspects of the ordering become fixed. Thus, by updating our ordering, we are moving towards to a fully specified, fixed preference order. This means that we have a total linear order and so all levels have size one – there are no equivalences. Note that most learned preferences fix multiple aspects of the ordering. Thus, achieving a fixed order will not require learning the preference of every incomparable pair individually. As the ordering gets increasingly fixed, future updates are less likely to change large sections of the ordering and, thus, will be more efficient to perform. The ordering also becomes a more reliable representation of the user's true preference order as more preferences are learned. Once it becomes completely fixed, their full preference order is known and no further updating is required. At this point, all preference reasoning tasks become fairly trivial.

If we start with a strict ordering (no equivalence statements), then we can ensure that it remains strict by swapping levels instead of adding improvable outcomes to the above level. In this case, swapping preserves consistency by Proposition A.4 and Lemma A.8. To prove that using swaps successfully updates $\succ^C$ is done by an almost identical proof to Theorem A.9, with some additional applications of Lemma A.8. Alternatively, we could 'flatten' the updated ordering into a strict order by arbitrarily ordering the outcomes on any level of size > 1. This preserves consistency by Lemma A.8. The latter could be done in any case where a strict ordering is required.

We have now introduced a method for iteratively updating any CP-net consistent ordering directly (without consulting N or G_N) as new (consistent) preferences are learned. This method can be used to update any consistent ordering, in particular the rank ordering we constructed in §2.3.3. As we defined this method in general terms, it can be used to update consistent orderings of any transitive preference structure (it is not restricted to CP-nets).

Appendix B

Outcome Rank Calculation Algorithm Details

In this appendix, we give the additional details necessary to understand and implement Algorithm 1, given in §2.3.5. We describe how CP-nets and outcomes should be formatted as inputs to Algorithm 1. We also explain how Algorithm 1 works and why it is correct. Further, we give Algorithms 6 and 7 for calculating ancestor sets and descendent paths, respectively, and we explain why they are correct. These algorithms are both called by Algorithm 1.

B.1 Rank Calculation Algorithm Input Formats

In this section, we give the input formats of CP-nets and outcomes for Algorithm 1. For this section, we assume that we have a CP-net N, over a set of variables, $V = \{X_1, ..., X_n\}$, which are in a topological order with respect to the structure of N. Further, we assume that $\text{Dom}(X_i) = \{x_i^1, ..., x_i^{n_i}\}$.

CP-nets are input to Algorithm 1 as a pair, $N = (A, CPT)$. The first entry, A, is the adjacency matrix of the structure of N, as described in §2.3.5.

Example B.1. The CP-net given in Example 1.2 has the following adjacency matrix:

$$\begin{array}{c c} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

The second entry in the pair is the set of CPTs associated with N. We input CPT as a list of the CPTs so, for any $1 \leq i \leq n$, we have $CPT[i] = \text{CPT}(X_i)$.

Let $\mathrm{Pa}(X_i) = \{X_{\beta_1}, ..., X_{\beta_\ell}\}$ $(\beta_1 < \beta_2 \cdots < \beta_\ell)$.

Let $\mathbf{u}$ be an assignment of values to $\mathrm{Pa}(X_i)$, $\mathbf{u} = x_{\beta_1}^{\alpha_1} \cdots x_{\beta_\ell}^{\alpha_\ell}$. Then $\mathbf{u}$ is a $|\mathrm{Pa}(X_i)|$-tuple in $\mathrm{Dom}(\mathrm{Pa}(X_i))$.

$\mathrm{CPT}(X_i)$ is input as a multi-dimensional array such that $\mathrm{CPT}(X_i)[\alpha_1, ..., \alpha_\ell]$ is a $|\mathrm{Dom}(X_i)|$-tuple, σ.

For all $1 \leq k \leq |\mathrm{Dom}(X_i)|$, $\sigma[k]$ is the position of preference of $X_i = x_i^k$ according to the CPTs, given that $\mathrm{Pa}(X_i) = \mathbf{u}$ ($\sigma[k] = 1$ if x_i^k is the most preferred value and so on).

Example B.2. For the CP-net given in Example 1.2, recall that $\mathrm{CPT}(C)$ is as follows:

ab	$c \succ \bar{c} \succ \bar{\bar{c}}$
$a\bar{b}$	$\bar{c} \succ \bar{\bar{c}} \succ c$
$\bar{a}\bar{b}$	$\bar{\bar{c}} \succ \bar{c} \succ c$
$\bar{a}b$	$\bar{\bar{c}} \succ c \succ \bar{c}$

In this example, $V = \{A, B, C, D\}$ (note that B, A, C, D is also a valid topological ordering, we use A, B, C, D for ease) and so

$$CPT = [\mathrm{CPT}(A), \mathrm{CPT}(B), \mathrm{CPT}(C), \mathrm{CPT}(D)].$$

We have $X_1 = A, X_2 = B, X_3 = C$, and $\mathrm{Dom}(A) = \{a, \bar{a}\}$, $\mathrm{Dom}(B) = \{b, \bar{b}\}$, $\mathrm{Dom}(C) = \{c, \bar{c}, \bar{\bar{c}}\}$. Thus, $x_1^1 = a, x_1^2 = \bar{a}$, and $x_2^1 = b, x_2^2 = \bar{b}$, and $x_3^1 = c, x_3^2 = \bar{c}$, $x_3^3 = \bar{\bar{c}}$. Also, $\mathrm{Pa}(C) = \{A, B\}$, so we would input $\mathrm{CPT}(C)$ ($CPT[3]$) as the following array:

	$[\cdot, 1]$	$[\cdot, 2]$
$[1, \cdot]$	$(1, 2, 3)$	$(3, 1, 2)$
$[2, \cdot]$	$(2, 3, 1)$	$(3, 2, 1)$

This dictates, for example, that $\mathrm{CPT}(C)[2, 1] = CPT[3][2, 1] = (2, 3, 1)$. In this entry, we input the user's preference over $\mathrm{Dom}(C)$ under $X_1 = x_1^2$ and $X_2 = x_2^1$, that is, $A = \bar{a}$ and $B = b$. We know that, in this case, we have $\bar{\bar{c}} \succ c \succ \bar{c}$, so $x_3^1 = c$ is in preference position 2, $x_3^2 = \bar{c}$ is in preference position 3, and $x_3^3 = \bar{\bar{c}}$ is in preference position 1. Hence $\mathrm{CPT}(C)[2, 1] = (2, 3, 1)$.

Note that, from this input, $CPT[3]$, we can clearly extract $|\mathrm{Dom}(C)|$ by looking at the length of the tuples in the array. To keep Algorithm 1 in §2.3.5 as readable as possible, we assume that, given $1 \leq i \leq n$, we can extract $|\mathrm{Dom}(X_i)|$ from the CPTs input, rather than putting the details of how this is achieved.

An outcome, o, should be input as a $|V|$-tuple in $\{1, ..., n_1\} \times \cdots \times \{1, ..., n_n\}$ (recall that $n_i = |\mathrm{Dom}(X_i)|$). If X_i takes value x_i^k and X_j takes value x_j^ℓ in outcome o, then $o[i] = k$ and $o[\{i,j\}] = (k,\ell)$. For our running example, consider the outcome $o = \bar{a}b\bar{\bar{c}}d$, we can rewrite this as $o = x_1^2x_2^1x_3^3x_4^1$ and we would input o as the tuple $(2,1,3,1)$. In o, B takes value b, that is, X_2 takes value x_2^1, and so $o[2] = 1$. Similarly, X_3 takes the value x_3^3 (C takes value $\bar{\bar{c}}$) so $o[3] = 3$.

B.2 Correctness of Rank Calculation Algorithm

In this section, we give the details of how Algorithm 1 works and why it is correct.

Algorithm 1 takes the CP-net, N, and some associated outcome, o, and outputs the rank of this outcome, $r(o)$. It calculates $r(o)$ by setting the value of $r(o)$ to 0 (step **1**) and successively adding the edge weights of the root-to-leaf path in W that corresponds to o (steps **2-11**). The weight attached to the edge indicating the value taken by X_i in o is given by Equation 2.3 in §2.3.2.

The algorithm calculates the edge weight given by Equation 2.3 for each X_i in several steps, and then adds it to the $r(o)$ term. The leftmost product term in Equation 2.3 is calculated in two steps (**3-4**). First, calling Algorithm 6 to obtain $\mathrm{Anc}(X_i)$, and then forming the product of the inverses of the domain sizes of all $Y \in \mathrm{Anc}(X_i)$ (an explicit explanation of how to obtain domain sizes can be found in Appendix B.1). We then call Algorithm 7 (step **5**) to obtain the number of descendent paths of X_i, d_{X_i}, in order to calculate the central product term in Equation 2.3.

Extracting the rightmost product term in Equation 2.3 from N and o is slightly more convoluted. The parent set of X_i, $\mathrm{Pa}(X_i)$, is the set of variables Y such that there is an edge $Y \to X_i$ in the structure of N. We can obtain this set directly from the adjacency matrix (step **6**). We then find the values taken by $\mathrm{Pa}(X_i)$ in o by extracting the appropriate entries of o, we call this assignment to the parent variables $\mathbf{u}$ (step **7**). So $\mathbf{u}$ is a $|\mathrm{Pa}(X_i)|$-tuple in $\mathrm{Dom}(\mathrm{Pa}(X_i))$. Next, we can find the user's order of preference over $\mathrm{Dom}(X_i)$ under $\mathrm{Pa}(X_i) = \mathbf{u}$ by extracting the appropriate entry of the $\mathrm{CPT}(X_i)$ array input, $\mathrm{CPT}(X_i)[\mathbf{u}]$ (step **8**).

The k in the rightmost product of Equation 2.3 is the position of preference of the value taken by X_i in o in the preference order we have just obtained. Thus, we can find k by extracting the element of this order that indicates the position of preference of the value taken by X_i in o (this is $o[i]$) (step **9**). This k is the position of preference of the choice $X_i = o[X_i]$, given that $\mathrm{Pa}(X_i) = \mathbf{u}$. Now that we have k, we can calculate the rightmost term in Equation 2.3 using $n_{X_i} = |\mathrm{Dom}(X_i)|$ (step **10**).

Finally, we form the whole term given in Equation 2.3 and add it to the $r(o)$ term (step **11**). Repeating this for every $X_i \in V$ gives us the rank of o by definition. At this point, Algorithm 1 exits its 'for' loop and outputs $r(o)$ correctly (step **12**), as we wanted.

B.3 Ancestor and Descendent Path Calculation Algorithms

In this section, we give the algorithms for calculating ancestor sets and descendent paths and explain how they work and why they are correct. These are called by Algorithm 1 for calculating outcome ranks. These algorithms assume that the variables are enumerated $\{X_1, ..., X_n\}$ and that the adjacency matrix is configured in this order. That is, $A_{i,j} = 1$ if and only if there is an edge $X_i \rightarrow X_j$ in the structure of N.

```
Algorithm 6: Ancestor Set Calculation
  Input  : $1 \leq i \leq |V|$ – Index of the variable of interest
           $A$ – Adjacency matrix of the structure of $N$
  Output: Anc($X_i$) – Set of ancestors of $X_i$ in the structure of $N$
  Paths = $\mathbf{0}_{|V|}$;                  // $0_{|V|}$ is the zero $|V|$-tuple
  a = $A_{.,i}$;                               // $A_{.,i}$ is the $i^{th}$ column of $A$
  while $sum(\boldsymbol{a}) > 0$ do
      Paths = Paths + a;
      a = $A$a;
  end
  $Anc = \{X_j | \mathbf{Paths}[j] \neq 0\}$; // The set of variables with a non-zero
    entry in Paths
  return $Anc$;
```

Algorithm 6 takes an integer, i ($1 \leq i \leq |V|$, indicating which variable's ancestor set we are interested in), and the adjacency matrix, A, and outputs Anc(X_i). For any $X \in V$, the following statements are equivalent.

$$\begin{aligned} Y \in \text{Anc}(X) &\iff \exists \text{ directed } Y \rightsquigarrow X \text{ path} \\ &\iff (A^k)_{Y,X} \neq 0 \text{ for some } 1 \leq k \leq |V| - 1 \end{aligned}$$

because $(A^k)_{i,j} = \#$ directed $X_i \rightsquigarrow X_j$ paths of length k in N. Also, no path in N can be of length greater than $|V| - 1$ as there are $|V|$ variables (vertices) in the (acyclic) structure. Thus, Algorithm 6 calculates $\text{Anc}(X_i)$ by summing, component-wise, the i^{th} columns of A^k for $k = 1, ..., |V| - 1$. By the above equivalences, $\text{Anc}(X_i)$ are the variables whose corresponding entry is non-zero.

```
Algorithm 7: Descendent Path Calculation
  Input   : $1 \leq i \leq |V|$ – Index of the variable of interest
            $A$ – Adjacency matrix of the structure of $N$
  Output: $d_{X_i}$ – Number of descendent paths of $X_i$ in the structure of $N$
  $\mathbf{a} = A_{i,\cdot}$;                      // $A_{i,\cdot}$ is the $i^{th}$ row of $A$
  $d = 0$;
  while $sum(\boldsymbol{a}) > 0$ do
      $d = d + sum(\mathbf{a})$;
      $\mathbf{a} = \mathbf{a}A$;
  end
  return $d$;
```

Algorithm 7 takes an integer, i ($1 \leq i \leq |V|$, indicating which variable's descendent paths we are interested in), and the adjacency matrix, A, and outputs d_{X_i}. As $(A^k)_{ij} = \#$ directed $X_i \rightsquigarrow X_j$ paths of length k in N, the following result holds for any variable, $X_i \in V$:

$$\begin{aligned} d_{X_i} &= \sum_{j=1}^{|V|} \#\text{directed paths } X_i \rightsquigarrow X_j \\ &= \sum_{j=1}^{|V|} \sum_{k=1}^{|V|-1} \#\text{directed paths } X_i \rightsquigarrow X_j \text{ of length } k \\ &= \sum_{j=1}^{|V|} \sum_{k=1}^{|V|-1} (A^k)_{i,j} = \sum_{k=1}^{|V|-1} \sum_{j=1}^{|V|} (A^k)_{i,j}. \end{aligned}$$

Therefore, Algorithm 7 calculates d_{X_i} by summing the entries of the i^{th} rows of A^k for $k = 1, ..., |V| - 1$.

Appendix C

Pruning Performance Experiments – Details and Further Results

In this appendix, we give some additional details and results from our experiments in §2.4.2. These experiments evaluated the performance of various methods of pruning the search tree in order to improve dominance testing efficiency. We first describe our method for randomly generating CP-nets (in order to simulate dominance queries for testing). We then examine the performance of the initial conditions of the various methods and consider their accuracy in predicting dominance query result. Finally, we give the performance results of each pruning measure when tested with all possible leaf prioritisation heuristics. These results illustrate the effect of the leaf prioritisation choice on performance. The results given in §2.4.2 show only the results where the various methods use their optimal prioritisation techniques.

C.1 CP-Net Generator

In this section, we describe how the CP-nets were randomly generated for the Chapter 2 experiments. In order to randomly generate an acyclic CP-net over n variables, we need the domain sizes, a directed acyclic graph (DAG) for the structure, and the CPTs.

Our generation method takes inspiration from Allen et al. (2017a), who illustrate how CP-nets can be generated uniformly at random given certain input parameters. However, the number of valid CP-nets over a given set of n variables gets incredibly large as n increases (and as the domain sizes increase). Thus, the number of possible CP-nets that can result from each choice in their procedure is very large. In fact, the associated probability weights become too large for our computational resources to handle. Because of this, we were unable to use their uniform generation method, meaning that our CP-nets were generated randomly but not uniformly. However, our CP-net generation allows CP-net variables to have distinct domain sizes, whereas Allen et al. (2017a) consider only CP-nets where all variables have the same domain size. In fact, all valid acyclic CP-nets can be produced by our generator.

Remark. As we do not have real-world CP-net data, it is not known what a realistic distribution of CP-nets is. That is, we do not know what properties a CP-net distribution should have to make our experimental results accurately reflect real world performance. Thus, while a uniform distribution gives a fair representation of all possibilities, we do not know if this would be an appropriate choice.

Suppose that we have n variables and the maximum domain size is M. We enumerate the variables $\{X_1, X_2, ..., X_n\}$ and assign each variable a domain size between 2 and M by choosing a value in this range uniformly at random.

In order to randomly generate the DAG, we use the dag-codes used by Allen et al. (2017a). Given n nodes, dag-codes are defined as follows and are in one to one correspondence with the set of possible DAGs over these nodes.

Definition C.1. Let n be a positive integer. A *dag-code* is a list $A = \langle A_1, ..., A_{n-1}\rangle$ of subsets $A_i \subseteq \{1, ..., n\}$ that satisfy the following condition; for every $1 \leq i \leq n - 1$,

$$\left|\bigcup_{k\leq i} A_k\right| \leq i.$$

Each dag-code corresponds to exactly one DAG over the n nodes. This is obtained by assuming the A_i sets to be the parent sets of $n-1$ of the variables. Note that, as the graph is acyclic, one variable (at least) has an empty parent set – this is the implicit final parent set. Allen et al. (2017a) give an explicit method of transforming a given dag-code into a DAG, which we use in our random generation procedure.

In order to randomly generate a dag-code, we randomly selected each A_i in turn, starting at A_1. Let $U = \bigcup_{k<i} A_k$. We repeatedly randomly select a subset $A_i \subseteq \{1, ..., n\}$ (by randomly selecting a size, $|A_i|$, from 1 to n each time and then performing uniform random sampling) until $|U \cup A_i| \leq i$ holds. This is then fixed as A_i and we repeat the process for A_{i+1}.

Once we have randomly generated a dag-code, we use the algorithm by Allen et al. (2017a) to transform this into an acyclic structure over our n variables. We now need to form the CPTs. As we have the structure, we know the parents of each variable (and their possible value assignments) so we only need to fill in the preferences order of each row. Suppose X_i has $|\mathrm{Dom}(X_i)| = k$. Each row of $\mathrm{CPT}(X_i)$ is a strict order over the k elements of $\mathrm{Dom}(X_i)$, which we can consider as a permutation of $\mathrm{Dom}(X_i)$. We fill each row of $\mathrm{CPT}(X_i)$ with a random permutation of $\mathrm{Dom}(X_i)$. However, the structure needs to be valid, that is, we need to make sure there are no degenerate parents. For any $Y \in \mathrm{Pa}(X_i)$, Y is a true parent of X_i if the preference over X_i is dependent on the value of Y. Thus, we check, for each $Y \in \mathrm{Pa}(X_i)$, whether there exists $\mathbf{u}_1, \mathbf{u}_2 \in \mathrm{Dom}(\mathrm{Pa}(X_i))$ that differ on the value of Y only, such that $\mathbf{u}_1$ and $\mathbf{u}_2$ result in different X_i preferences in $\mathrm{CPT}(X_i)$. If there is any $Y \in \mathrm{Pa}(X_i)$ for which this condition does not hold, then the CPT does not accurately reflect the dependency structure as X_i is not preferentially dependent on Y. In this case, we re-generate the CPT from scratch in the same way and check again. This continues until a non-degenerate CPT is obtained (note that there is always at least one such CPT). If X_i has no parents, any CPT is valid. We use the above process to populate all of the CPTs, resulting in a complete, randomly generated CP-net.

As we mentioned above, we cannot guarantee the CP-net distribution produced by this method. However, from what we have seen, it appears to favour sparser structures. Comparatively, using a uniform distribution is more likely to generate denser structures due to all of the CPT configuration possibilities. An exact analysis of the CP-net distribution produced by our generator is something we would like to pursue in our future work, as we discuss in §2.6.

C.2 Zero Outcomes Traversed Results

In §2.4.2, we experimentally evaluated the performance of seven different dominance testing functions by applying them to the same sets of dominance queries and recording outcomes traversed and time elapsed. These seven functions were all possible combinations of suffix fixing (Boutilier et al., 2004a), penalty pruning (Li et al., 2011a), and rank pruning (§2.4.1). Each combination has certain conditions that would result in a dominance query being immediately found false, in which case, the outcomes traversed would be recorded as zero (as discussed in §2.4.2). We shall refer to these as the *initial conditions* of the pruning combinations. Suppose we wish to answer the dominance query $N \vDash o \succ o'$, a summary of these initial conditions is given below.

Initial Condition	Pruning Combination
$o = o'$	All
$f(o') < 0$	All combinations including penalty pruning
$r(o) - r(o') < L_D(o, o')$	All combinations including rank pruning

In this section, we look at the proportion of queries from our §2.4.2 experiment that resulted in zero outcomes traversed (that is, that met one or more of the initial conditions) for each function (pruning combination). This proportion shows us how often a dominance testing function's initial conditions are strong enough to immediately answer the query. Further, by comparing these proportions to the proportion of queries that were false, we can evaluate how well a function's initial conditions can predict the outcome of a dominance query. If these predictions are reasonably accurate, then we can use them to answer dominance queries efficiently with good (though not perfect) accuracy.

First note that adding suffix fixing to a combination does not add any initial conditions. Thus, it is sufficient to evaluate these proportions only for the following four functions: rank pruning, penalty pruning, suffix fixing, and the combination of rank pruning and penalty pruning. Further, as initial conditions are assessed prior to constructing the tree, it does not matter what leaf prioritisation is used here.

In the case of binary CP-nets, for each of $3 \leq n \leq 19$, we tested all seven functions on a set of 1000 dominance queries in the §2.4.2 experiment. Thus, each function answered the same set of 17,000 dominance queries. Out of these queries, 13,703 (0.80606) of them were false. Note that, despite the random generation of queries, this is not close to 0.5. This is because there are three possibilities, either $N \vDash o \succ o'$, $N \vDash o' \succ o$, or $N \vDash o \bowtie o'$. For the dominance query '$N \vDash o \succ o'$?', only the first case makes the query true, the other two cases imply that the query is false. Further, the proportion of incomparable cases is likely to be greater for larger values of n. As the comparable cases must occur in equal proportions, this decreases both. In Table C.1, for each function, we give the proportion of the 17,000 queries that were determined to be false by the initial conditions (that is, were answered with zero outcomes traversed), Z_P, and the proportion of false queries that were identified correctly as false by these initial conditions, $Z_P/0.80606$.

The Z_P value for suffix fixing shows us the proportion of $o = o'$ cases. Thus, the initial rank condition determines $0.73995 = 0.75424 - 0.01429$ of the 17,000

	Rank	Penalty	Suffix Fixing	Rank + Penalty
Z_P	0.75424	0.58765	0.01429	0.75511
$Z_P/0.80606$	0.93571	0.72904	0.01773	0.93680

Table C.1: Zero Outcomes Traversed Proportions – Binary Case

queries to be false immediately and, similarly, the initial penalty condition determines 0.57336 of the queries to be false immediately. Clearly, the rank condition is much stronger than the penalty condition as it determines a greater number of queries to be false. Further, by looking at the Z_P value for rank and penalty pruning combined, we can see that utilising both conditions is only a slight improvement upon the rank condition alone. Thus, most cases identified as false by the penalty condition are also identified by the rank condition.

The $Z_P/0.80606$ values show us how many of the false dominance queries were detected by the initial conditions. Using rank pruning alone, over 93% of the false dominance queries were identified as false by the initial conditions. This suggests that our initial conditions could be used as fairly accurate predictor for the outcome of a dominance query. Any dominance query determined to be false by initial conditions must be false. Of those that do not meet any of the rank pruning initial conditions (that is, those we would 'predict' to be true), only $(0.80606 - Z_P)/(1 - Z_P) \times 100 = 24.576\%$ would actually be false (in these cases, o and o' are incomparable). This percentage would be slightly smaller if both the rank and penalty pruning initial conditions were used. In total, $1 - (0.80606 - Z_P) = 0.94818$ of dominance queries are successfully classified as either true or false by the initial rank conditions. This proportion would be slightly higher if the penalty condition is used as well. Thus, we could use these initial conditions, which are quick to check (polynomial in n), to predict dominance query outcomes with a reasonable level of accuracy.

For the multivalued case, we tested all seven functions on a set of 1000 queries for each $3 \leq n \leq 10$. In this case, we tested 8000 dominance queries and 6190 (0.77375) of these were false. Note that larger domains result in larger and more complex CP-nets and, thus, the proportion of incomparability is likely to be greater than the binary CP-nets. This explains why we have a similarly high proportion of false queries despite the fact we are using smaller n values in this case. Table C.2 gives the proportion of queries with zero outcomes traversed for each function and the proportion of false queries identified by the initial conditions for each function.

	Rank	Penalty	Suffix Fixing	Rank + Penalty
Z_P	0.66138	0.543	0.0055	0.6655
$Z_P/0.77375$	0.85477	0.70178	0.00711	0.86010

Table C.2: Zero Outcomes Traversed Proportions – Multivalued Case

These proportions show similar patterns to the binary case. The initial rank condition remains the strongest, determining the greatest number of queries to be false. Again, adding the penalty condition makes little improvement to this number. However, the proportions are smaller in general in this case; only 85% of false queries are identified by the initial conditions of rank pruning in this case. Thus, these initial conditions would be less accurate at predicting dominance query outcomes in this case. If we used the initial conditions of rank pruning as a predictor here, then $(0.77375 - Z_P)/(1 - Z_P) \times 100 = 33.185\%$ of queries predicted to be true would in fact be incomparable cases (false queries). In total, $1-(0.77375-Z_P) = 0.88763$ of dominance queries are successfully classified as either true or false by the initial rank conditions here. This proportion would be slightly higher if the penalty condition is used as well.

From the above proportions, we can see that including rank pruning in a dominance testing function results in a much larger proportion of dominance queries being answered immediately (by initial conditions). Thus, when using rank pruning, we are not required to construct a search tree at all for a large proportion of queries. The number of queries answered immediately by rank pruning initial conditions is greater than that answered by the initial conditions of penalty pruning, showing the rank conditions to be stronger. Further, the number of queries answered by their combination is only slightly greater than those answered by rank pruning initial conditions alone. This suggests that there are few queries answered by the initial penalty condition that are not already answered by the initial rank condition. We have also seen that the rank pruning initial conditions identify the majority of false queries. Thus, at least in the binary case, these initial conditions (which are quick to check) could be used as a reasonably accurate predictor of dominance query results. While using the penalty condition as well only results in a limited improvement, it is worth using both as they are simple conditions to check.

The relative performance of the rank and penalty conditions (and their combination) here is unsurprising as these conditions are equivalent to their pruning conditions. Thus, we see similar patterns as in the dominance testing outcomes traversed results. However, in the case of pruning, we found that the minor improvement resulting from using penalty pruning was not worth the implementation cost.

Remark. Note that checking the initial conditions (other than $o = o'$) is equivalent to answering an ordering query, as we discussed in §2.4.1. If the initial condition holds, then the dominance query is false and so $o' \succ o$ must be a consistent ordering. If the initial conditions do not hold, then we know that $N \not\models o' \succ o$ and so $o \succ o'$ is a consistent ordering (even if it is not an entailed preference). Thus, in this appendix we have assessed how accurately certain ordering query methods predict dominance query results. However, as we discussed in §2.4.1, performing ordering query tests in both directions can give more information. In particular, it can prove cases of incomparability. All of the misclassification above are incomparable cases labelled as 'true' – initial conditions as used above cannot predict incomparability, only true or false. Thus, our initial conditions may be improved as predictors by evaluating both 'directions' (that is, the initial conditions for both $N \models o \succ o'$ and $N \models o' \succ o$).

C.3 Leaf Prioritisation Comparison Results

In this section, we evaluate how the choice of leaf prioritisation heuristic affects the performance of the different dominance testing functions. Several prioritisation heuristics have been previously proposed but this is the first evaluation of the effect of this choice. As we discussed in §2.4.2, we only applied leaf heuristics that did not require additional calculations. Thus, suffix fixing used minimal depth only, penalty pruning used penalty prioritisation only, and so did the combination of penalty pruning with suffix fixing. Thus, we do not consider these pruning methods here as there was no variation in leaf prioritisation – their full results are given in §2.4.2.

Rank pruning was applied with both rank and rank + diff. prioritisation, as was rank pruning combined with suffix fixing. Rank pruning combined with penalty pruning was applied with rank, rank + diff., and penalty prioritisation, as was the combination of all three methods. The graphs below show the performance of each function applied with all possible prioritisation heuristics.

Each function has four sets of data, the same as the §2.4.2 results – outcomes traversed and time elapsed in both the binary and multivalued CP-net cases. Figures C.1 and C.2 give the binary and multivalued CP-net results for rank pruning, respectively. Similarly, the results for the combination of rank pruning and suffix fixing are given in Figures C.3 and C.4. The rank and penalty pruning combination results are given in Figures C.5 and C.6. Finally, the performance results for the combination of all three pruning measures are given in Figures C.7 and C.8. Each of Figures C.1 – C.8 have two graphs; plot (a) shows the outcomes traversed results and plot (b) shows the time elapsed results. In each plot, the shaded area gives the $\pm SE$ (standard error) interval for the method when using rank prioritisation (note that this is the function depicted in the §2.4.2 graphs). Each plot shows the results of the relevant pruning method using all possible leaf prioritisation heuristics. Some graphs also include the results of one or two other pruning schemas, in order to make it clear how these results relate to those given in the §2.4.2 plots. In these cases, the additional methods all use their optimal prioritisation heuristic (as they do in §2.4.2). All functions are given in the same colours as the §2.4.2 graphs. Different leaf prioritisation heuristics are distinguished by line type.

In all 16 graphs, all variations of the pruning method of interest lie comfortably within the standard error interval. This shows us that varying leaf prioritisation does not have a significant effect on either the effectiveness or efficiency of a pruning schema.

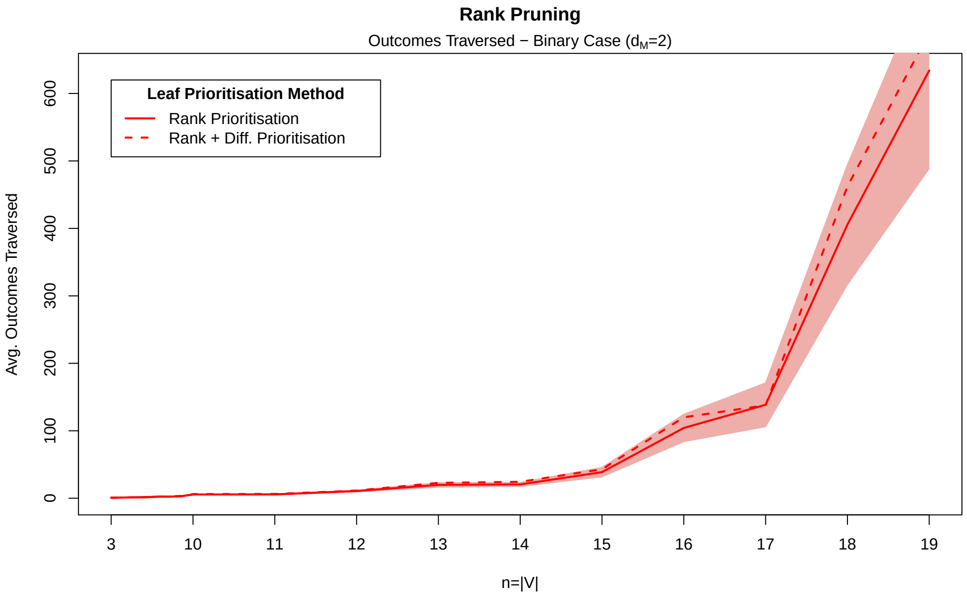
Despite this, the prioritisation choice can be the deciding factor in which pruning method performs better. As we can see in Figures C.5, C.6, C.7, and C.8 (particularly in the outcomes traversed results), changing the leaf prioritisation can change whether the relevant pruning method performs better or worse than another. In particular, in Figures C.7 and C.8, this decision changes which pruning combination is the most effective overall.

In the binary case (Figures C.1, C.3, C.5, and C.7), we see the following patterns in all graphs except for time elapsed by the combination of all three pruning methods. Rank prioritisation is always a better choice than penalty prioritisation. That is, at every data point, rank priority is faster and has less outcomes traversed on average than penalty prioritisation. In every graph, rank priority also performs consistently better than rank + diff. with the exception of at most one data point per graph, where rank + diff. may perform better by a much smaller margin. Thus, in all of these cases, the outcomes traversed and time elapsed results agree that rank priority is the optimal choice.

In the time elapsed results for the combination of all three pruning measures on binary CP-nets, penalty outperforms rank priority at four data points. This is only for smaller values of n and rank increasingly outperforms penalty priority as n gets larger. Also, in this case, rank + diff. outperforms rank at six data points, mostly for small n values and with less frequency as n increases. Thus, while the results are closer in this case, rank priority still performs best in the majority of cases and appears to be more consistently optimal as n increases. Therefore, the outcomes traversed and time elapsed results again agree that rank priority is the optimal choice.

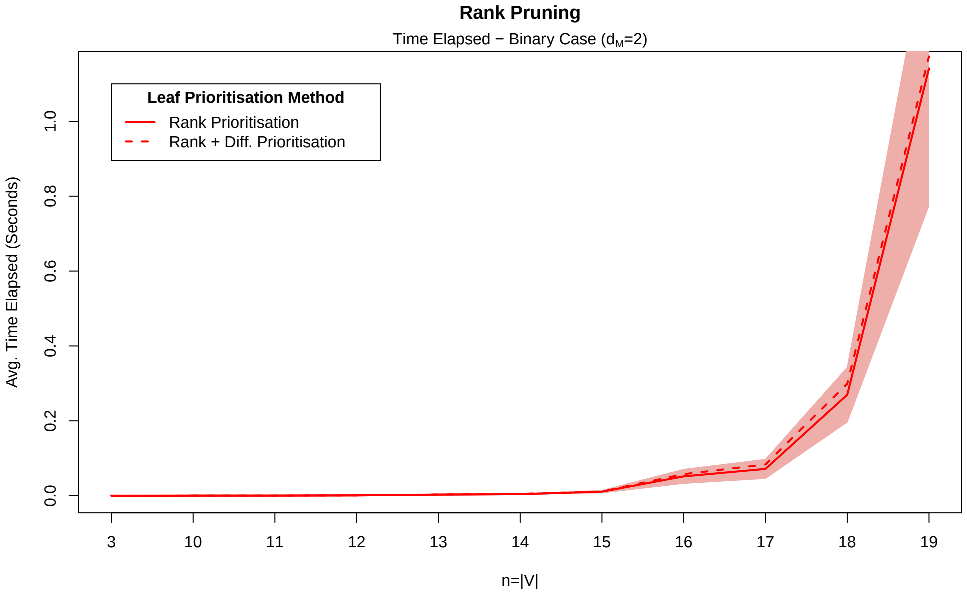
In the multivalued case (Figures C.2, C.4, C.6, and C.8), at every data point, rank priority results in a more effective pruning measure on average (that is, has fewer outcomes traversed on average) than penalty and rank + diff. prioritisation. Rank priority is also more efficient on average at every data point for rank pruning and the combination of rank pruning with suffix fixing. Thus, for these two pruning schemas, both measures indicate that rank priority is definitively the best choice. For rank and penalty pruning, rank priority is consistently more efficient than rank + diff. but is slower than penalty prioritisation at a single data point (by a relatively small margin). Thus, in general, rank priority is still the most efficient, as well as the most effective choice here. For the combination of all three pruning measures, penalty priority is faster than rank at two data points and rank + diff. is faster at one. However, rank priority is still the most efficient in the majority of cases. Thus, as rank priority is also the most effective (least outcomes traversed) choice, it is again the optimal choice of prioritisation heuristic.

In summary, the above results suggest that the choice of leaf prioritisation method does not make a significant difference to the performance (either outcomes traversed or time elapsed) of a given pruning method. However, the prioritisation choice can still affect performance enough to alter whether the given method is more or less effective and/or efficient than others. In particular, this choice can determine which pruning method is the most effective overall. In all cases, rank priority performs best (by both measures) in the majority of data points and, in general, the amount by which it outperforms the other choices increases with n. Thus, from these results, we have determined that our rank prioritisation is the optimal choice for all pruning methods for which priority was varied.



(a) Outcomes Traversed Results

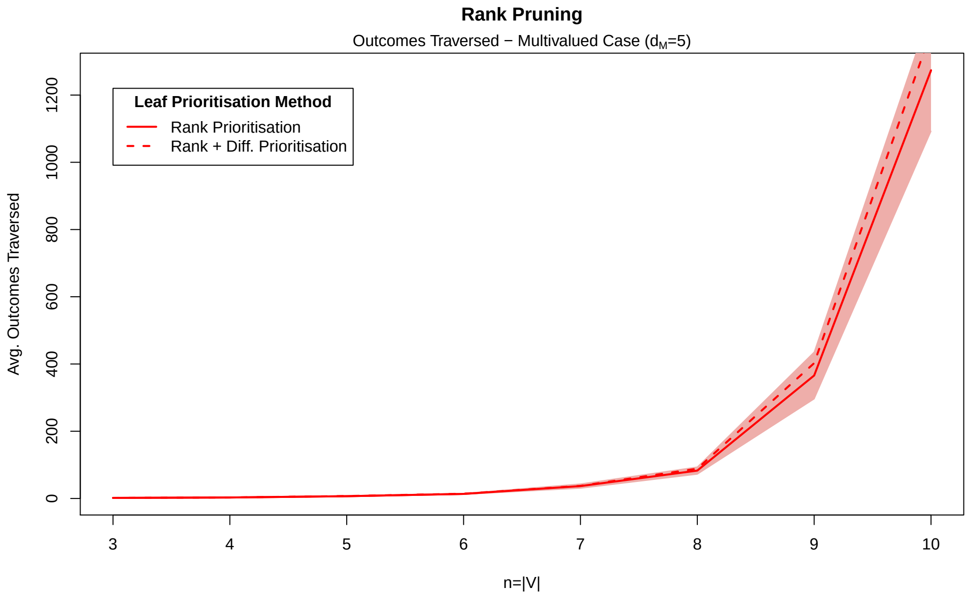
Note: n values between 3 and 10 are compressed in order to improve plot clarity for larger n values



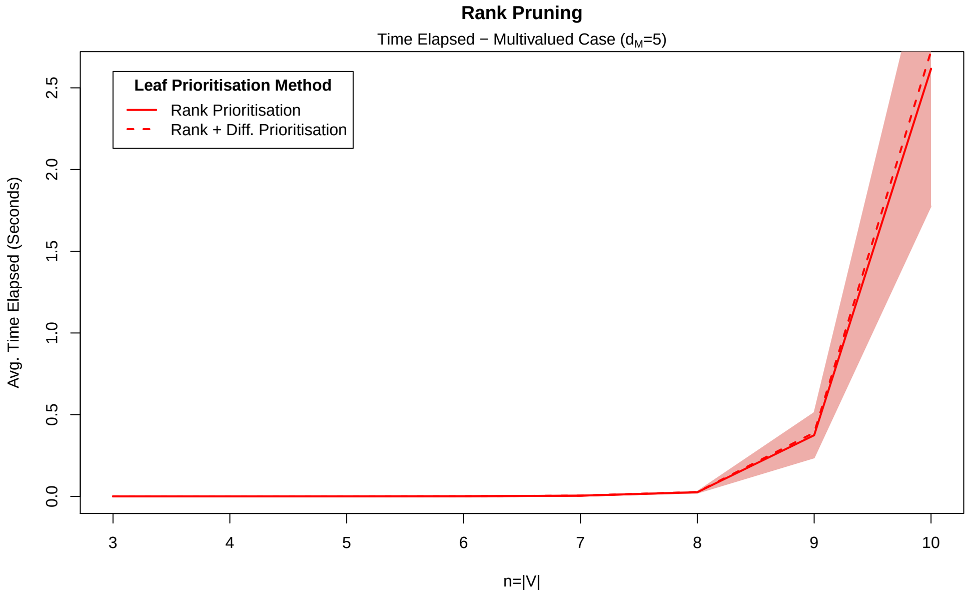
(b) Time Elapsed Results

Note: n values between 3 and 10 are compressed in order to improve plot clarity for larger n values

Figure C.1: Rank Pruning – Binary CP-Net Results

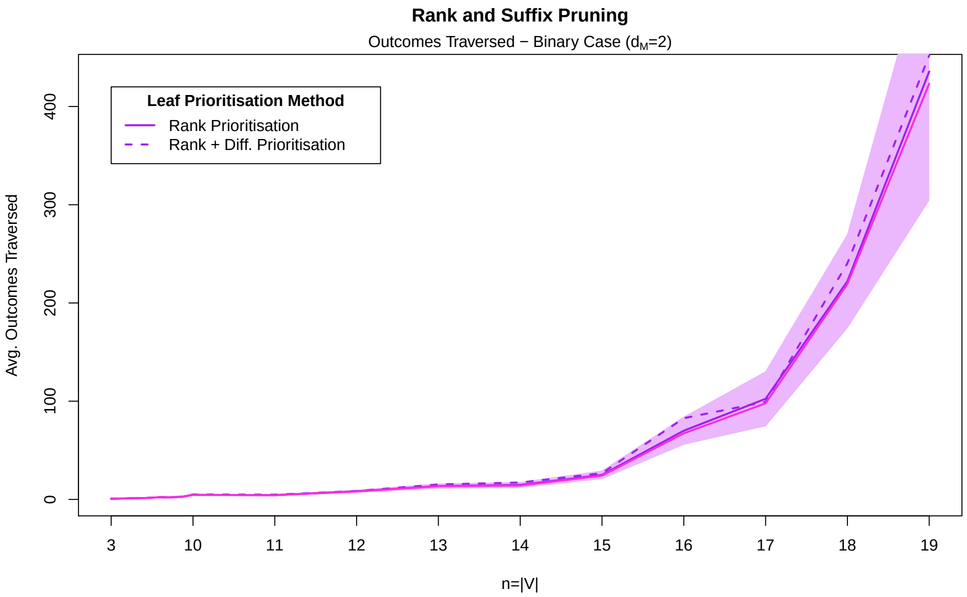


(a) Outcomes Traversed Results

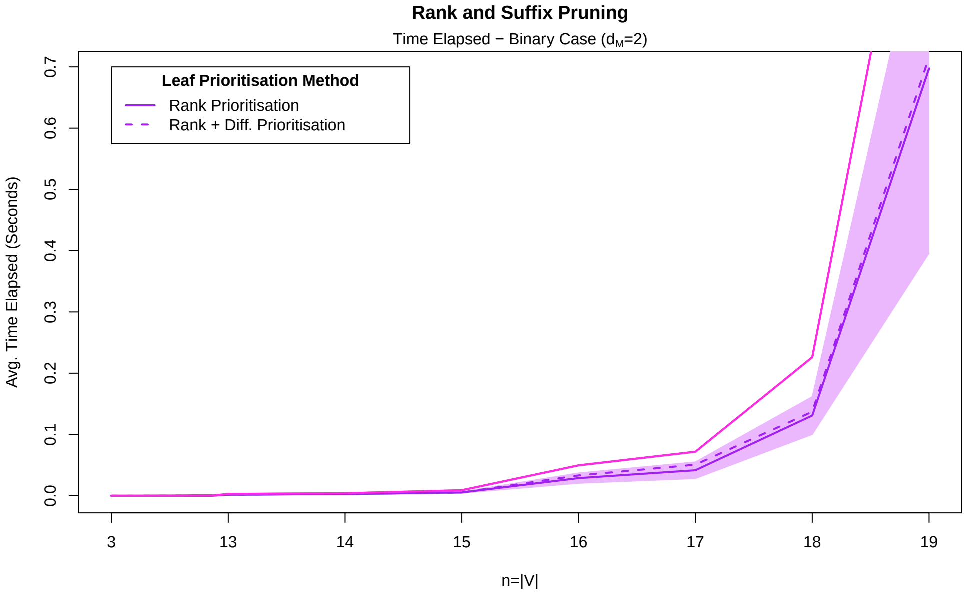


(b) Time Elapsed Results

Figure C.2: Rank Pruning – Multivalued CP-Net Results

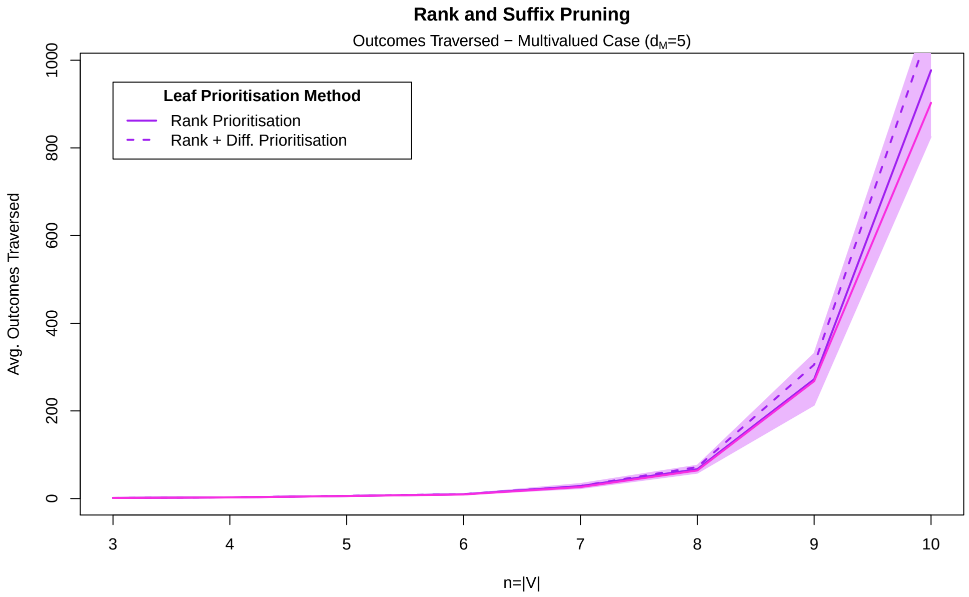


(a) Outcomes Traversed Results
Note: n values between 3 and 10 are compressed in order to improve plot clarity for larger n values

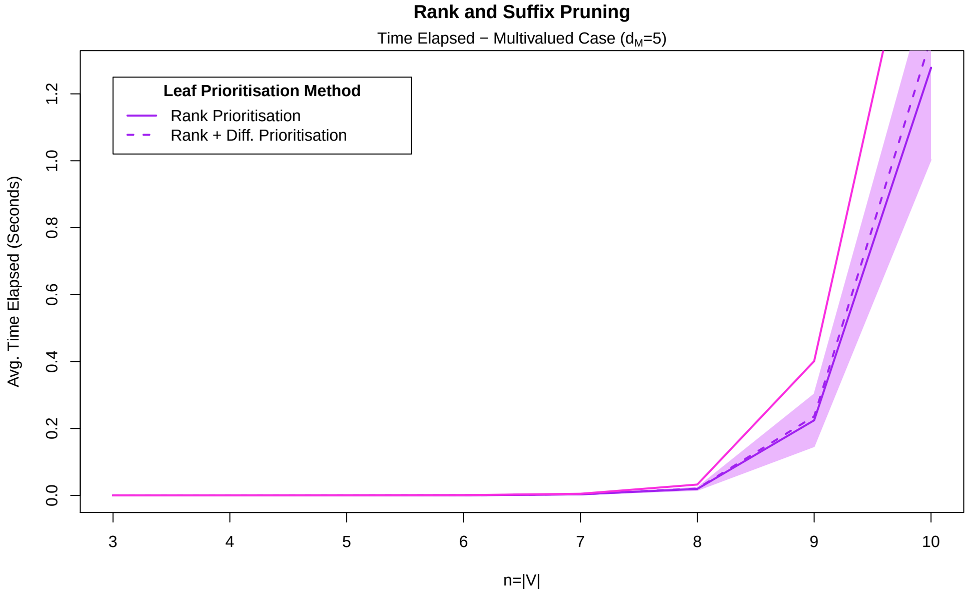


(b) Time Elapsed Results
Note: n values between 3 and 13 are compressed in order to improve plot clarity for larger n values

Figure C.3: Rank Pruning and Suffix Fixing – Binary CP-Net Results

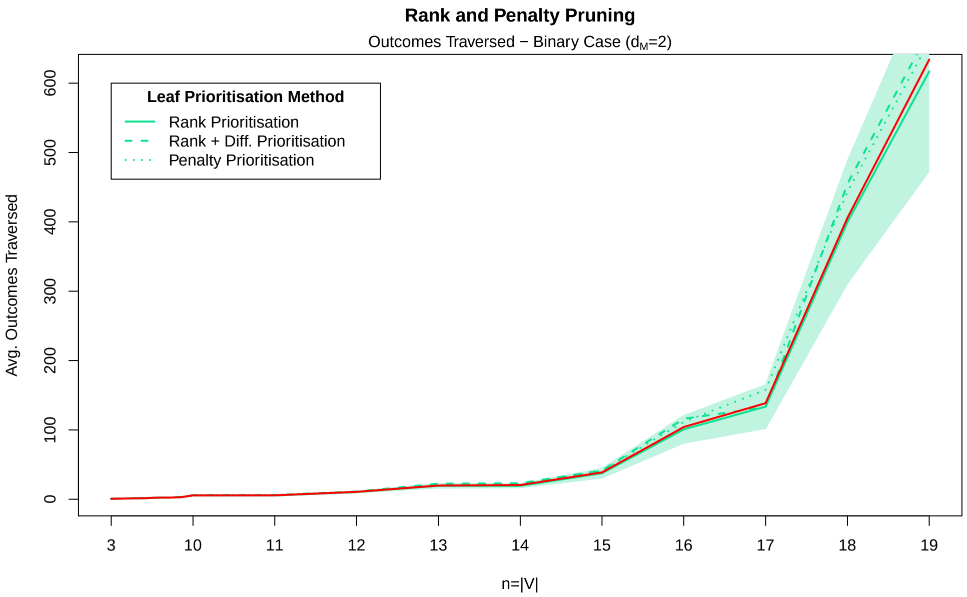


(a) Outcomes Traversed Results



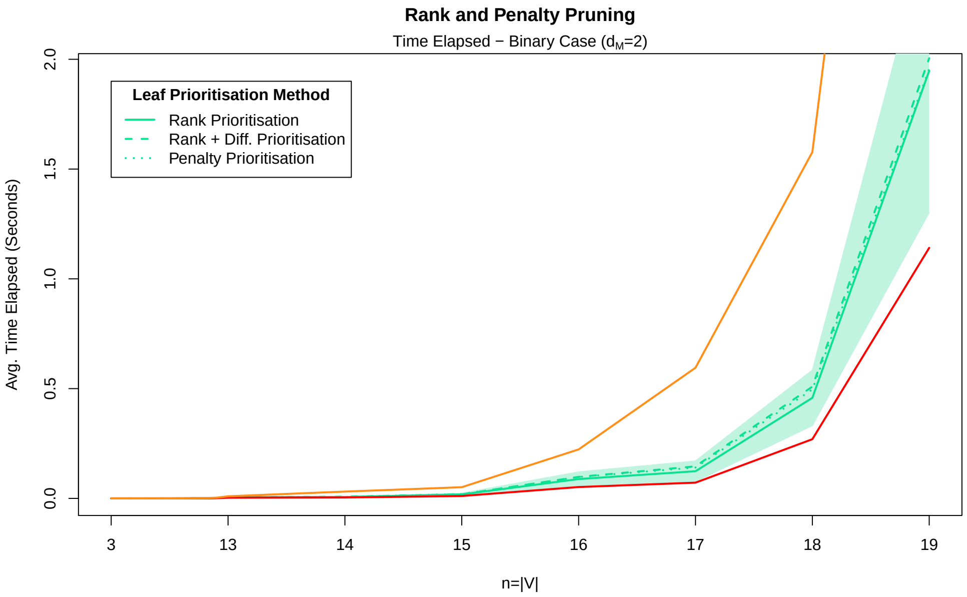
(b) Time Elapsed Results

Figure C.4: Rank Pruning and Suffix Fixing – Multivalued CP-Net Results



(a) Outcomes Traversed Results

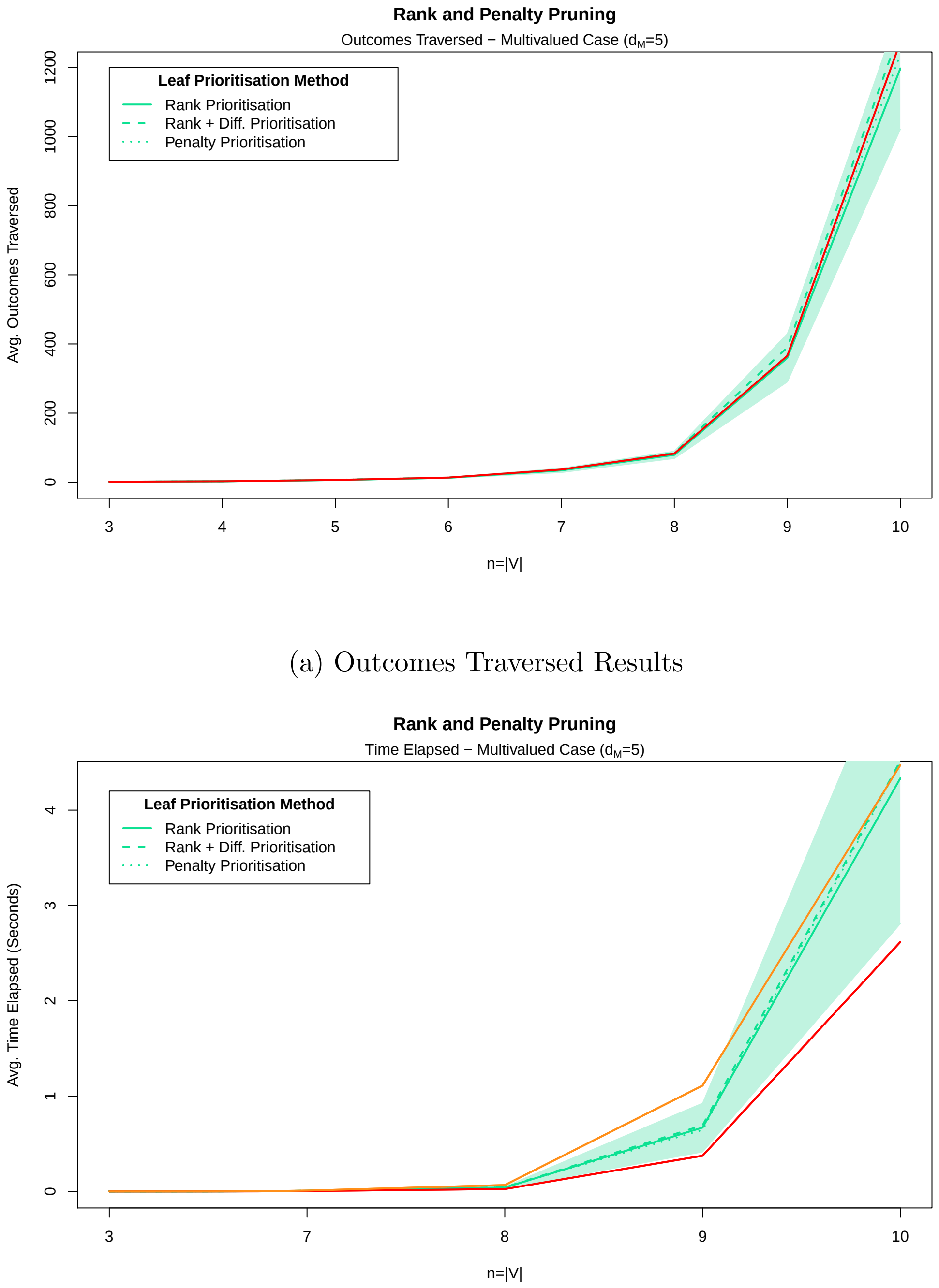
Note: n values between 3 and 10 are compressed in order to improve plot clarity for larger n values



(b) Time Elapsed Results

Note: n values between 3 and 13 are compressed in order to improve plot clarity for larger n values

Figure C.5: Rank and Penalty Pruning – Binary CP-Net Results

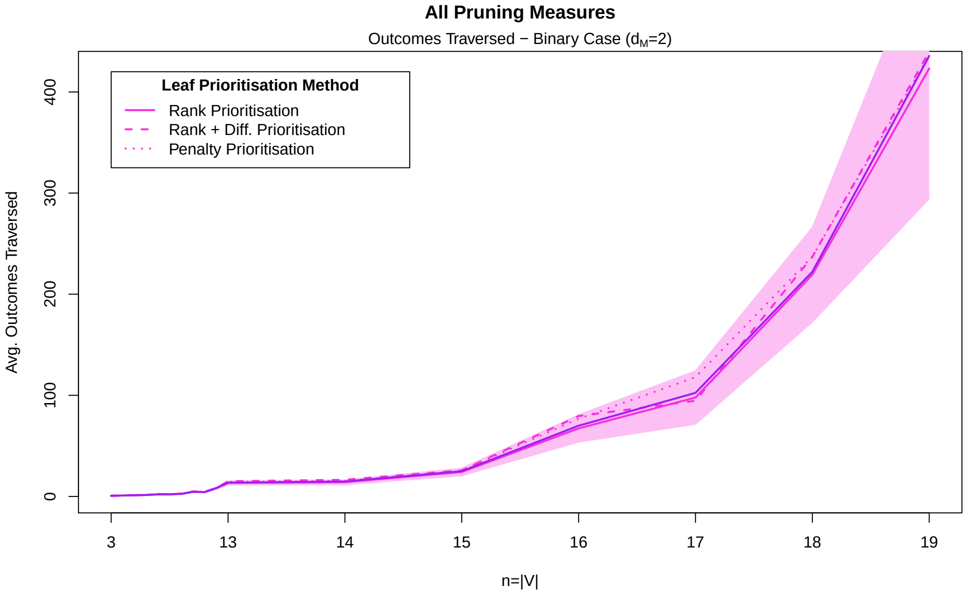


(a) Outcomes Traversed Results

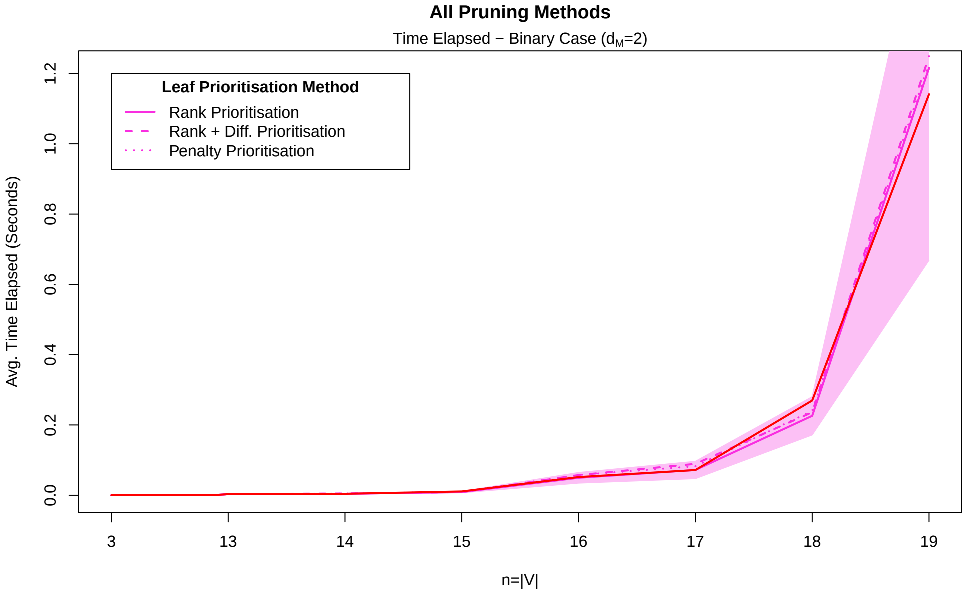
(b) Time Elapsed Results

Note: n values between 3 and 7 are compressed in order to improve plot clarity for larger n values

Figure C.6: Rank and Penalty Pruning – Multivalued CP-Net Results

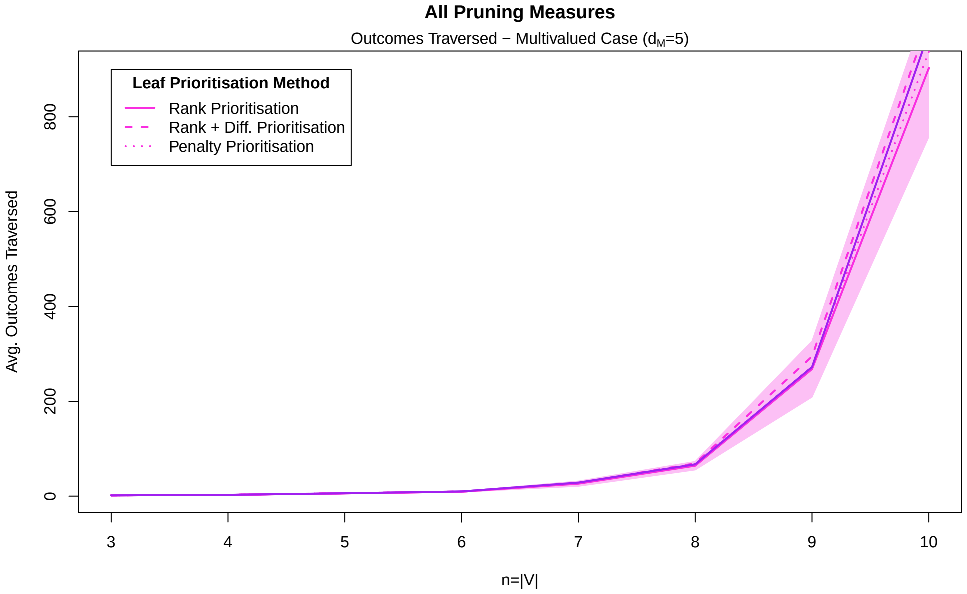


(a) Outcomes Traversed Results
Note: n values between 3 and 13 are compressed in order to improve plot clarity for larger n values

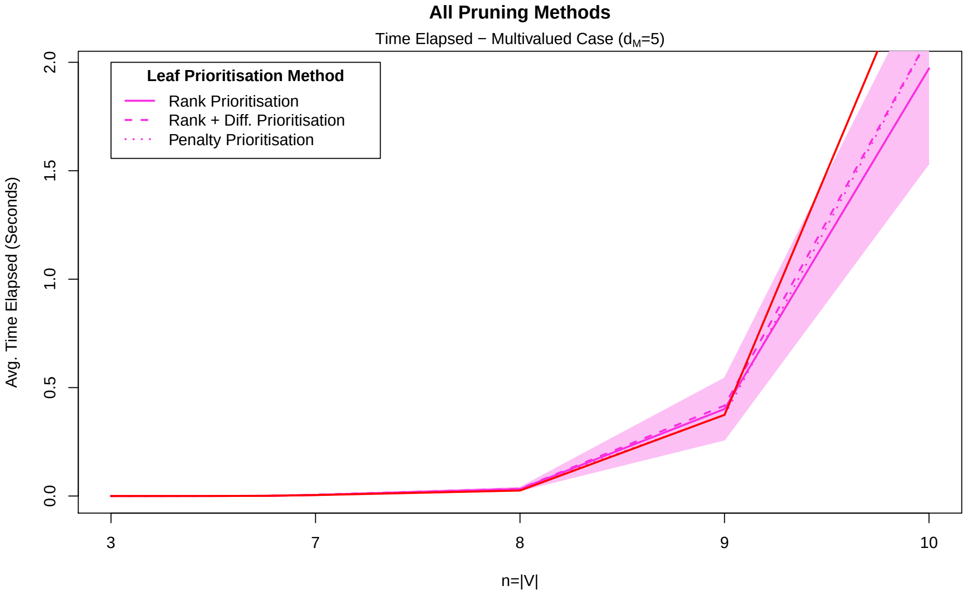


(b) Time Elapsed Results
Note: n values between 3 and 13 are compressed in order to improve plot clarity for larger n values

Figure C.7: All Pruning Methods – Binary CP-Net Results



(a) Outcomes Traversed Results



(b) Time Elapsed Results

Note: n values between 3 and 7 are compressed in order to improve plot clarity for larger n values

Figure C.8: All Pruning Methods – Multivalued CP-Net Results

Appendix D

CP-Net Learning Implementation Details

In this appendix, we give some additional details regarding how to implement our CP-net learning algorithm in practice.

D.1 Monte Carlo Estimation of CPT Scores

In this appendix, we give the details of how the table score defined in Equation 4.7 is estimated via Monte Carlo methods (Robert and Casella, 2004). We also evaluate the accuracy of this estimation.

Let N be a CP-net over variables V and let $U \subseteq V$ be the parent set of $X \in V$. Let us denote $W = V \backslash U \cup \{X\}$. We want to estimate the probability, S, that a given CPT(X) is supported. This probability is given in Equation 4.7:

$$S = \Pr\left(\bigwedge_{\mathbf{u}:x_1 \succ x_2 \in \mathrm{CPT}(X)} \left(\sum_{\mathbf{w} \in \mathrm{Dom}(W)} p_{\mathbf{u}x_1\mathbf{w}} > \sum_{\mathbf{w} \in \mathrm{Dom}(W)} p_{\mathbf{u}x_2\mathbf{w}} \right) \right).$$

This probability is calculated with respect to the distribution over the p_i values given in Equation 4.4:

$$p_1, ..., p_{\mathcal{O}} \sim Dir(\beta_1 + d(o_1), ..., \beta_{\mathcal{O}} + d(o_{\mathcal{O}})).$$

A random draw from the above Dirichlet distribution is a $\mathcal{O}$-length vector, θ, that assigns a value to each p_i:

$$\theta = (q_1, q_2, ..., q_{\mathcal{O}}).$$

As each p_i has a value in θ, we can determine whether the CPT support condition holds for θ. Let I be the indicator function for this condition:

$$I(\theta) = \begin{cases} 1 & \text{if } \bigwedge_{\mathbf{u}:x_1 \succ x_2 \in \text{CPT}(X)} \left(\sum_{\mathbf{w} \in \text{Dom}(W)} q_{\mathbf{u}x_1\mathbf{w}} > \sum_{\mathbf{w} \in \text{Dom}(W)} q_{\mathbf{u}x_2\mathbf{w}} \right), \\ 0 & \text{otherwise.} \end{cases}$$

Proposition D.1. *Let $\theta_1, \theta_2, ..., \theta_N$ be a random sample from the p_i Dirichlet distribution given above. Let us define the estimator, $\hat{S}$, as follows:*

$$\hat{S} = \frac{1}{N} \sum_{i=1}^{N} I(\theta_i).$$

Then $\hat{S}$ is an unbiased estimator of S with $Var(\hat{S}) \in [0, 1/N]$.

Proof. Let Δ^k be the standard $k-$simplex. This is the support set for a $k+1$ dimensional Dirichlet distribution.

$$\Delta^k = \left\{ (a_1, a_2, ..., a_{k+1}) \in \mathbb{R}^{k+1} \middle| \sum_{i=1}^{k+1} a_i = 1 \text{ and } \forall i,\ a_i \in [0,1] \right\}.$$

Let f be the density function for the p_i Dirichlet distribution. Then we can rewrite S as follows:

$$\begin{aligned} S = \Pr\left(\bigwedge_{\mathbf{u}:x_1 \succ x_2 \in \text{CPT}(X)} \left(\sum_{\mathbf{w} \in \text{Dom}(W)} p_{\mathbf{u}x_1\mathbf{w}} > \sum_{\mathbf{w} \in \text{Dom}(W)} p_{\mathbf{u}x_2\mathbf{w}} \right) \right) &= \underset{\{\theta \in \Delta^{\mathcal{O}-1} | I(\theta)=1\}}{\int \cdots \int} f(\theta) \\ &= \underset{\{\theta \in \Delta^{\mathcal{O}-1}\}}{\int \cdots \int} I(\theta) f(\theta) \\ &= E[I(\theta)]. \end{aligned}$$

As $\hat{S}$ is a sample mean of $I(\theta)$, it is an unbiased estimator of $E[I(\theta)]$ and, thus, an unbiased estimator of S. Now consider the variance of $\hat{S}$. First note that, as I is an indicator function, $I^2 = I$.

$$\begin{aligned} Var(\hat{S}) &= \frac{1}{N^2} \sum_{i=1}^{N} Var(I(\theta_i)) \\ &= \frac{1}{N^2} \sum_{i=1}^{N} (E[I(\theta_i)^2] - E[I(\theta_i)]^2) \\ &= \frac{1}{N^2} \sum_{i=1}^{N} (E[I(\theta_i)] - E[I(\theta_i)]^2) \\ &= \frac{1}{N^2} \sum_{i=1}^{N} (S - S^2) \\ &= \frac{S(1-S)}{N}. \end{aligned}$$

As S is a probability, this means $Var(\hat{S}) \in [0, 1/N]$. □

We estimate S by $\hat{S}$, which we can calculate by drawing N samples, θ_i, from the Dirichlet distribution and evaluating $I(\theta_i)$. The variance of $\hat{S}$ indicates the expected estimation error. As $Var(\hat{S}) \in [0, 1/N]$, one can arbitrarily reduce the error by increasing the number of Dirichlet samples used. In the §4.4 experiments, we use 100,000 draws, meaning that the variance is $\leq 10^{-5}$ and the standard deviation (expected error) is ≤ 0.00316.

Remark. Recall that a rule is supported if the left hand sum of Equation 4.5 is greater than the right hand sum. The opposite rule is supported if the right hand sum is greater than the left hand sum. However, if the sums are equal, neither rule is supported. This is not an issue, as these sums are continuous variables and, thus, the probability that they are equal is zero. Hence, in theory, our rule scores and our table scores all sum to one as we would expect.

However, in the above estimation process, we draw random samples from a Dirichlet distribution and evaluate the rule sums to determine whether each rule is supported or not. These draws are randomly generated by a computer and stored as a vector of doubles (a C++ data type). Thus, the draws can only be evaluated to finite precision and there is a non-zero probability that two rule sums could be evaluated as equal. In our experiments, when evaluating the CPT supported by a draw, θ, we found equal sums to be a rare occurrence. For $n = 5$, this happened less than once every 10^9 draws and for $n = 10$ it happened less than once every 10^8 draws. The q_i values are doubles, meaning they were stored to 16 digits of accuracy. Thus, the probability of equality remained very low, but not impossible.

If a Dirichlet sample returns equality, this would suggest that the user does not have a strong preference for either direction of the rule. Thus, such samples were randomly assigned to a direction of the rule. If such cases are common, they would produce equal support for each direction of the rule. Otherwise, if N is reasonably large, such samples will not significantly impact the resulting probability estimations. If many cases of equality occur, this may be a sign that insufficient accuracy is being used. From our experimental results, equality of sums appears to happen when evaluating scores for denser CP-net structures (possibly because the sums are then over a smaller set of q_i values or because a single draw has more inequalities to check) and more frequently when we have less data (meaning more of the Dirichlet parameters are equal or very close). Equality also occurs more often for $n = 10$, which is partially because the former conditions occur more so for larger n, but also because the q_i values will be much smaller (as the outcome set is larger) and, thus, more affected by the loss of accuracy due to computational limitations. The overall frequency in our experiments suggests that the randomised allocations likely had little to no effect on our learning experiments or our results. However, further work is necessary to determine the exact effect of such instances of equality and to evaluate at what frequency they become problematic (by significantly affecting our learning procedure).

D.2 Algorithm for Calculating $MaxS_t$ and $OptCPT$

In this section, we give the algorithm for calculating both

$$\max_{\mathrm{CPT}(X)\in\mathcal{T}(\mathrm{Pa}(X),X)}\{S_t(\mathrm{CPT}(X))\}$$
$$\text{and } \operatorname{argmax}_{\mathrm{CPT}(X)\in\mathcal{T}(\mathrm{Pa}(X),X)}\{S_t(\mathrm{CPT}(X))\}$$

using the Monte Carlo estimation described in Appendix D.1. For ease of notation, we will simplify the above terms to $MaxS_t(X|\mathrm{Pa}(X))$ and $OptCPT(X|\mathrm{Pa}(X))$. The pseudocode for this is given in Algorithm 8.

In Appendix D.1, we showed that we could estimate $S_t(\mathrm{CPT}(X))$ by

$$\hat{S} = \frac{1}{N}\sum_{i=1}^{N} I(\theta_i),$$

where $\theta_1, ..., \theta_N$ are drawn at random from the Dirichlet distribution of the p_i values. The I function is the indicator function for the condition that θ_i supports $\mathrm{CPT}(X)$. See Appendix D.1 for details.

However, as we show in Algorithm 8, identifying the $\mathrm{CPT}(X)$ with maximum S_t score (given a specified parent set) is almost as easy as evaluating the S_t score for a specific $\mathrm{CPT}(X)$. To do so, we first draw a sample of size N from the Dirichlet distribution. Each θ_i in the sample supports exactly one $\mathrm{CPT}(X)$ if $\mathrm{Pa}(X)$ is fixed. Thus, instead of evaluating whether or not θ_i supports a specific CPT, we identify which CPT is supported by θ_i. Then we find (one of) the CPT that is supported by the largest number of θ_i. Denote this CPT by CPT^*. Our probability estimates suggest that this CPT has maximum S_t score. This S_t score can be estimated by

$$\frac{|\{\theta_i|\theta_i \text{ supports } \mathrm{CPT}^*\}|}{N},$$

by Proposition D.1.

Suppose that, in the observed data, outcome o_i was chosen by the user $d(o_i)$ many times. Then the p_i values have the following distribution, as we explained in §4.3.2:

$$p_1, ..., p_{\mathcal{O}} \sim Dir(\beta_1 + d(o_1), ..., \beta_{\mathcal{O}} + d(o_{\mathcal{O}})).$$

Algorithm 8: $MaxS_t$ and $OptCPT$ Calculation

```
Input  : Pa($X$) – Parent set
         $X$ – Variable for which we want to maximise $S_t$
         $D = (d(o_1), ..., d(o_{\mathcal{O}}))$ – User choice data
         $N$ – Dirichlet sample size
         $\beta_1, ..., \beta_{\mathcal{O}}$ – Dirichlet prior parameters
Output: $MaxS_t(X|\mathrm{Pa}(X))$, $OptCPT(X|\mathrm{Pa}(X))$

Generate a random sample $\{\theta_1, ..., \theta_N\}$ from $\mathrm{Dir}(\beta_1 + d(o_1), ..., \beta_{\mathcal{O}} + d(o_{\mathcal{O}}))$;
Initialise two empty vectors, SupportedCPTs and SupportCounts;
for $\theta_i \in \{\theta_1, ..., \theta_N\}$ do
    Suppose $\theta_i = (q_1, q_2, ..., q_{\mathcal{O}})$;
    Initialise an empty CPT($X$) with parent set Pa($X$);
    // Identify the unique CPT supported by $\theta_i$:
    for $\boldsymbol{u} \in Dom(Pa(X))$ do
        $I_1 = \{i | o_i[\mathrm{Pa}(X)] = \mathbf{u}, o_i[X] = x\}$;
        $I_2 = \{i | o_i[\mathrm{Pa}(X)] = \mathbf{u}, o_i[X] = \bar{x}\}$;
        if $\sum_{i \in I_1} q_i > \sum_{i \in I_2} q_i$ then
            $\mathbf{u} : x \succ \bar{x} \rightarrow \mathrm{CPT}(X)$;        // Add this rule to the CPT
        end
        if $\sum_{i \in I_2} q_i > \sum_{i \in I_1} q_i$ then
            $\mathbf{u} : \bar{x} \succ x \rightarrow \mathrm{CPT}(X)$;        // Add this rule to the CPT
        end
    end
    // Either add a support count to CPT($X$) or add CPT($X$) to
       the list of supported CPTs:
    if $CPT(X) \in$ SupportedCPTs then
        Suppose SupportedCPTs[$i$] = CPT($X$);
        SupportCounts[$i$] =SupportCounts[$i$] + 1;
    end
    else
        // Append the supported CPT to the vector and assign a
           support count of 1:
        SupportedCPTs.append(CPT($X$));
        SupportCounts.append(1);
    end
end
$i^* = \mathrm{argmax}_i$(SupportCounts[$i$]);
return $OptCPT(X|Pa(X))$ = SupportedCPTs[$i^*$];
return $MaxS_t(X|Pa(X))$ = SupportCounts[$i^*$]$/N$;
```

Algorithm 8 starts by drawing a sample of size N from this distribution. The value N is a hyperparameter that dictates a tradeoff between the accuracy of our probability estimations (shown in Appendix D.1) and the efficiency of our learning algorithm. In our experiments, we set $N = 100,000$. Even though the pseudocode for Algorithm 8 starts by generating a new Dirichlet sample, in practice, the same sample is used every time Algorithm 8 is called from Algorithm 4. That is, Algorithm 8 only generates a Dirichlet sample the first time it is called. The p_i distribution is fixed, it doesn't change over the course of our learning process, so the Dirichlet sample remains valid at all points of Algorithm 4.

For each θ_i, we then want to identify which unique CPT(X) (given the parent set of X) is supported by θ_i – see the remark in Appendix D.1 for details on the possibility of θ_i supporting more than one CPT. Let $\theta_i = (q_1, q_2, ..., q_{\mathcal{O}})$ and $W = V\backslash\text{Pa}(X) \cup X$. Then we have

$$\begin{aligned} &\theta_i \text{ supports CPT}(X) \\ \iff &\forall \mathbf{u} \in \text{Dom}(\text{Pa}(X)),\ \ \theta_i \text{ supports the rule } \mathbf{u} : x_1 \succ x_2 \in \text{CPT}(X) \\ \iff &\forall \mathbf{u} \in \text{Dom}(\text{Pa}(X)),\ \ \sum_{\mathbf{w}\in W} q_{\mathbf{u}x_1\mathbf{w}} > \sum_{\mathbf{w}\in W} q_{\mathbf{u}x_2\mathbf{w}}. \end{aligned}$$

Note that we are abusing notation slightly here for clarity, using q_{o_i} to denote q_i.

Let us denote the CPT supported by θ_i by CPT$_i$. Then, by the above, we can determine for any $\mathbf{u} \in \text{Dom}(\text{Pa}(X))$, which of $\mathbf{u} : x \succ \bar{x}$ or $\mathbf{u} : \bar{x} \succ x$ is in CPT$_i$ as follows:

$$\begin{aligned} \sum_{\mathbf{w}\in W} q_{\mathbf{u}x\mathbf{w}} > \sum_{\mathbf{w}\in W} q_{\mathbf{u}\bar{x}\mathbf{w}} &\implies \mathbf{u} : x \succ \bar{x} \in \text{CPT}_i, \\ \sum_{\mathbf{w}\in W} q_{\mathbf{u}\bar{x}\mathbf{w}} > \sum_{\mathbf{w}\in W} q_{\mathbf{u}x\mathbf{w}} &\implies \mathbf{u} : \bar{x} \succ x \in \text{CPT}_i. \end{aligned}$$

By determining the rule for every $\mathbf{u} \in \text{Dom}(\text{Pa}(X))$, we have fully determined CPT$_i$, the unique CPT supported by θ_i.

For each θ_i, the algorithm loops through all $\mathbf{u} \in \text{Dom}(\text{Pa}(X))$ and uses the above method to determine which rule out of $\mathbf{u} : x \succ \bar{x}$ and $\mathbf{u} : \bar{x} \succ x$ is supported. From these results, it builds the unique CPT supported by θ_i.

We keep a record of the different CPTs supported by the θ_i samples. For each supported CPT, we also record the number of θ_i that support it. These are recorded in the vectors **SupportedCPTs** and **SupportCounts** respectively. The CPT with the highest support count is then identified. This CPT has the highest (approximated) S_t score out of all CPTs in $\mathcal{T}(\text{Pa}(X), X)$. Thus, this is returned as $OptCPT$. The corresponding (approximated) S_t value is the number of supporting samples as a proportion of N (by the estimator given in Appendix D.1), this is returned as $MaxS_t$.

It is possible that multiple CPTs may have the maximum support count. In this case, we randomly select one such CPT as any choice will be optimal according to our Dirichlet distribution. However, Algorithm 8 could return all optimal CPTs instead, without an increase in complexity. This would produce all optimal CPT configurations for the learned CP-net. One could then either ask the user to select from this set of models, or test each optimal model and use the one that performs best in the required task. This set of models will be much smaller than the space of all possible CP-nets and so such elicitation or testing will be more feasible. In our experiments, we use Algorithm 8 as is and return a single locally optimal CP-net.

Remark. When implementing this algorithm in practice, one has to be careful regarding how the CPTs are stored, due to the magnitude of possibilities. Every CPT contains $2^{|\mathrm{Pa}(X)|}$ rules, each of which can be one of two possibilities ($x \succ \bar{x}$ or $\bar{x} \succ x$). Thus, the number of possible CPTs we are considering is $2^{2^{|\mathrm{Pa}(X)|}}$ (as we do not have any rules against degenerate parents at this stage). Thus, enumeration of the possibilities quickly becomes impossible. Even for $|\mathrm{Pa}(X)| = 6$, the number of possibilities is beyond the range of integers C++ can handle (even if the data type `long long int` is used). This will differ between systems, but as the number of possibilities is double exponential in $|\mathrm{Pa}(X)|$, it is likely to be too much for most computers even for relatively small parent sets. For similar reasons, it is not possible to store a vector of length $2^{2^{|\mathrm{Pa}(X)|}}$ in general. Thus, keeping track of the supported CPTs via enumeration of the CPTs or keeping a vector of support counts for each CPT is not possible once you start considering larger CP-nets.

We get around this in our experiments as follows. We store CPTs as binary vectors of length $2^{|\mathrm{Pa}(X)|}$, where each entry corresponds to a rule – 0 means $x \succ \bar{x}$ and 1 means $\bar{x} \succ x$. This makes the task of determining whether a CPT has been previously supported more difficult, as it requires us to assess the equality of vectors (rather than checking the equality of integers or checking the relevant vector entry). By following the method given in Algorithm 8, we only have to store the distinct supported CPTs. This makes **SupportedCPTs** a vector of length $\leq N$, with entries of length $2^{|\mathrm{Pa}(X)|}$. This can be represented as a vector of length $\leq N \cdot 2^{|\mathrm{Pa}(X)|}$. In our experiments, we use $N = 100,000$. Due to the maximum length of C++ vectors, this allows parent sets of up to size 13. The Dirichlet sample is N random vectors of size 2^n so, by similar reasoning, this sample is possible up to $n = 13$. Thus, using our storage methods, learning can be implemented for CP-nets with up to 13 variables.

Other variations of the storage process might somewhat increase this limit, but due to the exponential growth, this is likely to only be a minor improvement. Alternatively, we can reduce the value of N, but this reduces the accuracy of our score estimations. Further, reducing N to 10,000 only increases the maximum to $n = 16$, but increases our estimation variance by up to a factor of 10. In order for this method to be widely implementable, the storage burden needs to be alleviated. This may be possible by altering the way in which we approximate the scores.

We previously claimed that the process used by Algorithm 8 was almost as easy as calculating the S_t score for a specified CPT(X). As we described in Appendix D.1, we calculate S_t by drawing N random Dirichlet samples and evaluating for each sample the indicator function, I, for supporting the specified CPT. So, as in Algorithm 8, we are looping through N Dirichlet samples, θ_i. In order to evaluate $I(\theta_i)$, we must cycle through $\mathbf{u} \in \text{Dom}(\text{Pa}(X))$ and evaluate the q_i sums, as in Algorithm 8. For each $\mathbf{u}$, we must check that the relative size of the q_i sums supports the relevant rule in the specified CPT(X). The main difference from Algorithm 8 is that, if the q_i sums do not support a rule, we can conclude $I(\theta_i) = 0$ without looking at any more $\mathbf{u} \in \text{Dom}(\text{Pa}(X))$. Thus, when calculating S_t we may not cycle through all $\mathbf{u}$ values every time. Once this loop is done, S_t is estimated by the proportion of θ_i for which $I(\theta_i) = 1$. Thus, calculating a specific S_t is only simpler than Algorithm 8 in that the second loop does not always need to consider all $\mathbf{u}$ values and the fact that the supported CPTs do not need storing and searching for equality or highest count.

This comparison of efficiency is important as the other way to determine $MaxS_t(X|\text{Pa}(X))$ and $OptCPT(X|\text{Pa}(X))$ would be to calculate S_t for each CPT$\in \mathcal{T}(\text{Pa}(X), X)$ and return the CPT with maximum S_t score. This would mean $2^{2^{|\text{Pa}(X)|}}$ calculations of S_t, whereas Algorithm 8 can perform the same task with only a slightly more complex method than a single S_t calculation. Note that the former method does not fix the storage problems as we still need to store a Dirichlet sample.

D.3 Algorithm for Updating Cycles Matrix

In this section, we give our algorithm for updating the cycles matrix described in §4.3.3. This algorithm updates the cycles matrix, C, of acyclic structure A to the cycles matrix for $A \oplus e$, where e is some valid edge change (in particular, changing e did not introduce cycles). This algorithm is more efficient than calculating the cycles matrix for $A \oplus e$ from scratch. This is why we store and update the cycles matrix in Algorithm 4, rather than recalculating after each structure change.

First consider how we calculate C for structure A in the first place. Let the variables be enumerated such that $V = \{X_1, X_2, ..., X_n\}$.

$$C_{i,j} = \begin{cases} 1 & \text{if changing the edge } (X_i, X_j) \text{ creates cycles,} \\ 0 & \text{if changing the edge } (X_i, X_j) \text{ does not create cycles.} \end{cases}$$

If $e = (X, Y)$, then this edge change introduces cycles if and only if $e \notin A$ (removing edges can't create cycles) and $Y = X$ or Y is an ancestor of X in A. Recall that Algorithm 6 calculates the ancestor set of a given variable. We can calculate C from scratch as follows. For each X_i, calculate $\text{Anc}(X_i)$ using Algorithm 6. For each j, $C_{j,i} = 1$ if $A_{j,i} = 0$ and either $j = i$ or $X_j \in \text{Anc}(X_i)$. Otherwise, $C_{j,i} = 0$. This requires calling the ancestor algorithm, Algorithm 6, n times. Algorithm 6 has complexity $O(n^3)$, so this process has complexity $O(n(n^3 + n)) = O(n^4)$. This is how we calculate C the first time in Algorithm 4.

Now we explain how to update C to the cycles matrix for $A \oplus e$, where $e = X_i \to X_j$. Let us denote the cycles matrix for $A \oplus e$ by C'. The pseudocode for this method is given by Algorithm 9. We start by setting $C' = C$ and then change the relevant entries. First, let us demonstrate that certain edges do not need their value updating (that is, the C' entry will be the same as the C entry).

Consider e itself. In order for this change to be implemented, it must have been valid so $C_{i,j} = 0$. If e was added, then $e \in A \oplus e$. As removing an edge cannot create cycles, we must have $C'_{i,j} = 0 = C_{i,j}$. If e was removed, then $e \notin A \oplus e$. If we add e to $A \oplus e$ then we obtain A. As A is acyclic by assumption, adding e to $A \oplus e$ does not create cycles. Thus, $C'_{i,j} = 0 = C_{i,j}$. So the $C'_{i,j}$ value does not need changing.

Consider $e' = X_k \to X_k$ for any k. As A and $A \oplus e$ are acyclic, $e' \notin A$ and $e' \notin A \oplus e$. Adding e' will always create a cycle so $C_{k,k} = 1$. Similarly, $C'_{k,k} = 1 = C_{k,k}$, so the $C'_{k,k}$ value does not need changing.

Let $e' = X_k \to X_m$ be any (non-loop) edge other than e or the reverse of e. That is, $\{k, m\} \neq \{i, j\}$ and $k \neq m$. If $e' \in A \oplus e$, then we must have $e' \in A$ as $e' \neq e$. As removing an edge cannot create cycles, $C_{k,m} = 0$. By the same argument, $C'_{k,m} = 0 = C_{k,m}$. Thus, the $C'_{k,m}$ value does not need changing.

Let $e' = X_k \to X_m$ be any edge such that $\{k, m\} \neq \{i, j\}$ and $k \neq m$ again. Now suppose $e' \notin A \oplus e$. We must have $e' \notin A$ also, as $e' \neq e$. Suppose $X_m \to X_k \in A \oplus e$, then $X_m \to X_k \in A$ as $e \neq X_m \to X_k$. Adding e' to either A

Algorithm 9: Cycle Matrix Update

```
Input  : A – Original structure
         $e = (X_i, X_j)$ – The (valid) edge to be changed
         $C$ – Cycles matrix for $A$
Output: $C'$ – Cycles matrix for $A \oplus e$

$C' = C$;
if $e \notin A$ then
    Calculate Anc($X_i$) (in $A \oplus e$);                // Use Algorithm 6
    Calculate Dec($X_j$) (in $A \oplus e$);                // Use Algorithm 6
    $C'_{j,i} = 1$;
    for $k, m \in \{1, ..., n\}$ such that $k \neq m$, and $\{k, m\} \neq \{i, j\}$ do
        $e' = X_k \to X_m$;
        $e'' = X_m \to X_k$;
        if $e', e'' \notin A \oplus e$ then
            if $C_{k,m} = 0$ then
                if $(k = j \vee X_k \in Dec(X_j)) \wedge (m = i \vee X_m \in Anc(X_i))$ then
                    $C'_{k,m} = 1$;
                end
            end
        end
    end
end
else
    for $k, m \in \{1, ..., n\}$ such that $k \neq m$, and $(k, m) \neq (i, j)$ do
        $e' = X_k \to X_m$;
        $e'' = X_m \to X_k$;
        if $e', e'' \notin A \oplus e$ then
            if $C_{k,m} = 1$ then
                Calculate Anc($X_k$) (for $A \oplus e$);       // Use Algorithm 6
                if $X_m \notin Anc(X_k)$ then
                    $C'_{k,m} = 0$;
                end
            end
        end
    end
end
return $C'$;
```

or $A \oplus e$ will create cycles as the reverse of e' is present. Thus, $C_{k,m} = 1 = C'_{k,m}$ so $C'_{k,m}$ does not need changing.

By the above arguments, the edges that may have different values in C and C' are the reverse of e and edges $X_k \to X_m$ such that $\{k, m\} \neq \{i, j\}$, $k \neq m$, $X_k \to X_m \notin A \oplus e$, and $X_m \to X_k \notin A \oplus e$. We now split into the two cases $e \in A$ and $e \notin A$ and show how the C' values should be updated for these edges.

Suppose $e \notin A$, so the edge change added e to the structure A. As $e \in \ A \oplus \ e$, we know that the reverse cannot be in the structure (as it is acyclic) and it cannot be added, as it would create a cycle, so set $C'_{j,i} = 1$. Let $e' = X_k \to X_m$ such that $\{k, m\} \neq \{i, j\}$, $k \neq m$, and let e'' denote the reverse of e'. Suppose that $e', e'' \notin A \oplus e$. As $e' \neq e$ and $e'' \neq e$, we must have that $e', e'' \notin A$. Suppose $C_{k,m} = 1$, then adding e' to A creates cycles. As $A \oplus e$ is A with an additional edge, adding e' to $A \oplus e$ must also create cycles. That is, $C'_{k,m} = 1 = C_{k,m}$ so $C'_{k,m}$ does not need changing. Suppose instead that $C_{k,m} = 0$. So adding e' to A does not create cycles. As $A \oplus e$ is obtained from A by adding edge e, adding e' to $A \oplus e$ can only create a cycle if that cycle contains e (otherwise adding e' to A would also create cycles). That is, $C'_{k,m} = 1$ if and only if adding e' to $A \oplus e$ creates a cycle containing both e' and e. As $e' = X_k \to X_m$ and $e = X_i \to X_j$, this occurs if and only if $X_k = X_j$ or $X_j \in \text{Anc}(X_k)$ and $X_m = X_i$ or $X_m \in \text{Anc}(X_i)$ (where ancestor sets are defined with respect to the $A \oplus e$ structure). We can rephrase this condition as

$$(X_k = X_j \vee X_k \in \text{Dec}(X_j)) \wedge (X_m = X_i \vee X_m \in \text{Anc}(X_i)).$$

Thus, if we calculate $\text{Anc}(X_i)$ and $\text{Dec}(X_j)$ for $A \oplus e$, we can check this condition for each such e' and change $C'_{k,m}$ to 1 if it holds. By the above argument, if $e \notin A$, then we have changed all appropriate C' entries and thus it is now the cycles matrix for $A \oplus e$.

Note that descendant sets can be found using Algorithm 6 if we flip the direction of all edges in the relevant structure.

Now consider the case where $e \in A$. Then $A \oplus e$ is obtained by removing e from A. Let $e' = X_k \to X_m$ such that $k \neq m$, $e' \neq e$, and let e'' denote the reverse of e'. Suppose that $e', e'' \notin A \oplus e$. Note that, as $e \in A$, the reverse of e cannot be in A and thus is also not present in $A \oplus e$. Thus, the reverse of e is included in the definition of e'. This means that all edges for which we still need to update C' are included in the definition of e'. Suppose $C_{k,m} = 0$, then adding e' to A does not create cycles. As $A \oplus e$ is obtained from A by removing e, adding e' to $A \oplus e$ cannot create cycles. If adding e' to $A \oplus e$ created a cycle, then the same cycle would be formed by adding e' to A, contradicting $C_{k,m} = 0$. Thus, $C'_{k,m} = 0 = C_{k,m}$, so $C'_{k,m}$ does not need changing.

Now suppose that $C_{k,m} = 1$. Adding e' to A creates a cycle. We want to determine whether adding e' to $A \oplus e$ (A with edge e removed) also creates a cycle. This is true if and only if there is a directed path from X_m to X_k in $A \oplus e$, that is, $X_m \in \text{Anc}(X_k)$. For each such e', we calculate $\text{Anc}(X_k)$ for $A \oplus e$ and determine whether $X_m \in \text{Anc}(X_k)$. If it is not, then we set $C'_{k,m} = 0$. By the above argument, if $e \in A$, then we have changed all appropriate C' entries and thus it is now the cycles matrix for $A \oplus e$.

The algorithm returns C', which, by the above arguments, is the cycles matrix for $A \oplus e$.

Let us consider the complexities of calculating and updating a cycles matrix. For now, let us ignore the calculation of ancestor sets. When calculating C, for each i, j pair, some simple conditions are checked and the $C_{i,j}$ value is assigned. When updating C, for each i, j pair, some simple conditions are checked and then, sometimes, the C' value is changed. Both of these tasks have complexity $O(n^2)$. Thus, what determines the difference in efficiency is the number of times ancestor sets are calculated (using Algorithm 6). Calculating C requires the ancestor set of every variable to be calculated. Algorithm 6 has complexity $O(n^3)$, thus calculating C has complexity $O(n \cdot n^3 + n^2) = O(n^4)$. If we have $e \notin A$, then Algorithm 9 only calculates two ancestor sets. Thus, the complexity of updating C is $O(2n^3 + n^2) = O(n^3)$. If we have $e \in A$, then Algorithm 9 calculates one ancestor set for every edge that satisfies a certain set of properties. However, if we assume that calculated ancestors are stored and thus never re-calculated, this means that the number of ancestor set calculations is $\leq n$. In fact, due to the edge conditions, it must be $\leq n - 2$ (if $n \geq 3$) and may be much less than that depending on the structure of A. Thus, in this case, the complexity is $O((n-2)n^3 + n^2) = O(n^4)$. Therefore, updating will always be more efficient (to varying degrees) than calculating a new cycles matrix from scratch, even though theoretical complexity is the same in the $e \in A$ case.

Appendix E

Proofs

In this Appendix, we provide the proofs of various results stated throughout Chapters 2 – 4 and the previous appendices.

E.1 Proof of Proposition 2.2

Proposition 2.2. *Let N be a CP-net and let $T(N)$ be the event tree representation of N. Then N and $T(N)$ are equivalent structures (they encode identical information). Recall that a CP-net consists of both the structure and the CPTs.*

Proof. We have already described in §2.3.1 the process for obtaining $T(N)$ from N. Thus, to prove that $T(N)$ encodes exactly the same information as N (that is, no information is lost by moving from N to $T(N)$), we will show that N can be reconstructed from $T(N)$.

From $T(N)$, we can directly obtain the variable domains (from edge labels) and a topological ordering (this is the order of the variables in the tree). Suppose $X_1, X_2, ..., X_n$ is this topological ordering, then $\text{Pa}(X_i) \subseteq \{X_1, ..., X_{i-1}\}$. If $Y \in \text{Pa}(X_i)$ (and is non-degenerate), then there must be two $\text{Pa}(X_i)$ assignments, $\mathbf{u}_1$ and $\mathbf{u}_2$, that differ only on the value of Y and that result in distinct X_i preference orders. Let A denote the predecessors of X_i that are not parents of X_i, $A = \{X_1, ..., X_{i-1}\} \backslash \text{Pa}(X_i)$. Let $\mathbf{a} \in \text{Dom}(A)$. As a variable's preference depends only on the parent values, the assignments $\mathbf{au}_1$ and $\mathbf{au}_2$ imply distinct *ceteris paribus* preference orders over X_i. Further, $\mathbf{au}_1$ and $\mathbf{au}_2$ differ only on the value taken by Y. Conversely, if such a pair of assignments to $\{X_1, ..., X_{i-1}\}$ exist, that differ only on Y and imply distinct preference orders over X_i, then X_i must be preferentially dependent upon the value taken by Y. Thus, by definition, $Y \in \text{Pa}(X_i)$. Therefore, $Y \in \text{Pa}(X_i)$ if and only if there are two assignments of $\{X_1, ..., X_{i-1}\}$ that differ only on Y and that imply distinct preference orders over X_i. By the way in which we construct $T(N)$ from N, this is equivalent to the existence of two directed paths p and p' in $T(N)$ with the following properties. First, both p and p' originate at the root of $T(N)$ and have length $(i-1)$. By definition of $T(N)$, p and p' must assign values to $\{X_1, ..., X_{i-1}\}$ only. Second, p and p' differ only on the value assigned to Y and the implied preference order over X_i is different for the two associated $\{X_1, ..., X_{i-1}\}$ assignments. That is, the labelling of the branches corresponding to the X_i assignments that originate at the end of path p is different to those that originate at the end of p'. For each $Y \in \{X_1, ..., X_{i-1}\}$, we can check whether such a pair of paths exists. In exactly those cases where such paths exist, Y is a parent of X_i by the above argument. Thus, we can construct the parent set, $\mathrm{Pa}(X_i)$ by checking this condition on $T(N)$ for each $Y \in \{X_1, ..., X_{i-1}\}$. Identifying the parent set of each variable determines the whole structure of N and, thus, it only remains to construct the CPTs.

Given the parents of a variable and their respective domains, we know all of the possible parental assignments, which correspond to the CPT rows. Given a row of CPT(X_i), that is, an assignment of values $\mathbf{u} \in \mathrm{Dom}(\mathrm{Pa}(X_i))$, we need to recover the corresponding preference order over X_i. This can be done as follows; first, obtain any path p of length $(i-1)$ that begins at the root of $T(N)$ and assigns $\mathrm{Pa}(X_i)$ the values in $\mathbf{u}$. By the way in which we constructed $T(N)$ from N, the labels of the X_i value branches that come directly after p encode the preference order over X_i implied by the assignment of values to $\{X_1, ..., X_{i-1}\}$ by p. However, as X_i is only preferentially dependent upon $\mathrm{Pa}(X_i)$, this is the preference order over X_i implied by $\mathrm{Pa}(X_i) = \mathbf{u}$. These edge labels can then be used to obtain the preference order; the value of X_i assigned the label '1^{st}' comes first in the preference order (most preferred), the value assigned the label '2^{nd}' comes second and so on. Thus, from these labels we can determine the relevant preference order over X_i and fill in the CPT row. Following this procedure, we can populate all of the CPTs for N. Thus, N has been fully reconstructed from $T(N)$. □

E.2 Proof of Theorem 2.8

Theorem 2.8. *Given a CP-net, N, for any outcomes o and o', we have that $N \vDash o \succ o' \implies r(o) > r(o')$.*

Proof. Before we commence the proof, recall the following. The edge of $W(N)$ that indicates that $X = x$, given $\mathrm{Pa}(X) = \mathbf{u}$ previously, has the following weight:

$$AF_X(d_X + 1)P_P(X = x \mid \mathrm{Pa}(X) = \mathbf{u}). \tag{E.1}$$

Full explanation of this notation is given in Definition 2.3.

If $N \vDash o \succ o'$, then there exists an improving flipping sequence of outcomes, $o' = o_1, o_2, ..., o_m = o$, such that o_{i+1} differs from o_i on the value taken by exactly one variable and $N \vDash o_i \prec o_{i+1}$ (Boutilier et al., 2004a). Thus, proving the theorem for o and o' that differ on the value of exactly one variable is sufficient, as the more general result follows by transitivity.

Suppose o and o' differ only on the value taken by X. Let x and x' be the values assigned to X in o and o' respectively (that is, $o[X] = x$ and $o'[X] = x'$). Let $\mathbf{u}$ be the set of values assigned to $\text{Pa}(X)$ in both outcomes ($\mathbf{u} = o[\text{Pa}(X)] = o'[\text{Pa}(X)]$).

Let $x_1 \succ x_2 \succ \cdots \succ x_m$ be the preference ordering over $\text{Dom}(X)$ given that $\text{Pa}(X) = \mathbf{u}$. This is the row of $\text{CPT}(X)$ that corresponds to $\text{Pa}(X) = \mathbf{u}$. Suppose $x = x_i$ and $x' = x_j$, we know that $i < j$ as $o' \to o$ is an improving flip of X (as $N \vDash o \succ o'$).

Let o_k denote the outcome where $o_k[X] = x_k$ and, for all variables $Y \neq X$, $o_k[Y] = o[Y](= o'[Y])$. Then the sequence of outcomes, $o_m, o_{m-1}, ..., o_1$, is a sequence of X flips through the values $x_m, x_{m-1}, ..., x_1$. As $\text{Pa}(X) = \mathbf{u}$ in each o_k, these are improving flips of X, so $N \vDash o_1 \succ o_2 \succ \cdots \succ o_m$. Notice that $o = o_i$ and $o' = o_j$ with $i < j$, so we have $N \vDash o = o_i \succ o_{i+1} \succ \cdots \succ o_j = o'$. Hence, it is sufficient to prove $r(o) > r(o')$ for the specific case where x and x' are adjacent in the ordering $x_1 \succ x_2 \succ \cdots \succ x_m$, that is, when $j = i + 1$. The more general case, where x and x' are not adjacent ($j > i + 1$), follows by the fact that $>$ is transitive.

To see this explicitly, suppose we have proven the case where x and x' are adjacent ($j = i + 1$). If we then have the non-adjacent case ($j > i+1$), then we have $N \vDash \ o = \ o_i \succ o_{i+1} \succ \cdots \succ o_j = o'$. From the adjacent case, we get that $r(o_i) > r(o_{i+1})$, $r(o_{i+1}) > r(o_{i+2})$,...,$r(o_{j-1}) > r(o_j)$. Thus, by the transitivity of $>$, we have $r(o_i) > r(o_j)$, that is, $r(o) > r(o')$. It is therefore sufficient to prove the theorem in the case where $o' \to o$ is an improving flip of X between adjacent values of X in the ordering $x_1 \succ x_2 \succ \cdots \succ x_m$. Thus, we can assume that x and x' are adjacent values, that is, x and x' are the i^{th} and $(i+1)^{th}$ most preferred values of X, given $\text{Pa}(X) = \mathbf{u}$ (for some i).

We now demonstrate that $r(o) > r(o')$ under the above assumptions. Let p and p' be the root-to-leaf paths of $W(N)$ that correspond to o and o'. Recall that $r(o)$ is the sum of the edge weights of p. Similarly, $r(o')$ is the sum of the edge weights of p'. Thus, to evaluate these ranks, we must first determine what these edge weights are.

Let $Y \in V$ be a variable such that $Y \neq X$ and $X \notin \text{Pa}(Y)$. As o and o' differ only on the value of X, Y and $\text{Pa}(Y)$ must take the same values in both o and o'. Let $y = o[Y] = \ o'[Y]$ ($y \in \text{Dom}(Y)$) and $\mathbf{w} = o[\text{Pa}(Y)] = o'[\text{Pa}(Y)]$ ($\mathbf{w} \in \text{Dom}(\text{Pa}(Y))$). By Definition 2.3, the weight assigned to the edge of p indicating that Y takes the value y is $AF_Y(d_Y + 1)P_P(Y = y \mid \text{Pa}(Y) = \mathbf{w})$. The weight assigned by Definition 2.3 to the edge of p' indicating that $Y = y$ is identical. Thus, any such variable (any variable that is neither X itself, nor a child of X) contributes exactly the same quantity to both sums, $r(o)$ and $r(o')$. Let α denote the total contribution to $r(o)$ (and thus $r(o')$ also) by such variables.

The weight on the edge of p indicating that X takes the value x is $AF_X(d_X+1)P_P(X = x \mid \mathrm{Pa}(X) = \mathbf{u})$. As we have assumed x to be the i^{th} most preferred value of X, given $\mathrm{Pa}(X) = \mathbf{u}$, $P_P(X = x \mid \mathrm{Pa}(X) = \mathbf{u}) = \frac{n_X - i + 1}{n_X}$.

The weight on the edge of p' indicating that X takes the value x' is $AF_X(d_X+1)P_P(X = x' \mid \mathrm{Pa}(X) = \mathbf{u})$. As we have assumed x' to be the $(i+1)^{th}$ most preferred value of X, given $\mathrm{Pa}(X) = \mathbf{u}$, $P_P(X = x' \mid \mathrm{Pa}(X) = \mathbf{u}) = \frac{n_X - (i+1) + 1}{n_X}$.

Let $\mathrm{Ch}(X) = \{Y_1, ..., Y_\ell\}$ be the set of variables that have X as one of their parent variables in the structure of N. These are the children of X. As N is acyclic, $Y_j \neq X$ and so Y_j takes the same value in both o and o'. Let $y_j = o[Y_j]$ $(= o'[Y_j])$. Let $\mathbf{v}_j = o[\mathrm{Pa}(Y_j)]$ and $\mathbf{v}'_j = o'[\mathrm{Pa}(Y_j)]$. The weight on the edge of p that indicates $Y_j = y_j$ is $AF_{Y_j}(d_{Y_j}+1)P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}_j)$. The weight on the edge of p' that indicates $Y_j = y_j$ is $AF_{Y_j}(d_{Y_j}+1)P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}'_j)$.

Now that we know the weights of all edges in p and p', we can evaluate $r(o)$ and $r(o')$.

$$r(o) = \alpha + AF_X(d_X+1)\frac{n_X - i + 1}{n_X} + \sum_{j=1}^{\ell} AF_{Y_j}(d_{Y_j}+1)P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}_j),$$

$$r(o') = \alpha + AF_X(d_X+1)\frac{n_X - i}{n_X} + \sum_{j=1}^{\ell} AF_{Y_j}(d_{Y_j}+1)P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}'_j).$$

Recall that, for any $Z \in V$, $AF_Z = \prod_{W \in \mathrm{Anc}(Z)} \frac{1}{n_W}$. If Y_j is a child of X, then $\mathrm{Anc}(X) \cup \{X\} \subseteq \mathrm{Anc}(Y_j)$. We know $X \notin \mathrm{Anc}(X)$ as N is acyclic. Thus, $AF_{Y_j} = AF_X \frac{1}{n_X}\beta_j$, for some $0 < \beta_j \leq 1$.

Notice that, for any $Z \in V$, $z \in \mathrm{Dom}(Z)$, and $\mathbf{w} \in \mathrm{Dom}(\mathrm{Pa}(Z))$, we have $\frac{1}{n_Z} \leq P_P(Z = z \mid \mathrm{Pa}(Z) = \mathbf{w}) \leq 1$. Thus, for any $1 \leq j \leq \ell$, we have

$$\frac{1}{n_{Y_j}} \leq P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}_j) \leq 1,$$
$$\frac{1}{n_{Y_j}} \leq P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}'_j) \leq 1.$$

Using these results, we can rewrite $r(o)$ and $r(o')$ and obtain the following inequalities:

$$\begin{aligned}
r(o) =&\alpha + AF_X(d_X+1)\frac{n_X-i+1}{n_X}\\
&+\sum_{j=1}^{\ell} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)P_P(Y_j=y_j \mid \mathrm{Pa}(Y_j)=\mathbf{v}_j)\\
\geq&\alpha + AF_X(d_X+1)\frac{n_X-i+1}{n_X} + \sum_{j=1}^{\ell} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)\frac{1}{n_{Y_j}},
\end{aligned}$$

$$\begin{aligned}
r(o') =&\alpha + AF_X(d_X+1)\frac{n_X-i}{n_X}\\
&+\sum_{j=1}^{\ell} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)P_P(Y_j=y_j \mid \mathrm{Pa}(Y_j)=\mathbf{v}'_j)\\
\leq&\alpha + AF_X(d_X+1)\frac{n_X-i}{n_X} + \sum_{j=1}^{\ell} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)\cdot 1.
\end{aligned}$$

Next we show that $d_X = \sum_{j=1}^{\ell}(d_{Y_j}+1)$. As $|\mathrm{Ch}(X)| = \ell$, there are exactly ℓ directed paths of length 1 that originate at X in the structure of N. Thus, $d_X - \ell$ is the number of directed paths of length greater than 1 that originate at X (in the structure of N). Every such path can be turned into a distinct directed path that originates at one of $\{Y_1, ..., Y_\ell\}$ by removing the first edge. Further, any path that originates at some $Y_j \in \{Y_1, ..., Y_\ell\}$ can be turned into a distinct directed path of length greater than 1 that originates at X by attaching $X \rightarrow Y_j$ to the beginning. Thus, the number of directed paths of length greater than 1 that originate at X is equal to the number of directed paths that originate at some $Y_j \in \{Y_1, ..., Y_\ell\}$. That is, $d_X - \ell = \sum_{j=1}^{\ell} d_{Y_j}$ or, equivalently, $d_X = \sum_{j=1}^{\ell} d_{Y_j} + \ell = \sum_{j=1}^{\ell}(d_{Y_j}+1)$.

Recall that our aim is to prove that $r(o) > r(o')$. For the purposes of contradiction, suppose that $r(o) \leq r(o')$. Then

$$\begin{aligned}
&\alpha + AF_X(d_X+1)\frac{n_X-i+1}{n_X} + \sum_{j=1}^{\ell} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)\frac{1}{n_{Y_j}}\\
&\leq \alpha + AF_X(d_X+1)\frac{n_X-i}{n_X} + \sum_{j=1}^{\ell} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)\cdot 1
\end{aligned}$$

$$\begin{aligned}
&\implies (d_X+1)(n_X-i+1)+\sum_{j=1}^{\ell}\beta_j(d_{Y_j}+1)\frac{1}{n_{Y_j}}\\
&\leq (d_X+1)(n_X-i)+\sum_{j=1}^{\ell}\beta_j(d_{Y_j}+1)\cdot 1\\
&\implies d_X+1\leq \sum_{j=1}^{\ell}\beta_j(d_{Y_j}+1)\bigg(1-\frac{1}{n_{Y_j}}\bigg).
\end{aligned}$$

Now let $m=\max\{n_{Y_j}\mid 1\leq j\leq \ell\}$. This implies

$$d_X+1\leq \sum_{j=1}^{\ell}\beta_j(d_{Y_j}+1)\bigg(1-\frac{1}{m}\bigg).$$

Since $0<\beta_j\leq 1$ for all $1\leq j\leq \ell$, it follows that

$$d_X+1\leq \sum_{j=1}^{\ell}1\cdot(d_{Y_j}+1)\bigg(1-\frac{1}{m}\bigg).$$

Recall that $d_X=\sum_{j=1}^{\ell}(d_{Y_j}+1)$, this implies

$$(d_X+1)\leq d_X\bigg(1-\frac{1}{m}\bigg).$$

If X has no descendent paths, that is, $d_X=0$, then we have shown $r(o)\leq\ r(o')\implies 1\leq 0$. So we have derived a contradiction.

If $d_X>0$, then $r(o)\leq\ r(o')$ implies that

$$\begin{aligned}
&1+\frac{1}{d_X}\leq 1-\frac{1}{m}<1\\
&\implies \frac{1}{d_X}<0\\
&\implies 1<0.
\end{aligned}$$

Thus, we have again derived a contradiction and so we can conclude $r(o)>r(o')$. The general result, for any outcomes o and o', follows by transitivity as we discussed above. □

E.3 Proof of Theorem 2.12

Theorem 2.12. *Let N be a CP-net and $\succsim^C$ be any consistent ordering over the outcomes. Then no information is lost by reducing N to $\succsim^C$. That is, $\succsim^C$ encodes all of the preference information given by N.*

Proof. To prove this, we will show that we can reconstruct N from $\succsim^C$. The number of variables and their respective domains can be read off the outcomes themselves. This gives us the nodes of the CP-net structure. It remains to evaluate the edges of the structure and the CPTs.

Let $X, Y \in V$, $X \neq Y$. If $X \in \text{Pa}(Y)$ (that is, there is an edge $X \to Y$ in the structure of N), then the preference over Y must be dependent upon X. Let $\text{Dom}(Y) = \{y_1, ..., y_m\}$ and $W = V \backslash \{X, Y\}$. If Y is preferentially dependent upon X, then there exists $x, \bar{x} \in \text{Dom}(X)$ and $\mathbf{w} \in \text{Dom}(W)$ such that the preferential order over $\text{Dom}(Y)$ is different under $XW = x\mathbf{w}$ and $XW = \bar{x}\mathbf{w}$ – otherwise, we do not need to know the assignment of X to determine the preference order over Y and so it cannot be a parent. Every pair of outcomes in $\{xy_1\mathbf{w}, xy_2\mathbf{w}, ..., xy_m\mathbf{w}\}$ differ only on Y (they constitute a Y flip) and so N entails a preference between them. Thus, N entails a total ordering over these outcomes that must be reflected by $\succsim^C$, as it is a consistent ordering. Thus, if $\succsim^C$ is restricted to this outcome set, it produces the same strict total ordering. Similarly for $\{\bar{x}y_1\mathbf{w}, \bar{x}y_2\mathbf{w}, ..., \bar{x}y_m\mathbf{w}\}$. By the above argument, there is an edge $X \to Y$ if and only if there exists $x, \bar{x} \in \text{Dom}(X)$ and $\mathbf{w} \in \text{Dom}(W)$ such that the strict $\succ^C$ orders over $\{xy_1\mathbf{w}, xy_2\mathbf{w}, ..., xy_m\mathbf{w}\}$ and $\{\bar{x}y_1\mathbf{w}, \bar{x}y_2\mathbf{w}, ..., \bar{x}y_m\mathbf{w}\}$ are different. Thus, by testing this condition for each $x, \bar{x} \in \text{Dom}(X)$ and $\mathbf{w} \in \text{Dom}(W)$, we can determine whether there exists a $X \to Y$ edge directly from $\succsim^C$. Thus, the structure of N can be reconstructed from $\succsim^C$.

Consider $\text{CPT}(Y)$, the rows correspond to the possible assignments of values to $\text{Pa}(Y)$. We already know these assignments as we know the CP-net structure and variable domains. Thus, we only need to fill in the relevant preferences. Let $U = \text{Pa}(Y)$ and redefine $W = V \backslash U \cup \{Y\}$. The row of $\text{CPT}(Y)$ corresponding to $\mathbf{u} \in \text{Dom}(U)$ gives the preference order over Y, given that $U = \mathbf{u}$. By CP-net semantics, this preference induces the same entailed total ordering over $\{\mathbf{u}y_1\mathbf{w}, ..., \mathbf{u}y_m\mathbf{w}\}$ for every $\mathbf{w} \in \text{Dom}(W)$. As this ordering is entailed, it is reflected in $\succsim^C$. Thus, the preference order over Y, given $U = \mathbf{u}$, can be read off $\succsim^C$ by restricting $\succsim^C$ to $\{\mathbf{u}y_1\mathbf{w}, ..., \mathbf{u}y_m\mathbf{w}\}$ for any $\mathbf{w} \in \text{Dom}(W)$. Thus, we can also populate the CPT rows from $\succsim^C$ directly. Hence, we have completely reconstructed N from $\succsim^C$, showing that no information is lost by reducing N to a consistent ordering. □

E.4 Proof of Lemma 2.17

Lemma 2.17. *Let N be a CP-net over variables V. For any $X \in V$, $L(X) > 0$.*

Proof. Let $Y \in \mathrm{Ch}(X)$ for some $Y, X \in V$, then, by the same reasoning given in the proof of Theorem 2.8, $AF_Y = AF_X \frac{1}{n_X} \beta_Y$ for some $0 < \beta_Y \leq 1$. Also, $\sum_{Y \in \mathrm{Ch}(X)} (d_Y + 1) = d_X$, as shown in the proof of Theorem 2.8.

Suppose, for the sake of contradiction, that $L(X) \leq 0$ for some $X \in V$. This implies that

$$\begin{aligned}
&AF_X(d_X+1)\frac{1}{n_X} - \sum_{Y \in \mathrm{Ch}(X)} AF_Y(d_Y+1)\frac{n_Y-1}{n_Y} \leq 0 \\
\implies &AF_X(d_X+1)\frac{1}{n_X} \leq \sum_{Y \in \mathrm{Ch}(X)} AF_X \frac{1}{n_X}\beta_Y(d_Y+1)\frac{n_Y-1}{n_Y} \\
\implies &(d_X+1) \leq \sum_{Y \in \mathrm{Ch}(X)} \beta_Y(d_Y+1)\frac{n_Y-1}{n_Y}.
\end{aligned}$$

As $\beta_Y, \frac{n_Y-1}{n_Y} \leq 1$ for all $Y \in \mathrm{Ch}(X)$, it follows that

$$(d_X+1) \leq \sum_{Y \in \mathrm{Ch}(X)} (d_Y+1) = d_X.$$

Thus, we have reached a contradiction and so we can conclude that $L(X) > 0$ for all $X \in V$. □

E.5 Proof of Corollary 2.18

Corollary 2.18. *Let N be a CP-net over variables V. Let o_1 and o_2 be associated outcomes and $D = \{X \in V \mid o_1[X] \neq o_2[X]\}$. Then,*

$$N \vDash o_1 \succ o_2 \implies r(o_1) - r(o_2) \geq \sum_{X \in D} L(X) > 0.$$

This is a tight lower bound on the rank difference implied by entailment.

Proof. If $N \vDash o_1 \succ o_2$, then there exists a sequence of outcomes

$$o_2 = \ p_1, p_2, ..., p_m = \ o_1,$$

such that $N \vDash p_1 \prec p_2 \prec \cdots \prec p_m$ and p_i and p_{i+1} differ on the value taken by exactly one variable (Boutilier et al., 2004a). That is, starting at o_2, we can reach o_1 through $m-1$ improving variable flips. By Theorem 2.8, we know that $r(p_{i+1}) - \ r(p_i) > 0$. We can rewrite $r(o_1) - r(o_2)$ as the sum of the rank improvements of each flip as follows:

$$r(o_1) - r(o_2) = [r(p_2) - r(p_1)] + [r(p_3) - r(p_2)] + \cdots + [r(p_m) - r(p_{m-1})].$$

Suppose $\alpha \to \beta$ is an improving flip of variable X, that is, α and β differ only on the value taken by X and $N \vDash \beta \succ \alpha$. Thus, X must be in a more preferred position in β than α, given $\text{Pa}(X) = \beta[\text{Pa}(X)](= \alpha[\text{Pa}(X)])$.

The only variables whose preference position may differ in α and β are X and the children of X, $\text{Ch}(X)$. Thus, we can deduce the following lower bound on the increase in rank, $r(\beta) - r(\alpha)$.

$$\begin{aligned}
r(\beta) - r(\alpha) = &\Bigg[AF_X(d_X+1)P_P(X = \beta[X] \mid \text{Pa}(X) = \beta[\text{Pa}(X)]) \\
&+ \sum_{Y \in \text{Ch}(X)} AF_Y(d_Y+1)P_P(Y = \beta[Y] \mid \text{Pa}(Y) = \beta[\text{Pa}(Y)]) \Bigg] \\
-&\Bigg[AF_X(d_X+1)P_P(X = \alpha[X] \mid \text{Pa}(X) = \alpha[\text{Pa}(X)]) \\
&+ \sum_{Y \in \text{Ch}(X)} AF_Y(d_Y+1)P_P(Y = \alpha[Y] \mid \text{Pa}(Y) = \alpha[\text{Pa}(Y)]) \Bigg].
\end{aligned}$$

Recall that $P_P(Y = y \mid \text{Pa}(Y) = \mathbf{z}) \in \{1/n_Y, 2/n_Y, ..., 1\}$ $\forall Y \in V, y \in \text{Dom}(Y)$, and $\mathbf{z} \in$ $\text{Dom}(\text{Pa}(Y))$. Thus, we have that

$$\begin{aligned}
r(\beta) - r(\alpha) \geq AF_X(d_X+1)&\Bigg[P_P(X = \beta[X] \mid \text{Pa}(X) = \beta[\text{Pa}(X)]) - \\
&P_P(X = \alpha[X] \mid \text{Pa}(X) = \alpha[\text{Pa}(X)]) \Bigg] + \sum_{Y \in \text{Ch}(X)} AF_Y(d_Y+1)\left[\frac{1}{n_Y} - 1\right].
\end{aligned}$$

As $P_P(X = \beta[X] \mid \text{Pa}(X) = \beta[\text{Pa}(X)]) > P_P(X = \alpha[X] \mid \text{Pa}(X) = \alpha[\text{Pa}(X)])$, we have that

$$\begin{aligned}
r(\beta) - r(\alpha) \geq & AF_X(d_X+1)\frac{1}{n_X} - \sum_{Y \in \text{Ch}(X)} AF_Y(d_Y+1)\frac{n_Y - 1}{n_Y} \\
= & L(X) > 0.
\end{aligned}$$

In order to reach o_1 from o_2, each $X \in D$ must be flipped at least once in the sequence of $m-1$ flips. We know from the above that any improving flip of X corresponds a rank increase of at least $L(X)$. Thus, as $r(o_1) - r(o_2)$ is the sum of the rank increases of each of the $m-1$ flips (each of which has been shown to produce an increase in rank), we have that $r(o_1) - r(o_2) \geq \sum_{X \in D} L(X)$. As $N \vDash o_1 \succ o_2$, we cannot have $o_1 = o_2$. Thus $D \neq \varnothing$ and so $\sum_{X \in D} L(X) > 0$ by Lemma 2.17.

Finally, to show the given lower bound is tight, we simply need to show that, for any CP-net, the bound is achieved. Let $X \in V$ be a variable with no children (at least one such variable must exist as the structure is acyclic). As X has no children, $L(X) = AF_X(d_X + 1)\frac{1}{n_X}$. Let $\mathbf{u} \in \text{Dom}(\text{Pa}(X))$ be any assignment of values to $\text{Pa}(X)$. Let $x_1 \succ \cdots \succ x_m$ be the implied preference order over X, given that $\text{Pa}(X) = \mathbf{u}$. Let $Y = V \backslash \text{Pa}(X) \cup \{X\}$. Let o_1 and o_2 be any two outcomes such that $o_1[Y] = o_2[Y]$, $o_1[\text{Pa}(X)] = o_2[\text{Pa}(X)] = \mathbf{u}$, $o_1[X] = x_1$, and $o_2[X] = x_2$. Moving from o_2 to o_1 changes the value of X only and, as $\text{Pa}(X) = \mathbf{u}$, improves the X value (according to the $\text{CPT}(X)$ rule, $\mathbf{u} : x_1 \succ \cdots \succ x_m$). Thus, $o_2 \to o_1$ constitutes an improving X flip and so $N \vDash o_2 \succ o_1$. Suppose $Y \in V$ such that $Y \neq X$. Then Y and $\text{Pa}(Y)$ take the same values in o_1 and o_2 (as no variable has X as a parent). Thus, Y is in the same preference position in both o_1 and o_2. Thus, every variable $Y \neq X$ contributes the same weight to both rank sums, $r(o_1)$ and $r(o_2)$. Therefore,

$$\begin{aligned}
r(o_1) - r(o_2) =& \sum_{Z \in V} AF_Z(d_Z + 1)P_P(Z = o_1[Z]|\text{Pa}(Z) = o_1[\text{Pa}(Z)]) \\
&- \sum_{Z \in V} AF_Z(d_Z + 1)P_P(Z = o_2[Z]|\text{Pa}(Z) = o_2[\text{Pa}(Z)]) \\
=& AF_X(d_X + 1)P_P(X = o_1[X]|\text{Pa}(X) = o_1[\text{Pa}(X)]) \\
&- AF_X(d_X + 1)P_P(X = o_2[X]|\text{Pa}(X) = o_2[\text{Pa}(X)]) \\
=& AF_X(d_X + 1)P_P(X = x_1|\text{Pa}(X) = \mathbf{u}) \\
&- AF_X(d_X + 1)P_P(X = x_2|\text{Pa}(X) = \mathbf{u}) \\
=& AF_X(d_X + 1)\frac{n_X - 1 + 1}{n_X} - AF_X(d_X + 1)\frac{n_X - 2 + 1}{n_X} \\
=& AF_X(d_X + 1)\left(\frac{n_X}{n_X} - \frac{n_X - 1}{n_X}\right) \\
=& AF_X(d_X + 1)\frac{1}{n_x} = L(X).
\end{aligned}$$

By construction, o_1 and o_2 differ only on the value of X, so $D = \{X\}$ in this case. Thus, $\sum_{Y \in D} L(Y) = L(X)$. Thus, we have shown that there is a case where $N \vDash o_1 \succ o_2$ and $r(o_1) - r(o_2) = \sum_{Y \in D} L(Y)$. That is, there is a case where the lower bound is achieved (and so it is a tight bound). As such an outcome pair can be constructed for any acyclic CP-net, our lower bound is tight for every CP-net. □

E.6 Proof of Theorem 2.24

Theorem 2.24. *Let N be a CP-net over a set of variables V, which may have indifference statements in its CPTs. Let o and o' be associated outcomes. Then,*

$$\begin{aligned}
N \vDash o \succ o' &\implies r_G(o) > r_G(o') \\
\textit{and } N \vDash o \sim o' &\implies r_G(o) = r_G(o').
\end{aligned}$$

Proof. This proof will progress similarly to the proof of Theorem 2.8 (particularly Part C). It has three main parts, A, B, and C. In Part A, we show that the special case where o and o' differ on exactly one variable is sufficient to prove both results in general. In Part B, we prove that $N \vDash o \sim o' \implies r_G(o) = r_G(o')$ holds in this special case. In Part C we prove that $N \vDash o \succ o' \implies r_G(o) > r_G(o')$ holds in this special case. Part C also consists of three parts. In Part C.1, we further simplify the sufficient special case. In Part C.2, we evaluate $r_G(o)$ and $r_G(o')$ in this special case. Finally, in Part C.3, we prove $r_G(o) > r_G(o')$.

Note that, for the entirety of this proof, P_P refers to the generalised preference position given by Definition 2.21. Also, the preference graph of N is defined as before, with the addition of undirected edges for indifference. That is, if o_1 and o_2 constitute an X flip, and $o_1[X]$ is preferred to $o_2[X]$, given the values assigned to $\text{Pa}(X)$ by both o_1 and o_2, then there is an edge $o_2 \rightarrow o_1$ in the preference graph. If the user is indifferent between $o_1[X]$ and $o_2[X]$, given the values assigned to $\text{Pa}(X)$, then there is an undirected edge between o_1 and o_2 in the preference graph.

Part A

$N \vDash o \succ o'$ holds if and only if there is a directed path $o' \rightsquigarrow o$ in the preference graph. A directed path may utilise undirected edges, but must include at least one directed edge. This means that $N \vDash o \succ o'$ holds if and only if there exists a sequence of outcomes, $o = o_1, o_2, ..., o_m = o'$, such that, for all i, o_i and o_{i+1} differ on the value of exactly one variable and either $N \vDash o_i \succ o_{i+1}$ or $N \vDash o_i \sim o_{i+1}$ (with $N \vDash o_j \succ o_{j+1}$ for at least one j).

$N \vDash o \sim o'$ holds if and only if there is a path in the preference graph between o and o' that exclusively uses undirected edges. This means that $N \vDash o \sim o'$ holds if and only if there exists a sequence of outcomes, $o = o_1, o_2, ..., o_m = o'$, such that, for all i, o_i and o_{i+1} differ on the value of exactly one variable and $N \vDash o_i \sim o_{i+1}$.

The above results imply that it is sufficient to prove that $N \vDash o \succ o' \implies r_G(o) > r_G(o')$ and $N \vDash o \sim o' \implies r_G(o) = r_G(o')$ hold in the case where o and o' differ on exactly one variable. The more general results then follow from these specific results and the transitivity of $=$ and $>$.

Let us assume that o and o' differ only on the value taken by $X \in V$. Let X take the value x in o ($o[X] = x$) and the value x' in o' ($o'[X] = x'$).

Part B

First, we show that $N \vDash o \sim o' \implies r_G(o) = r_G(o')$. We assume that $N \vDash o \sim o'$. Let $\mathbf{u} = o[\text{Pa}(X)] = o'[\text{Pa}(X)]$.

Recall that

$$r_G(o) = \sum_{Z \in V} AF_Z(d_Z + 1)P_P(Z = o[Z] \mid \mathrm{Pa}(Z) = o[\mathrm{Pa}(Z)])$$

and similarly for $r_G(o')$. Thus, to evaluate $r_G(o)$ and $r_G(o')$, and subsequently prove that $r_G(o) = r_G(o')$, we must first evaluate these summation terms for all $Z \in V$, for both $r_G(o)$ and $r_G(o')$.

Let $Y \in V$ be a variable such that $Y \neq X$ and $X \notin \mathrm{Pa}(Y)$. As o and o' differ only on the value of X, Y and $\mathrm{Pa}(Y)$ must take the same values in both o and o'. Let $y = o[Y] = o'[Y]$ and $\mathbf{w} = o[\mathrm{Pa}(Y)] = o'[\mathrm{Pa}(Y)]$. Then we have

$$\begin{aligned} &AF_Y(d_Y + 1)P_P(Y = o[Y] \mid \mathrm{Pa}(Y) = o[\mathrm{Pa}(Y)]) \\ =&AF_Y(d_Y + 1)P_P(Y = y \mid \mathrm{Pa}(Y) = \mathbf{w}) \\ =&AF_Y(d_Y + 1)P_P(Y = o'[Y] \mid \mathrm{Pa}(Y) = o'[\mathrm{Pa}(Y)]). \end{aligned}$$

Thus, any such variable (that is, any variable that is neither X itself, nor a child of X) contributes exactly the same quantity to both sums, $r_G(o)$ and $r_G(o')$.

As $N \vDash o \sim o'$, and o and o' differ only on X, we must have that $x \sim x'$ under $\mathrm{Pa}(X) = \mathbf{u}$. Therefore, x and x' are in the same preference position in the row of $\mathrm{CPT}(X)$ corresponding to $\mathrm{Pa}(X) = \mathbf{u}$. Let x and x' be in preference position i, given $\mathrm{Pa}(X) = \mathbf{u}$.

Consider the summation term contributed by X. By our assumptions about o and o', the X summation terms in $r_G(o)$ and $r_G(o')$, respectively, are

$$\begin{aligned} AF_X(d_X + 1)P_P(X = x \mid \mathrm{Pa}(X) = \mathbf{u}) &= AF_X(d_X + 1)\frac{n_X - \ell - i + 1}{n_X - \ell}, \\ AF_X(d_X + 1)P_P(X = x' \mid \mathrm{Pa}(X) = \mathbf{u}) &= AF_X(d_X + 1)\frac{n_X - \ell - i + 1}{n_X - \ell}, \end{aligned}$$

where ℓ is the number of indifferences in the $\mathrm{CPT}(X)$ preference order over $\mathrm{Dom}(X)$ corresponding to $\mathrm{Pa}(X) = \mathbf{u}$. Thus, X contributes exactly the same quantity to both sums, $r_G(o)$ and $r_G(o')$.

Finally, we must consider the weights contributed by $\mathrm{Ch}(X) = \{Y_1, ..., Y_k\}$. Let $y_j = o[Y_j] = o'[Y_j]$, $\mathbf{v}_j = o[\mathrm{Pa}(Y_j)]$, and $\mathbf{v}'_j = o'[\mathrm{Pa}(Y_j)]$. The Y_j summation term in $r_G(o)$ is $AF_{Y_j}(d_{Y_j} + 1)P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}_j)$. The Y_j summation term in $r_G(o')$ is $AF_{Y_j}(d_{Y_j} + 1)P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}'_j)$.

We know that x and x' are indifferent under $\mathrm{Pa}(X) = \mathbf{u}$. We know that $o[\mathrm{Pa}(X)] = \mathbf{u}$ and $o[\mathrm{Pa}(Y_j)] = \mathbf{v}_j$. Thus, if $\mathrm{Pa}(X)$ and $\mathrm{Pa}(Y_j)$ have a non empty intersection, $\mathbf{v}_j$ and $\mathbf{u}$ must assign these variables the same values. Similarly, $\mathbf{v}'_j$ must also assign the same values as $\mathbf{u}$ to this intersection as $o'[\mathrm{Pa}(X)] = \mathbf{u}$ and $o'[\mathrm{Pa}(Y_j)] = \mathbf{v}'_j$. Note that $\mathbf{v}_j$ and $\mathbf{v}'_j$ differ only on the value taken by X (in particular, $\mathbf{v}_j[X] = x$ and $\mathbf{v}'_j[X] = x'$). In §2.5, we discussed that, in order to ensure consistency, we assume that changing between indifferent parental assignments does not affect the user's preference over children. By this assumption and the above results, the user's preference over Y_j should be identical under $\text{Pa}(Y_j) = \mathbf{v}_j$ and $\text{Pa}(Y_j) = \mathbf{v}'_j$. This means that, under both $\text{Pa}(Y_j) = \mathbf{v}_j$ and $\text{Pa}(Y_j) = \mathbf{v}'_j$, there are the same number of indifferences in the preference order over $\text{Dom}(Y_j)$, and y_j is in the same position of preference in these preference orders. By our generalised definition of P_P (Definition 2.21), this implies that $P_P(Y_j = y_j \mid \text{Pa}(Y_j) = \mathbf{v}_j) = P_P(Y_j = y_j \mid \text{Pa}(Y_j) = \mathbf{v}'_j)$. Thus, Y_j contributes exactly the same quantity to both sums, $r_G(o)$ and $r_G(o')$.

We have now shown that all variables, $Z \in V$, contribute exactly the same quantity to both sums, $r_G(o)$ and $r_G(o')$. Thus, we must have $r_G(o) = r_G(o')$. We have therefore shown that $N \vDash o \sim o' \implies r_G(o) = r_G(o')$. The general case, where o and o' may differ on more than one variable, follows from this result and the transitivity of $=$.

Part C.1

Next, we show that $N \vDash o \succ o' \implies r_G(o) > r_G(o')$. Suppose $N \vDash o \succ o'$. Let $\mathbf{u} = o[\text{Pa}(X)] = o'[\text{Pa}(X)]$ again. Let $x_1 \succsim x_2 \succsim \cdots \succsim x_m$ be the preference ordering over $\text{Dom}(X)$, given that $\text{Pa}(X) = \mathbf{u}$. This is the row of $\text{CPT}(X)$ that corresponds to $\text{Pa}(X) = \mathbf{u}$. Suppose $x = x_p$ and $x' = x_q$, we know that $p < q$ as $o' \rightarrow o$ is an improving flip of X.

Let o_k denote the outcome that has $o_k[X] = x_k$ and, for all variables $Y \neq X$, has $o_k[Y] = o[Y](= o'[Y])$. Then $o_1, ..., o_m$ is a sequence of X flips through the values $x_1, x_2, ..., x_m$. As $\text{Pa}(X) = \mathbf{u}$ in all o_k, we know that $o_m, ..., o_1$ is a sequence of improving or indifferent flips (as $x_1 \succsim x_2 \succsim \cdots \succsim x_m$ when $\text{Pa}(X) = \mathbf{u}$). Thus, we have $N \vDash o_1 \succsim o_2 \succsim \cdots \succsim o_m$. Notice that $o = o_p$ and $o' = o_q$ for $p < q$, so we have $N \vDash o = o_p \succsim o_{p+1} \succsim \cdots \succsim o_q = o'$. Further, at least one of these $o_k \succsim o_{k+1}$ relations must be strict, $o_k \succ o_{k+1}$, as $N \vDash o \succ o'$ (not $N \vDash o \sim o'$). This shows that it is sufficient to prove that $r_G(o) > r_G(o')$ for the special case where x and x' are adjacent in the ordering $x_1, x_2, ..., x_m$, that is, $q = p + 1$. The more general case, when x and x' are not adjacent, follows from this specific case and the result $N \vDash o \sim o' \implies r_G(o) = r_G(o')$, proven above, and the transitivity of $>$ and $=$ – this can be seen via similar reasoning to that given in the proof of Theorem 2.8. Thus, we can assume that x and x' are adjacent values. This implies that x and x' are either (one of) the i^{th} and (one of) the $(i+1)^{th}$ most preferred values of X (for some i), respectively, given $\text{Pa}(X) = \mathbf{u}$, or they are in same preference position. However, x and x' cannot be in the same preference position, as then we must have $x \sim x'$ under $\mathrm{Pa}(X) = \mathbf{u}$ and, therefore, $N \vDash o \sim o'$. This is a contradiction to our assumption that $N \vDash o \succ o'$. So we may assume that x and x' are (one of) the i^{th} and (one of) the $(i+1)^{th}$ most preferred values of X, respectively, given $\mathrm{Pa}(X) = \mathbf{u}$.

Part C.2

We have now assumed that o and o' differ only on the value taken by $X \in V$, where $o[X] = x$ and $o'[X] = x'$. Further, under the values assigned to $\mathrm{Pa}(X)$, $\mathbf{u}$, by both o and o', we have assumed that x is (one of) the i^{th} most preferred value(s) of X and x' is (one of) the $(i+1)^{th}$ most preferred value(s).

In order to evaluate $r_G(o)$ and $r_G(o')$, and subsequently prove that $r_G(o) > r_G(o')$, we must first consider the individual summation terms in $r_G(o)$ and $r_G(o')$, as we did in the indifference case above.

Let $Y \in V$ be a variable such that $Y \neq X$ and $X \notin \mathrm{Pa}(Y)$. Then, by the same reasoning as in the indifference case above, Y contributes exactly the same quantity to both sums, $r_G(o)$ and $r_G(o')$. Let α denote the total contribution to $r_G(o)$ (and thus to $r_G(o')$ also) by such variables.

Now, consider the X summation terms. By our assumptions about o and o', the X summation terms in $r_G(o)$ and $r_G(o')$ (respectively) are

$$AF_X(d_X+1)P_P(X = x \mid \mathrm{Pa}(X) = \mathbf{u}) = AF_X(d_X+1)\frac{n_X - \ell - i + 1}{n_X - \ell},$$
$$AF_X(d_X+1)P_P(X = x' \mid \mathrm{Pa}(X) = \mathbf{u}) = AF_X(d_X+1)\frac{n_X - \ell - (i+1) + 1}{n_X - \ell},$$

where ℓ is the number of indifferences in the preference ordering over $\mathrm{Dom}(X)$ under $\mathrm{Pa}(X) = \mathbf{u}$, given in $\mathrm{CPT}(X)$. Note that $0 \leq \ell \leq n_X - 2$ and $1 \leq i \leq n_X - \ell - 1$.

Finally, we must consider the weights contributed by $\mathrm{Ch}(X) = \{Y_1, ..., Y_k\}$. Let $y_j = o[Y_j] = o'[Y_j]$, $\mathbf{v}_j = o[\mathrm{Pa}(Y_j)]$, and $\mathbf{v}'_j = o'[\mathrm{Pa}(Y_j)]$. The Y_j summation term in $r_G(o)$ is $AF_{Y_j}(d_{Y_j}+1)P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}_j)$. The Y_j summation term in $r_G(o')$ is $AF_{Y_j}(d_{Y_j}+1)P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}'_j)$.

Now that we know all of the summation terms, we can evaluate $r_G(o)$ and $r_G(o')$ as follows:

$$\begin{aligned} r_G(o) =& \alpha + AF_X(d_X+1)\frac{n_X - \ell - i + 1}{n_X - \ell} \\ &+ \sum_{j=1}^{k} AF_{Y_j}(d_{Y_j}+1)P_P(Y_j = y_j \mid \mathrm{Pa}(Y_j) = \mathbf{v}_j), \end{aligned}$$

$$\begin{aligned} r_G(o') =& \alpha + AF_X(d_X+1)\frac{n_X-\ell-(i+1)+1}{n_X-\ell} \\ &+ \sum_{j=1}^{k} AF_{Y_j}(d_{Y_j}+1)P_P(Y_j = y_j \mid \text{Pa}(Y_j) = \mathbf{v}'_j). \end{aligned}$$

Part C.3

By the same reasoning given in the proof of Theorem 2.8, $AF_{Y_j} = AF_X\frac{1}{n_X}\beta_j$, for some $0 < \beta_j \leq 1$.

Let $Z \in V$ be any variable, $z \in \text{Dom}(Z)$, $\mathbf{w} \in \text{Dom}(\text{Pa}(Z))$, and let ℓ be the number of indifferences in the row of CPT(Z) corresponding to $\text{Pa}(Z) = \mathbf{w}$. Then, by definition, $\frac{1}{n_Z-\ell} \leq P_P(Z = z \mid \text{Pa}(Z) = \mathbf{w}) \leq 1$. As $0 \leq \ell \leq n_Z - 1$, this means that $\frac{1}{n_Z} \leq P_P(Z = z \mid \text{Pa}(Z) = \mathbf{w}) \leq 1$. Thus, for any $1 \leq j \leq k$, we have that

$$\begin{aligned} \frac{1}{n_{Y_j}} &\leq P_P(Y_j = y_j \mid \text{Pa}(Y_j) = \mathbf{v}_j) \leq 1, \\ \frac{1}{n_{Y_j}} &\leq P_P(Y_j = y_j \mid \text{Pa}(Y_j) = \mathbf{v}'_j) \leq 1. \end{aligned}$$

Using these results, we can rewrite $r_G(o)$ and $r_G(o')$ and obtain the following inequalities:

$$\begin{aligned} r_G(o) =& \alpha + AF_X(d_X+1)\frac{n_X-\ell-i+1}{n_X-\ell} \\ &+ \sum_{j=1}^{k} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)P_P(Y_j = y_j \mid \text{Pa}(Y_j) = \mathbf{v}_j) \\ \geq& \alpha + AF_X(d_X+1)\frac{n_X-\ell-i+1}{n_X-\ell} + \sum_{j=1}^{k} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)\frac{1}{n_{Y_j}}, \end{aligned}$$

$$\begin{aligned} r_G(o') =& \alpha + AF_X(d_X+1)\frac{n_X-\ell-(i+1)+1}{n_X-\ell} \\ &+ \sum_{j=1}^{k} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)P_P(Y_j = y_j \mid \text{Pa}(Y_j) = \mathbf{v}'_j) \\ \leq& \alpha + AF_X(d_X+1)\frac{n_X-\ell-(i+1)+1}{n_X-\ell} + \sum_{j=1}^{k} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)\cdot 1. \end{aligned}$$

Recall that our aim is to prove $r_G(o) > r_G(o')$. For the purposes of contradiction, suppose that $r_G(o) \leq r_G(o')$. This implies

$$\begin{aligned}
&\alpha + AF_X(d_X+1)\frac{n_X-\ell-i+1}{n_X-\ell} + \sum_{j=1}^{k} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)\frac{1}{n_{Y_j}} \\
&\leq \alpha + AF_X(d_X+1)\frac{n_X-\ell-(i+1)+1}{n_X-\ell} + \sum_{j=1}^{k} AF_X\frac{1}{n_X}\beta_j(d_{Y_j}+1)\cdot 1 \\
\implies &(d_X+1)\frac{n_X-\ell-i+1}{n_X-\ell} + \sum_{j=1}^{k} \frac{1}{n_X}\beta_j(d_{Y_j}+1)\frac{1}{n_{Y_j}} \\
&\leq (d_X+1)\frac{n_X-\ell-i}{n_X-\ell} + \sum_{j=1}^{k} \frac{1}{n_X}\beta_j(d_{Y_j}+1)\cdot 1 \\
\implies &(d_X+1)\frac{n_X}{n_X-\ell} \leq \sum_{j=1}^{k}\beta_j(d_{Y_j}+1)\left(1-\frac{1}{n_{Y_j}}\right).
\end{aligned}$$

As $0 \leq \ell \leq n_X - 2$ and, thus, $1 \leq \frac{n_X}{n_X-\ell}$, it follows that

$$d_X + 1 \leq \sum_{j=1}^{k}\beta_j(d_{Y_j}+1)\left(1-\frac{1}{n_{Y_j}}\right).$$

From this point, we derive a contradiction in an identical manner to the proof of Theorem 2.8. Thus, we have shown that $N \vDash o \succ o' \implies r_G(o) > r_G(o')$ in the case of x and x' adjacent. The non-adjacent case follows from this specific result, the previous indifference result, and the transitivity of $>$ and $=$, as we explained above.

We have now proven the required results for o and o' that differ on exactly one variable. The more general results follow from these special cases and the transitivity of $>$ and $=$, as we argued above. □

E.7 Proof of Proposition 3.3

Proposition 3.3. *Let N be a CP-net over variables V and let o and o' be associated outcomes. Let $U \subseteq V$ denote the variables that are unimportant to the query $N \vDash o \succ o'$. As the variables in U are unimportant, we must have $o[U] = o'[U]$. Let M be the CP-net obtained by removing U from N as described above (by fixing $U = o[U]$). Let C denote the constraint $U = o[U]$ and let N_C denote the CP-net N with this additional plausibility constraint. Let o_1 and o_2 be any two outcomes associated with N that obey constraint C, that is, $o_1[U] = o_2[U] = o[U]$. Then $N_C \vDash o_1 \succ o_2$ if and only if $M \vDash o_1[V \backslash U] \succ o_2[V \backslash U]$.*

Proof. If $U = \varnothing$, then $M = N$ and C is a trivial constraint and so $N = N_C$. Thus, we have $M = N = N_C$. As $V \backslash U = V$, the result follows trivially.

Now suppose $U \neq \varnothing$. Recall that, for any CP-net, the preference $o_1 \succ o_2$ is entailed if and only if the preference graph contains the directed path $o_2 \rightsquigarrow o_1$. Let G_{N_C} denote the preference graph of N_C and G_M the preference graph of M. For any outcomes associated with N, o_1 and o_2, that obey C, we want to show that the following property holds; G_{N_C} contains a directed path $o_2 \rightsquigarrow o_1$ if and only if G_M contains a directed path $o_2[V \backslash U] \rightsquigarrow o_1[V \backslash U]$.

The outcomes associated with the CP-nets N, N_C, and M are as follows, where × denotes Cartesian product:

$$\begin{aligned}
\Omega_N &= \bigtimes_{X \in V} \text{Dom}(X), \\
\Omega_{N_C} &= \{\alpha \in \Omega_N | \alpha[U] = o[U](= o'[U])\}, \\
\Omega_M &= \bigtimes_{X \in V \backslash U} \text{Dom}(X).
\end{aligned}$$

The outcomes (nodes) in G_{N_C} are the outcomes of N that obey C. The outcomes (nodes) in G_M are exactly the G_{N_C} outcomes restricted to $V \backslash U$, as we shall show. Suppose $\alpha \in \Omega_M$. Let $\beta \in \Omega_N$ be the outcome such that $\beta[U] = o[U]$ and $\beta[V \backslash U] = \ \alpha$. As $\beta[U] = o[U]$, we have $\beta \in \Omega_{N_C}$ and, thus, α is an N_C outcome restricted to $V \backslash U$. Conversely, any $\beta \in \Omega_{N_C}$ must also be in Ω_N and so it is an assignment to all variables in V. Restricting β to $V \backslash U$ gives an assignment to $V \backslash U$ and so we have $\beta[V \backslash U] \in \Omega_M$. Thus, Ω_M is exactly the outcomes in Ω_{N_C} restricted to $V \backslash U$.

Rather than referring to o_1 and o_2 as outcomes of N that obey C, we can instead consider them as outcomes of N_C. Given such an outcome, o_1, we shall refer to $o_1[V \backslash U]$ as the *reduced form* of o_1. As we have seen above, Ω_M is the reduced forms of the outcomes in Ω_{N_C}. Outcomes of N_C are in one to one correspondence with their reduced forms. Any outcome $o_1 \in \Omega_{N_C} \subseteq \Omega_N$ specifies exactly one assignment to $V \backslash U$, this is its reduced form. If o_1 and o_2 have the same reduced form, then they agree on the assignment to $V \backslash U$. However, they must obey C, so $o_1[U] = o_2[U] = o[U]$. Thus, we have $o_1 = o_2$. This shows the outcomes in Ω_{N_C} are in one to one correspondence with their reduced forms.

It is sufficient to prove our result for edges, rather than directed paths in general. That is, there is an edge $o_2 \rightarrow o_1$ in G_{N_C} if and only if there is an edge $o_2[V \backslash U] \rightarrow o_1[V \backslash U]$ in G_M. Suppose we have proved this specific case. If there is a directed path $o_2 \rightsquigarrow o_1$ in G_{N_C}, say $o_2 = p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_m = o_1$, then by the above result, G_M contains an edge $p_i[V \backslash U] \rightarrow p_{i+1}[V \backslash U]$ for each i. These edges form a directed path $o_2[V \backslash U] \rightsquigarrow o_1[V \backslash U]$ in G_M. Now suppose there is a path $o_2[V \backslash U] \rightsquigarrow o_1[V \backslash U]$ in G_M, say $o_2[V \backslash U] = q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_k = o_1[V \backslash U]$.

As we have shown above, each q_i is the reduced form of a unique outcome in N_C, say q'_i. By the above result, the edge $q_i \to q_{i+1}$ in G_M implies the edge $q'_i \to q'_{i+1}$ is in G_{N_C}. These edges form a directed path $q'_1 \rightsquigarrow q'_k$ in G_{N_C}. As $o_2[V \backslash U]$ is the reduced form of o_2 and the q'_i are unique, we must have $o_2 = q'_1$ as $o_2[V \backslash U] = q_1$. By similar reasoning, we have $o_1 = q'_k$. Thus, we have found a directed path $o_2 \rightsquigarrow o_1$ in G_{N_C}. This proves that the general result follows from the specific edge case given above.

We now prove the specific edge case described above. Let o_1 and o_2 be any outcomes of N_C and suppose the edge $o_2 \to o_1$ is in G_{N_C}. The preference graph G_{N_C} is the induced graph of G_N on the set of outcomes (nodes) that obey C, Ω_{N_C}. Thus, if there is an edge $o_2 \to o_1$ in G_{N_C}, then there is also an edge $o_2 \to o_1$ in G_N. By preference graph definition, $o_2 \to o_1$ must be an improving flip for N. As o_1 and o_2 obey C, we must have $o_1[U] = o_2[U] = o[U]$. Thus, $o_2 \to o_1$ must be an improving flip of some $X \notin U$, that is, $X \in V \backslash U$. As o_1 and o_2 differ only on the value of X, $o_1[V \backslash U]$ and $o_2[V \backslash U]$ also differ on X only. Recall that these reductions of o_1 and o_2 are in Ω_M. As they differ only on X, $o_2[V \backslash U] \to o_1[V \backslash U]$ constitutes a variable flip in M. By preference graph definition, there must be an edge in G_M between $o_1[V \backslash U]$ and $o_2[V \backslash U]$. To prove that it is oriented $o_2[V \backslash U] \to o_1[V \backslash U]$, we must prove this to be an improving flip for M.

Suppose that X lost no parents in obtaining M (by removing variables U from N). In this case, CPT(X) is the same in both N and M, as no adjustment was required (C imposes no constraint on this CPT). As $o_2 \to o_1$ is an improving X flip in N, we know that, under the Pa(X) assignments in o_1 and o_2, $o_1[X]$ is preferred to $o_2[X]$ according to this CPT. As X lost no parents, it has the same parent set in N and M, so all of its parents are in $V \backslash U$. As $o_1[V \backslash U]$ and $o_2[V \backslash U]$ are simply restrictions of o_1 and o_2 to $V \backslash U$, they have the same assignments to Pa(X). Thus, in determining the preference order of $o_1[V \backslash U]$ and $o_2[V \backslash U]$ as an X flip in M, we consult the same row of the CPT as we did for N. Thus, we again conclude that $o_1[X]$ is preferred to $o_2[X]$ under this parental assignment (as the CPT is unchanged) and so $o_2[V \backslash U] \to o_1[V \backslash U]$ is an improving flip and, thus, an edge in G_M, as we wanted.

Now suppose that some parents of X are lost in reducing N to M. The CPT of X in M is obtained by reducing the original CPT to the rows in which $U = o[U]$ in the parental assignment. We then omit any U variables from the parental assignments as they are fixed. Let P_N denote the parent set of X in N and $P_M \subset P_N$ the reduced parent set in M. As the P_M variables are not removed, we must have $P_M \subseteq V \backslash U$. To determine whether $o_2[V \backslash U] \to o_1[V \backslash U]$ is an improving or worsening flip, we consult the row of CPT(X) in M that corresponds to the assignment of P_M in both $o_1[V \backslash U]$ and $o_2[V \backslash U]$. Note that this is the same assignment given to P_M by o_1 and o_2. The CPT row corresponding to this P_M assignment is the row from the original CPT in N that corresponds to the P_N assignment that has the same P_M assignment and fixes all $P_N \cap U = P_N \backslash P_M$ parents to their $o[U]$ values (thus, the CPTs in M give the preference rules under constraint C). Recall that o_1 and o_2 both assign $U = o[U]$ and they agree with $o_1[V \backslash U]$ and $o_2[V \backslash U]$ on the P_M assignment. Thus, the CPT row in M corresponding to the P_M assignment by $o_1[V \backslash U]$ and $o_2[V \backslash U]$ is the row in the original CPT corresponding to the P_N assignment by o_1 and o_2. Thus, as $o_2 \rightarrow o_1$ is an improving flip in N, $o_1[X]$ is preferred to $o_2[X]$ in the original CPT (given the P_N assignment). Therefore, $o_1[X]$ must be preferred to $o_2[X]$ in the CPT of M also (given the reduced P_M assignment), as we are consulting the same CPT row. Thus, again, $o_2[V \backslash U] \rightarrow o_1[V \backslash U]$ is an improving flip, and, thus, an edge in G_M. This concludes the proof that if $o_2 \rightarrow o_1$ is an edge in G_{N_C}, then $o_2[V \backslash U] \rightarrow o_1[V \backslash U]$ is an edge in G_M.

Now, to prove the other direction, let us suppose that G_M contains the edge $o_2[V \backslash U] \rightarrow o_1[V \backslash U]$. We want to prove that the edge $o_2 \rightarrow o_1$ is in G_{N_C}. As $o_1, o_2 \in \Omega_{N_C}$ and G_{N_C} is the induced graph of G_N on the outcomes in Ω_{N_C}, it is sufficient to prove that the edge $o_2 \rightarrow o_1$ is in G_N. By the definition of a preference graph, as the edge $o_2[V \backslash U] \rightarrow o_1[V \backslash U]$ is in G_M, this must be an improving flip of some $X \in V \backslash U$ (by construction, these are the variables of M). Thus, $o_1[V \backslash U]$ and $o_2[V \backslash U]$ differ only on the value of X. As o_1 and o_2 obey C, we must have $o_1[U] = o_2[U] = o[U]$ and so o_1 and o_2 differ only on the value of X. Thus, $o_2 \rightarrow o_1$ constitutes a variable flip for N and so o_1 and o_2 are connected by an edge in G_N, by the preference graph definition. To prove this edge is oriented $o_2 \rightarrow o_1$, we must prove that it is an improving flip. By the same arguments as above, the CPT(X) row in N that corresponds to the parental assignment in o_1 and o_2 must be the same as the row in the CPT(X) of M corresponding to the parental assignment in $o_1[V \backslash U]$ and $o_2[V \backslash U]$. As $o_2[V \backslash U] \rightarrow o_1[V \backslash U]$ is an improving X flip in M, we must have that $o_1[X]$ is preferred to $o_2[X]$ according to the relevant row of CPT(X). As this row is the same as the CPT(X) row in N corresponding to the o_1 and o_2 parent assignments, $o_1[X]$ must also be preferred to $o_2[X]$ in N, given the parental assignments of o_1 and o_2. Thus, as $o_2 \rightarrow o_1$ is an X flip, it is an improving flip for N. Thus, $o_2 \rightarrow o_1$ is an edge in G_N (and so in G_{N_C}), as we wanted to prove. This concludes our proof of the specific edge case of the result. The general result follows from this case, as we showed above. □

E.8 Proof of Proposition 3.4

Proposition 3.4. *Let N be a CP-net over variables V and let o and o' be any two associated outcomes. Let P denote the set of variables, Y, such that Y and* *all ancestors of Y take the same values in both o and o'. If there is an IFS $o' \rightsquigarrow o$ in N, then no variable in P is flipped in this IFS.*

Proof. If $Y \in P$, then all ancestors of Y must also be in P. Suppose there is an IFS $o' \rightsquigarrow o$ in N that, at some point, flips a variable in P. Let $P_F \neq \varnothing$ be the set of variables in P that are flipped by this IFS. Let $Y \in P_F$ have minimal ancestor set size. That is

$$Y = \text{argmin}_{X \in P_F}\{|\text{Anc}(X)|\},$$

where $\text{Anc}(X)$ denotes the set of ancestors of X. Such a variable, Y, must exist as P_F is non-empty and finite (as V is finite) and $|\text{Anc}(X)|$ are finite integers (as N is acyclic and V is finite). Suppose Z is a parent of Y. As $\text{Anc}(Z) \subsetneq \text{Anc}(Y)$ (as $Z \notin \text{Anc}(Z)$), we must have $|\text{Anc}(Z)| < |\text{Anc}(Y)|$ and, thus, $Z \notin P_F$. As $Y \in P$ implies that all ancestors (including parents) of Y are in P, this means that none of the parents of Y are flipped in this IFS.

Let $\mathbf{u} = o[\text{Pa}(Y)] = o'[\text{Pa}(Y)]$. Suppose the associated $\text{CPT}(Y)$ entry is $\mathbf{u} : y_1 \succ y_2 \succ \cdots \succ y_m$. As none of $\text{Pa}(Y)$ change value throughout the IFS, this preference order over Y is also fixed throughout the IFS. Every variable change in an IFS must be an improving flip. Thus, every flip of variable Y must be from some y_i to some y_j where $i > j$. Suppose we have $o'[Y] = y_i$. As $Y \in P_F$, Y must be changed at least once. Thus, at the end of the IFS, we must have $Y = y_j$ with $j < i$. As the IFS ends at o, we have $o[Y] = y_j \neq y_i = o'[Y]$. This is a contradiction as $Y \in P$ implies $o[Y] = o'[Y]$.

As we have derived a contradiction, our initial assumption that there exists an $o' \rightsquigarrow o$ IFS that flips a variable in P must be incorrect. That is, we have shown that all IFS $o' \rightsquigarrow o$ preserve P throughout, as we wanted. □

E.9 Proof of Proposition 3.10

Proposition 3.10. *Let N be a CP-net over variables V with structure G. Let $G_1, G_2, ..., G_m$ be the connected components of G. Let $V_i \subseteq V$ denote the variables in G_i. Let N_i be the induced sub-CP-net of N over G_i. Let o and o' be any two outcomes associated with N such that $o \neq o'$ (otherwise the dominance query is trivially false). Then we have*

$$N \vDash o \succ o' \iff \forall i \;\; (o[V_i] = o'[V_i] \vee N_i \vDash o[V_i] \succ o'[V_i]).$$

Proof. Suppose we have $N \vDash o \succ o'$. Then there exists an IFS $o' \rightsquigarrow o$. Suppose N_i satisfies $o[V_i] \neq o'[V_i]$ – there must be at least one, otherwise we have $o = o'$ (a contradiction) as they agree on all connected components. As our IFS starts at o' and ends at o, the variables in V_i must start at $o'[V_i]$ and end at $o[V_i]$. Thus, looking only at the flips of variables in V_i, the $o' \rightsquigarrow o$ IFS specifies a flipping sequence for V_i, $o'[V_i] \rightsquigarrow o[V_i]$. If we can prove that each flip in this sequence is an improving flip in N_i, then we have found an IFS $o'[V_i] \rightsquigarrow o[V_i]$ and, thus, proved $N_i \vDash o[V_i] \succ o'[V_i]$, as we wanted. Suppose that $o_1 \to o_2$ is an X flip in the $o' \rightsquigarrow o$ IFS for some $X \in V_i$. Let $\mathbf{u}$ be the assignment of values to $\mathrm{Pa}(X)$ in both o_1 and o_2. As G_i is a connected component, X has the same parent set in G and G_i. As $o_1 \to o_2$ is an improving flip, $o_2[X]$ must be preferred to $o_1[X]$, given $\mathrm{Pa}(X) = \mathbf{u}$, in the $\mathrm{CPT}(X)$ of N. By construction, X has the same CPT in N_i as N. Thus, $o_2[X]$ is also preferred to $o_1[X]$, given $\mathrm{Pa}(X) = \mathbf{u}$, in N_i. Thus, $o_1[V_i] \to o_2[V_i]$ is an improving X flip in N_i. As these are the flips we used to construct the $o'[V_i] \rightsquigarrow o[V_i]$ flipping sequence, this must be an IFS in N_i. Thus, we have $N_i \vDash o[V_i] \succ o'[V_i]$. We have shown that, if $o[V_i] \neq o'[V_i]$, then $N_i \vDash o[V_i] \succ o'[V_i]$. This is equivalent to showing that, for all i, either $o[V_i] = o'[V_i]$ or $N_i \vDash o[V_i] \succ o'[V_i]$, this proves the first direction of our equivalence.

Now suppose that, for every N_i, we have either $o[V_i] = o'[V_i]$ or $N_i \vDash o[V_i] \succ o'[V_i]$. That is, for any N_i such that $o[V_i] \neq o'[V_i]$, there exists some IFS $o'[V_i] \rightsquigarrow o[V_i]$ in N_i. We prove our result by constructing an $o' \rightsquigarrow o$ flipping sequence that we shall prove to be an IFS in N. To do this we construct the following series of flipping sequences:

$$o' = o_1 \rightsquigarrow o_2 \rightsquigarrow o_2 \cdots \rightsquigarrow o_m \rightsquigarrow o_{m+1} = o,$$
$$\text{where } o_i[V_j] = o'[V_j] \text{ for } 1 \leq j < i \text{ and } o_i[V_j] = o[V_j] \text{ for } i \leq j \leq m.$$

By definition, we have $o' = o_1$ and $o = o_{m+1}$. If $o[V_i] = o'[V_i]$, then $o_i = o_{i+1}$. In this case, we let $o_i \rightsquigarrow o_{i+1}$ be the trivial sequence with no flips. Now suppose $o[V_i] \neq o'[V_i]$ (there must be at least one such V_i as $o \neq o'$). The only difference between o_i and o_{i+1} is that $o_i[V_i] = o'[V_i]$ and $o_{i+1}[V_i] = o[V_i]$. By our assumption, $N_i \vDash o[V_i] \succ o'[V_i]$ and so N_i has an IFS $o'[V_i] \rightsquigarrow o[V_i]$. The flipping sequence $o_i \rightsquigarrow o_{i+1}$ simply performs the V_i flips dictated by this $o'[V_i] \rightsquigarrow o[V_i]$ IFS. As no variables outside of V_i are changed, these flips successfully change o_i into o_{i+1}. We have now constructed the above flipping sequence from o_1 to o_m ($o' \rightsquigarrow o$). To prove that $N \vDash o \succ o'$, it is sufficient to show that this sequence is an IFS for N.

Let N_i be such that $o[V_i] \neq o'[V_i]$, then $o_i \rightsquigarrow o_{i+1}$ is a non-trivial flipping sequence. Let $p \to p'$ be some flip in this sequence. By construction, this must be a flip of some $X \in V_i$. Further, by construction, the flip $p[V_i] \to p'[V_i]$ is one of the flips in an IFS $o'[V_i] \rightsquigarrow o[V_i]$ in N_i. Let $\mathbf{u}$ be the values taken by $\mathrm{Pa}(X)$ in p and p' (and thus in $p[V_i]$ and $p'[V_i]$, as G_i is a connected component so $X \in V_i \implies \mathrm{Pa}(X) \subseteq V_i$). As $p[V_i] \to p'[V_i]$ is an improving flip in N_i, $\mathrm{CPT}(X)$ in N_i must dictate that $p[X]$ is preferred to $p'[X]$ given $\mathrm{Pa}(X) = \mathbf{u}$. By definition, N_i and N

have the same CPT for X. Thus, $p[X]$ is preferred to $p'[X]$ given $\mathrm{Pa}(X) = \mathbf{u}$ in N also. Thus, the flip $p' \rightarrow p$ is an improving flip for N. Thus, every flip on our $o_i \rightsquigarrow o_{i+1}$ path is an improving flip. Hence, all flips in our constructed $o_1 \rightsquigarrow o_m$ sequence are improving flips for N (as $o_i \rightsquigarrow o_{i+1}$ is a trivial sequence if $o[V_i] = o'[V_i]$). We have therefore found an $o' \rightsquigarrow o$ IFS for N, proving that $N \vDash o \succ o'$. This proves the other direction of our equivalence. □

E.10 Proof of Proposition 4.6

Proposition 4.6. *Let N be a CP-net over variables V. Let $Pa(X) = U \cup \{Y\}$, where $X, Y \in V$, $U \subseteq V$, and $Y \notin U$. Suppose that Y is a degenerate parent of X. Let CPT_1 be the current $CPT(X)$ and let CPT_2 be the $CPT(X)$ obtained by removing Y as a parent, as we did in §3.2.1. Then we have*

$$S_t(CPT_2) \geq S_t(CPT_1).$$

Proof. Let $Z = U \cup \{Y\}$, $W_1 = V \backslash Z \cup \{X\}$ and $W_2 = V \backslash U \cup \{X\}$. As we showed in Appendix D.1, $S_t(\mathrm{CPT}_1) = E[I_1(\theta)]$, where

$$\theta \sim Dir(\beta_1 + d(o_1), ..., \beta_{\mathcal{O}} + d(o_{\mathcal{O}}))$$

and, if $\theta = (q_1, q_2, ..., q_{\mathcal{O}})$, then

$$I_1(\theta) = \begin{cases} 1 & \text{if } \bigwedge_{\mathbf{z}:x_1 \succ x_2 \in \mathrm{CPT}_1} \left(\sum_{\mathbf{w} \in \mathrm{Dom}(W_1)} q_{\mathbf{z}x_1\mathbf{w}} > \sum_{\mathbf{w} \in \mathrm{Dom}(W_1)} q_{\mathbf{z}x_2\mathbf{w}} \right), \\ 0 & \text{otherwise.} \end{cases}$$

In general, for indicator functions we have that $I_{A \wedge B} = I_A I_B$. For $\mathbf{z} \in \mathrm{Dom}(Z)$, suppose the corresponding rule in CPT_1 is $\mathbf{z} : x \succ \bar{x}$. Then let us define $I_{1,\mathbf{z}}$ as follows:

$$I_{1,\mathbf{z}}(\theta) = \begin{cases} 1 & \text{if } \sum_{\mathbf{w} \in \mathrm{Dom}(W_1)} q_{\mathbf{z}x\mathbf{w}} > \sum_{\mathbf{w} \in \mathrm{Dom}(W_1)} q_{\mathbf{z}\bar{x}\mathbf{w}}, \\ 0 & \text{otherwise.} \end{cases}$$

This is defined similarly if the corresponding rule in CPT_1 is $\mathbf{z} : \bar{x} \succ x$. By definition, we have

$$I_1(\theta) = \prod_{\mathbf{z} \in \mathrm{Dom}(Z)} I_{1,\mathbf{z}}(\theta).$$

As $Z = U \cup \{Y\}$, we can re-write this product as

$$I_1(\theta) = \prod_{\mathbf{u} \in \mathrm{Dom}(U)} I_{1,\mathbf{u}y}(\theta) I_{1,\mathbf{u}\bar{y}}(\theta). \tag{E.2}$$

Now consider CPT$_2$. By similar reasoning, $S_t(\text{CPT}_2) = E[I_2(\theta)]$, where

$$I_2(\theta) = \begin{cases} 1 & \text{if } \bigwedge_{\mathbf{u}:x_1 \succ x_2 \in \text{CPT}_2} \left(\sum_{\mathbf{w} \in \text{Dom}(W_2)} q_{\mathbf{u}x_1\mathbf{w}} > \sum_{\mathbf{w} \in \text{Dom}(W_2)} q_{\mathbf{u}x_2\mathbf{w}} \right), \\ 0 & \text{otherwise.} \end{cases}$$

For any $\mathbf{u} \in \text{Dom}(U)$, if the associated rule in CPT$_2$ is $\mathbf{u} : x \succ \bar{x}$, let us define $I_{2,\mathbf{u}}$ as follows:

$$I_{2,\mathbf{u}}(\theta) = \begin{cases} 1 & \text{if } \sum_{\mathbf{w} \in \text{Dom}(W_2)} q_{\mathbf{u}x\mathbf{w}} > \sum_{\mathbf{w} \in \text{Dom}(W_2)} q_{\mathbf{u}\bar{x}\mathbf{w}}, \\ 0 & \text{otherwise.} \end{cases}$$

This is defined similarly if the corresponding rule in CPT$_2$ is $\mathbf{u} : \bar{x} \succ x$. By definition, we have

$$I_2(\theta) = \prod_{\mathbf{u} \in \text{Dom}(U)} I_{2,\mathbf{u}}(\theta). \tag{E.3}$$

Let $\mathbf{u} \in \text{Dom}(U)$ and suppose the corresponding rule in CPT$_2$ is $\mathbf{u} : x \succ \bar{x}$. As Y is a degenerate parent in CPT$_1$, it must contain the rules $\mathbf{u}y : x \succ \bar{x}$ and $\mathbf{u}\bar{y} : x \succ \bar{x}$. Thus, if $I_{1,\mathbf{u}y}(\theta) = I_{1,\mathbf{u}\bar{y}}(\theta) = 1$, then we must have both

$$\sum_{\mathbf{w} \in \text{Dom}(W_1)} q_{\mathbf{u}yx\mathbf{w}} > \sum_{\mathbf{w} \in \text{Dom}(W_1)} q_{\mathbf{u}y\bar{x}\mathbf{w}}, \tag{E.4}$$

$$\sum_{\mathbf{w} \in \text{Dom}(W_1)} q_{\mathbf{u}\bar{y}x\mathbf{w}} > \sum_{\mathbf{w} \in \text{Dom}(W_1)} q_{\mathbf{u}\bar{y}\bar{x}\mathbf{w}}. \tag{E.5}$$

Now consider $I_{2,\mathbf{u}}(\theta)$.

$$\begin{aligned} \sum_{\mathbf{w} \in \text{Dom}(W_2)} q_{\mathbf{u}x\mathbf{w}} &= \sum_{\mathbf{w} \in \text{Dom}(W_1)} \sum_{y' \in \text{Dom}(Y)} q_{\mathbf{u}x\mathbf{w}y'} \\ &= \sum_{\mathbf{w} \in \text{Dom}(W_1)} q_{\mathbf{u}x\mathbf{w}y} + \sum_{\mathbf{w} \in \text{Dom}(W_1)} q_{\mathbf{u}x\mathbf{w}\bar{y}} \\ &= \sum_{\mathbf{w} \in \text{Dom}(W_1)} q_{\mathbf{u}yx\mathbf{w}} + \sum_{\mathbf{w} \in \text{Dom}(W_1)} q_{\mathbf{u}\bar{y}x\mathbf{w}}. \end{aligned}$$

Similarly, we have

$$\sum_{\mathbf{w} \in \text{Dom}(W_2)} q_{\mathbf{u}\bar{x}\mathbf{w}} = \sum_{\mathbf{w} \in \text{Dom}(W_1)} q_{\mathbf{u}y\bar{x}\mathbf{w}} + \sum_{\mathbf{w} \in \text{Dom}(W_1)} q_{\mathbf{u}\bar{y}\bar{x}\mathbf{w}}.$$

Thus, if the inequalities E.4 and E.5 are true, we must have

$$\sum_{\mathbf{w} \in \text{Dom}(W_2)} q_{\mathbf{u}x\mathbf{w}} > \sum_{\mathbf{w} \in \text{Dom}(W_2)} q_{\mathbf{u}\bar{x}\mathbf{w}},$$

which implies $I_{2,\mathbf{u}}(\theta) = 1$. We have therefore shown that $I_{1,\mathbf{u}y}(\theta) = I_{1,\mathbf{u}\bar{y}}(\theta) = 1 \implies I_{2,\mathbf{u}}(\theta) = 1$. From this, we can conclude

$$\begin{aligned}
& I_{2,\mathbf{u}}(\theta) \geq I_{1,\mathbf{u}y}(\theta) I_{1,\mathbf{u}\bar{y}}(\theta) \\
\implies & \prod_{\mathbf{u} \in \mathrm{Dom}(U)} I_{2,\mathbf{u}}(\theta) \geq \prod_{\mathbf{u} \in \mathrm{Dom}(U)} I_{1,\mathbf{u}y}(\theta) I_{1,\mathbf{u}\bar{y}}(\theta) \\
\implies & I_2(\theta) \geq I_1(\theta) \qquad \text{by Equations E.2 and E.3.}
\end{aligned}$$

Let Δ^k be the standard $k-$simplex. This is the support set for a $k+1$ dimensional Dirichlet distribution (see Appendix D.1 for details). Let f be the density function for the Dirichlet distribution of θ.

$$\begin{aligned}
& I_1(\theta) \geq I_2(\theta) \\
\implies & \underset{\theta \in \Delta^{\mathcal{O}-1}}{\int \cdots \int} I_1(\theta) f(\theta) \geq \underset{\theta \in \Delta^{\mathcal{O}-1}}{\int \cdots \int} I_2(\theta) f(\theta) \\
\implies & E[I_2(\theta)] \geq E[I_1(\theta)] \\
\implies & S_t(CPT_2) \geq S_t(CPT_1).
\end{aligned}$$

□

E.11 Proof of Proposition 4.16

Proposition 4.16. *Let N be a binary acyclic CP-net with n variables. Let E_N denote the number of distinct pairwise outcome preferences that are entailed by N. Let N_0 be the binary CP-net over n variables that has no edges in its structure. Then we must have*

$$E_N \geq E_{N_0} = 3^n - 2^n.$$

Proof. Let us begin by proving that $E_{N_0} = 3^n - 2^n$. Without loss of generality, we assume N_0 is a CP-net over variables $\{X_1, ..., X_n\}$, where each X_i has the CPT $x_i \succ \bar{x}_i$. Let o be an outcome associated with N_0. Let us define functions that identify which variables take the 'good' value in o and which take the 'bad' value:

$$\begin{aligned}
g(o) =& \{X_i | o[X_i] = x_i\}, \\
b(o) =& \{X_i | o[X_i] = \bar{x}_i\}.
\end{aligned}$$

We start by proving that $N \vDash o \succ o'$ if and only if $b(o) \subsetneq b(o')$. Suppose $b(o) \subsetneq b(o')$. Let $b(o) = \{X_1, X_2, ..., X_k\}$ and $b(o') = \{X_1, X_2, ..., X_m\}$, $m > k$ (possibly with some re-labelling). We can transform o' into o by flipping each

$X_i \in \{X_{k+1}, ..., X_m\}$ from $\bar{x}_i$ to x_i. Each of these flips changes a variable to its more preferred value (the values taken by other variables are irrelevant, as there are no parents in N_0). That is, performing these flips in any order constitutes an $o' \rightsquigarrow o$ IFS in N_0. Thus, $N_0 \vDash o \succ o'$.

Now suppose $N_0 \vDash o \succ o'$. If $b(o) = b(o')$, then $g(o) = g(o')$ also and so $o = o'$. This contradicts $N_0 \vDash o \succ o'$, thus, $b(o) \neq b(o')$. As $N_0 \vDash o \succ o'$, there is an IFS $o' \rightsquigarrow o$, $o' = o_1 \prec o_2 \prec \cdots \prec o_m = o$. If $X_i \in g(o')$, then it cannot be flipped in this sequence, regardless of what changes are made to other variables. This is because X_i is in its preferred position in o' and has no parents, thus, X_i remains in its preferred position whenever other variables are changed. Therefore, X_i cannot be changed to improve the user's preference. This means that no $X_i \in \ g(o')$ is changed in this sequence. Thus, if $o'[X_i] = x_i$, then $o[X_i] = x_i$, which implies $g(o') \subseteq g(o)$. As $b(o) = V \backslash g(o)$ and similarly for o', this implies that $b(o) \subseteq b(o')$. As we know $b(o) \neq b(o')$, we now have $b(o) \subsetneq b(o')$. We have thus proven that $N \vDash o \succ o'$ if and only if $b(o) \subsetneq b(o')$.

Let $|b(o)| = k$ for some $0 \leq k \leq n$. The set of outcomes, o', such that $N_0 \vDash o \succ \ o'$, is the set of outcome o' such that $b(o') \supsetneq b(o)$. Such o' can be identified with the subset $G \subseteq g(o)$, $G \neq \varnothing$, such that $b(o') = b(o) \cup G$. As $|g(o)| = n - |b(o)| = n - k$, there are $2^{n-k} - 1$ such subsets G. Thus, there are $2^{n-k} - 1$ outcomes o' such that $N_0 \vDash o \succ o'$.

Any outcome, o, can be fully determined by $b(o)$ as $g(o) = V \backslash b(o)$. Thus, the outcomes with $|b(o)| = k$ can be identified by $b(o)$, which is a subset of V of size k. Every subset of V corresponds to $b(o)$ for some outcome. Thus, the number of outcomes with $|b(o)| = k$ is equal to the number of subsets of V with size k. This is $\binom{n}{k}$.

From these results we can conclude the following:

$$\begin{aligned}
E_{N_0} &= |\{o \succ o' | N_0 \vDash o \succ o'\}| \\
&= \sum_{k=0}^{n} |\{o \succ o' | N_0 \vDash o \succ o' \wedge |b(o)| = k\}| \\
&= \sum_{k=0}^{n} \sum_{\substack{o \in \Omega \\ \text{s.t. } |b(o)|=k}} |\{o' | N_0 \vDash o \succ o'\}| \\
&= \sum_{k=0}^{n} \sum_{\substack{o \in \Omega \\ \text{s.t. } |b(o)|=k}} 2^{n-k} - 1 \\
&= \sum_{k=0}^{n} \binom{n}{k} (2^{n-k} - 1) = \sum_{k=0}^{n} \binom{n}{k} 2^{n-k} - \sum_{k=0}^{n} \binom{n}{k}.
\end{aligned}$$

The binomial formula states $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k}y^k$ for $n \in \mathbb{N}$. Thus, if we apply this formula to the above with $(x,y) = (2,1)$ and $(x,y) = (1,1)$, it simplifies to $E_{N_0} = 3^n - 2^n$.

We shall now demonstrate that, for any acyclic binary CP-net over n variables, N, we have $E_N \geq E_{N_0}$. We first define functions g' and b' for N that are analogous to g and b. These need to be re-defined as variables in N may have parents, so the 'good' and 'bad' values are not so easily determined – they depend on the values taken by parents:

$$g'(o) = \{X_i | o[X_i] = x_i^1 \wedge o[\text{Pa}(X_i)] : x_i^1 \succ x_i^2 \in \text{CPT}(X_i)\},$$
$$b'(o) = \{X_i | o[X_i] = x_i^1 \wedge o[\text{Pa}(X_i)] : x_i^2 \succ x_i^1 \in \text{CPT}(X_i)\}.$$

We shall now show that $b'(o) = b'(o') \implies o = o'$. Note that $b'(o) = V \backslash g'(o)$, so $b(o) = b(o')$ if and only if $g(o) = g'(o)$. Suppose $b'(o) = b'(o')$ and that $X_1, X_2, ..., X_n$ is a topological ordering of the variables according to N. That is, $\text{Pa}(X_i) \in \{X_1, ..., X_{i-1}\}$. Such an order exists as N is acyclic. As X_1 has no parents, we can assume (without loss of generality) that $\text{CPT}(X_1) = x_1 \succ \bar{x}_1$. If $X_1 \in g'(o) = g'(o')$, then by definition of g', we must have $o[X_1] = x_1 = o'[X_1]$. Similarly, if $X_1 \in b'(o) = b'(o')$, then we must have $o[X_1] = \bar{x}_1 = o'[X_1]$. Thus, $o[X_1] = o'[X_1]$. Now suppose that, for $X_i \in \{X_1, ..., X_k\}$, we have $o[X_i] = o'[X_i]$. As this is a topological ordering, we know that $o[\text{Pa}(X_{k+1})] = o'[\text{Pa}(X_{k+1})] = \mathbf{u}$. Suppose (without loss of generality) that $\text{CPT}(X_{k+1})$ in N has the rule $\mathbf{u} : x_{k+1} \succ \bar{x}_{k+1}$. If $X_{k+1} \in g'(o) = g'(o')$, then $o[X_{k+1}] = x_{k+1} = o'[X_{k+1}]$, by the definition of g'. If $X_{k+1} \in b'(o) = b'(o')$, then $o[X_{k+1}] = \bar{x}_{k+1} = o'[X_{k+1}]$, by the definition of b'. Thus, $o[X_{k+1}] = o'[X_{k+1}]$. By induction, we have shown $o[X_i] = o'[X_i]$ for all i, therefore $o = o'$.

Given any subset $B \subseteq V$, we can determine the unique outcome such that $b'(o) = B$ as follows. Note that $g'(o) = V \backslash B$. Suppose that $X_1, X_2, ..., X_n$ is a topological ordering. We go through the variables in order, so that a variable's parents are always assigned values before the variable itself. If we want $X_i \in g'(o)$, then assign $o[X_i]$ to be the preferred value in the rule of $\text{CPT}(X_i)$ corresponding to $o[\text{Pa}(X_i)]$. If we want $X_i \in b'(o)$, assign $o[X_i]$ to be the not preferred value. By construction, o satisfies the definitions of $b'(o)$ and $g'(o)$. It is unique by the above result that $b'(o) = b'(o') \implies o = o'$.

By the above results, we have shown that b' is a bijection between the outcomes and the subsets $B \subseteq V$. Thus, the number of outcomes with $|b'(o)| = k$, for $0 \leq k \leq n$, is equal to the number of subsets of V of size k. Thus, there are $\binom{n}{k}$ outcomes with $|b'(o)| = k$.

Now suppose $|b'(o)| = k$. Let $G \subseteq g'(o)$, so we have $|G| \leq n - k$. Let o' be the outcome obtained from o by flipping all variables in G. If $G \neq \varnothing$, we will show that $N \vDash o \succ o'$. Let us again assume that $X_1, ..., X_n$ is a topological order of the variables and that $G = \{X_{i_1}, ..., X_{i_m}\}$, where $i_1 < i_2 < \cdots < i_m$ and $m \geq 1$ (as we assume $G \neq \varnothing$). We will prove that $N \vDash o \succ o'$ by constructing a sequence of worsening flips from $o \rightsquigarrow o'$. Let us start by flipping X_{i_m} and move in reverse order to X_{i_1}. As $X_{i_m} \in G \subseteq g'(o)$, by the definition of g', X_{i_m} must be in its preferred position in o, according to the assignment of $\mathrm{Pa}(X_{i_m})$ in o. This means that flipping the value of X_{i_m} constitutes a worsening flip from o. Now suppose we have flipped the values of $X_{i_m}, X_{i_{m-1}}, ..., X_{i_{m-j+1}}$ successively. As G is in topological order, this means that none of $\mathrm{Pa}(X_{i_{m-j}})$, or $X_{i_{m-j}}$ itself, have been flipped thus far. Thus, $\mathrm{Pa}(X_{i_{m-j}})$ still take the same values they did in o. Thus, as $X_{i_{m-j}} \in g(o)$, $X_{i_{m-j}}$ takes its preferred value, according to the current assignment to $\mathrm{Pa}(X_{i_{m-j}})$. Therefore, flipping the value of $X_{i_{m-j}}$ constitutes a worsening flip. Hence, by induction, successively flipping $X_{i_m}, X_{i_{m-1}}, ..., X_{i_1}$ constitutes a worsening flipping sequence from o that returns o' (as described above) and so $N \vDash o \succ o'$.

Let $G_1, G_2 \subseteq g'(o)$, such that $G_1 \neq G_2$ and $G_1, G_2 \neq \varnothing$. Let o'_1 and o'_2 be the outcomes obtained from o by flipping G_1 and G_2 respectively. By the above reasoning, $N \vDash o \succ o'_1$ and $N \vDash o \succ o'_2$. Without loss of generality, there exists $X \in G_1$ such that $X \notin G_2$. As $X \in G_1$, $o'_1[X] \neq o[X]$ (as X is flipped) and, as $X \notin G_2$, $o'_2[X] = o[X]$ (as X is not flipped). Thus, $G_1 \neq G_2 \implies o'_1 \neq o'_2$.

By the above two arguments, every $G \subseteq g'(o)$, $G \neq \varnothing$, corresponds to a unique o' such that $N \vDash o \succ o'$. As $|g(o')| = |V \backslash b'(o)| = |V| - |b'(o)| = n - k$, there are $2^{n-k} - 1$ such subsets, G. Thus, for any outcome o with $|b'(o)| = k$, there are at least $2^{n-k} - 1$ distinct o', such that $N \vDash o \succ o'$.

From these results, we can conclude the following:

$$\begin{aligned}
E_N &= |\{o \succ o' | N \vDash o \succ o'\}| \\
&= \sum_{k=0}^{n} |\{o \succ o' | N \vDash o \succ o' \wedge |b'(o)| = k\}| \\
&= \sum_{k=0}^{n} \sum_{\substack{o \in \Omega \\ \text{s.t. } |b'(o)|=k}} |\{o' | N \vDash o \succ o'\}| \\
&\geq \sum_{k=0}^{n} \sum_{\substack{o \in \Omega \\ \text{s.t. } |b'(o)|=k}} 2^{n-k} - 1 \\
&= \sum_{k=0}^{n} \binom{n}{k} (2^{n-k} - 1) = \sum_{k=0}^{n} \binom{n}{k} 2^{n-k} - \sum_{k=0}^{n} \binom{n}{k} \\
&= 3^n - 2^n = E_{N_0}.
\end{aligned}$$

□

E.12 Proof of Proposition A.4

Proposition A.4. *Let $\succsim^{C_0}$ be any ordering consistent with acyclic graph G. Suppose $\succsim^{C_0}$ has ℓ levels. Let O_k denote the outcomes on level k of $\succsim^{C_0}$, for any $k \leq \ell$. Let $S = \{o \in O_{k+1} | \forall o' \in O_k,\ o \to o' \notin G\}$. Let $\succsim^{C_1}$ be the ordering obtained from $\succsim^{C_0}$ by moving some $o \in O_{k+1}$ up to level k. If level $k+1$ is now empty, the level is removed. Then $\succsim^{C_1}$ is also consistent with G if and only if $o \in S$.*

Proof. We first assume that $\succsim^{C_1}$ is obtained by moving some element of S and prove that the resulting ordering is consistent with G.

Let $o \in S$ and $o' \in O_k$. There cannot be a directed path $o' \rightsquigarrow o$ in G as this would imply $o \succ^{C_0} o'$ (as $\succsim^{C_0}$ is consistent with G) and o' is on a higher level of $\succsim^{C_0}$ than o. Suppose there exists a directed path $o \rightsquigarrow o'$ in G:

$$o = p_1 \to p_2 \to \cdots \to p_m = o'.$$

As $\succsim^{C_0}$ is consistent with G, this implies that $p_m \succ^{C_0} p_{m-1} \succ^{C_0} \cdots \succ^{C_0} p_1$. This means that each p_i is on a lower level of $\succsim^{C_0}$ than p_{i+1}. Thus, p_1 must be at least $m-1$ levels below p_m. However, as o and o' are on adjacent levels of $\succsim^{C_0}$, we must have $m = 2$ (we can not have $m = 1$ as $o \neq o'$). This means that $o = p_1 \to p_2 = o'$ is an edge (path) in G. This contradicts the fact that $o \in S$. Thus, there is no path $o \rightsquigarrow o'$ in G and so o and o' are not connected by a directed path in G.

Now let $o, o' \in O_{k+1}$ with $o \neq o'$. suppose there is a path $o \rightsquigarrow o'$ in G. If this path has length $m-1$, then there are at least $m-1$ levels between o and o' by the same reasoning as above. As $o \neq o'$, any such path must have a length of at least 1. Thus, there is at least one level between o and o'. This is a contradiction as o and o' are on the same level of $\succsim^{C_0}$. This shows that distinct outcomes in O_{k+1} are not connected by directed paths in G. Thus, if $o \in S$ and $o' \in O_{k+1} \backslash \{o\}$, then there is no directed path between o and o' in G (as $S \subseteq O_{k+1}$ and $o \neq o'$).

We have now proved that, for any $o \in S$, there is no directed path in G between o and any element of O_k or $O_{k+1} \backslash \{o\}$.

Let $o \in S$ be the outcome that we move up a level in constructing $\succsim^{C_1}$ from $\succsim^{C_0}$. The only relative positions that are changed in this construction are between o and O_k and between o and $O_{k+1} \backslash \{o\}$. The outcome o remains above all outcomes in levels $> k+1$ and below all outcomes in levels $< k$. Thus, if $a \succ^{C_0} b$ and either $a \neq o$ or $b \notin O_k \cup O_{k+1} \backslash \{o\}$ (and vice versa), then $a \succ^{C_1} b$ as the relative positions of a and b have not changed.

Suppose there exists a directed path $o_1 \rightsquigarrow o_2$ in G. As there are no paths between o and O_k or between o and $O_{k+1} \backslash \{o\}$, we must have either $o_1 \neq o$ or $o_2 \notin O_k \cup O_{k+1} \backslash \{o\}$ (and vice versa). We know that $o_2 \succ^{C_0} o_1$ as $\succsim^{C_0}$ is consistent with G. Thus, by the above argument, we must also have $o_2 \succ^{C_1} o_1$. Hence, we have shown that, for any path $o_1 \rightsquigarrow o_2$ in G, we have $o_2 \succ^{C_1} o_1$. That is, $\succsim^{C_1}$ is consistent with G.

Now suppose that we obtained $\succsim^{C_1}$ by moving some $o \in O_{k+1} \backslash S$ up to level k. As $o \notin S$, there must be some $o' \in O_k$ such that the edge $o \to o'$ is in G. As o' is on level k of $\succsim^{C_0}$, and only o is moved in the construction of $\succsim^{C_1}$, o' is also on level k of $\succsim^{C_1}$. By construction, o is also on level k of $\succsim^{C_1}$. As G contains the edge (path) $o \to o'$, if $\succsim^{C_1}$ is consistent, then we must have $o' \succ^{C_1} o$. This is a contradiction as o and o' are on the same level of $\succsim^{C_1}$. Thus, in this case, $\succsim^{C_1}$ is not consistent with G. We have now proven that $\succsim^{C_1}$ is consistent with G if and only if the moved outcome, o, is in S. □

E.13 Proof of Theorem A.9

Theorem A.9. *Let G be a graph representing user preference and let $o_1 \succ o_2$ be a preference consistent with G. Suppose $\succsim^{C_0}$ is an ordering consistent with G such that $o_2 \succ^{C_0} o_1$. Let G_1 be obtained from G by adding the edge $o_2 \to o_1$ and let $\succsim^{C_1}$ be the ordering obtained from $\succsim^{C_0}$ by applying Algorithm 5. Then $\succsim^{C_1}$ is consistent with G_1.*

Proof. As argued in Appendix A, this procedure preserves consistency with G. This is because only two action types are performed. First, moving an outcome or set of outcomes on a given level, i, to a new level directly above level i (steps **3** and **19–20**). If this leaves level i empty, then it is removed. This preserves consistency with G by Lemma A.8. The second type of action is moving improvable outcomes up to the next level, again removing any empty levels produced (step **14**). This preserves consistency with G by Proposition A.4. Therefore, as the original ordering, $\succsim^{C_0}$ is consistent with G, the produced ordering, $\succsim^{C_1}$, is also consistent with G. Thus, in order to prove that $\succsim^{C_1}$ is consistent with G_1, it is sufficient to prove that $o_1 \succ^{C_1} o_2$ (by Proposition A.2).

The first thing Algorithm 5 does (steps **2–7**) is move o_2 up to its own level (if it is on a level with multiple outcomes in $\succsim^{C_0}$. Let o_2 now be on level k and o_1 be on level ℓ, $k < \ell$. Let outcome o be on any level i such that $k < i \leq \ell$. We will show that, if there is no directed path $o \rightsquigarrow o_2$ in G, then $o \succ^{C_1} o_2$. This is proved by induction. As $o_1 \succ o_2$ is consistent with G, there cannot be a path $o_1 \rightsquigarrow o_2$ in G. As o_1 is on level ℓ, it constitutes one of the outcomes in question and so this result proves $o_1 \succ^{C_1} o_2$, as we needed.

Suppose o is on level $k+1$ after step **7** and suppose there is no $o \rightsquigarrow o_2$ path in G. The first iteration of the 'for' loop (steps **10–22**) moves all improvable outcomes on level $k+1$ as far up the ordering as possible, until they pass level k. By design, o_2 is the only outcome on level k after step **7**. We know by assumption that there is no directed path $o \rightsquigarrow o_2$ in G and, thus, G cannot contain the edge $o \to o_2$. Therefore, o is improvable and so we move it up a level to level k (step **14**). At this point, the improvable outcomes of level $k+1$ are now on level k and, thus, they are not moved up any higher. The 'if' statement (steps **18–21**) then moves them all (including o) up to a new level directly above level k. Thus, o is now on a level above o_2. The remaining iterations of the 'for' loop (steps **10–22**) move only outcomes that were on levels $k+2$ to ℓ after step **7** (either moving them between levels or moving them into a new level). Thus, the remaining procedure will not impact the relative positions of o and o_2. Thus, o will be on a level above o_2 in the resulting order, $\succsim^{C_1}$. That is, $o \succ^{C_1} o_2$, as we wanted.

For every outcome, o, that was on level i after step **7**, $k < i < j \leq \ell$, such that G does not contain a directed path $o \rightsquigarrow o_2$, we now assume that $o \succ^{C_1} o_2$. The first $j-k-1$ iterations of the 'for' loop (steps **10–22**) move the improvable outcomes of levels $k+1, k+2, ..., j-1$ (using the level numbering from immediately after step **7**) as far up the ordering as possible until they pass level k. As we discussed above, the remaining procedure will not affect the positions of these outcomes relative to o_2. Thus, by our assumption, if o is on level i after step **7**, $k < i < j \leq \ell$, and G does not contain a directed path $o \rightsquigarrow o_2$, then o must be on a level above o_2 after the first $j-k-1$ iterations of the 'for' loop.

Let us suppose that o is on level j after step **7** and that G does not contain a directed path $o \rightsquigarrow o_2$. Recall that after step **7**, o_2 is on level k. The first $j-\ k-\ 1$ iterations of the 'for' loop (steps **10–22**) move only the outcomes on levels $k+1, k+2, ..., j-1$. Suppose the first $j-k-1$ iterations have occurred. Let i_k denote the level number that o_2 is now on and let i_j denote level number of the original (after step **7**) level j. If there are levels between i_k and i_j (that is $i_j > i_k + 1$), then they must consist of outcomes, o', that were previously on levels $k+1, k+2, ..., j-1$ (after step **7**) as these are the only outcomes that have moved and were the only outcomes between levels k and j originally. Further, as any such outcomes, o', are below o_2 in the ordering, G must contain a directed path $o' \rightsquigarrow o_2$, by the above assumptions.

Suppose G contains the edge $o \to o'$ for any o' that lies on a level between i_k and i_j. Combining this edge with the $o' \rightsquigarrow o_2$ path in G implies that there is a path $o \rightsquigarrow o_2$ in G, This contradicts our assumptions about o. Thus, for any outcome, o', on a level between i_k and i_j, G does not contain the edge $o \to o'$. Level i_k only contains o_2 as any outcomes added to this level are removed at the end of each iteration of the 'for' loop by steps **17–21**. We know that the edge $o \to o_2$ is not in G as there is no directed path $o \rightsquigarrow o_2$ in G. Thus, for every outcome, o', on levels i_k to to $i_j - 1$, the edge $o \rightarrow o'$ is not in G. Thus, o is improvable with respect to all of these levels and can continue to be moved up by the 'while' loop (steps **13–16**) until it is moved up into level i_k. Once the improvable outcomes on level i_j are moved up as far as possible by this 'while' loop, those that reached level i_k (including o) are moved up to a level above i_k by the 'if' statement (steps **18–21**). Thus, o is now on a level above o_2. As before, the remaining procedure will not affect this relative position. That is, in the resulting ordering, $\succsim^{C_1}$, o will be above o_2 and, hence, $o \succ^{C_1} o_2$, as we wanted.

Thus, by induction, we have proven that, for any outcome on level i (after step **7**), where $k < i \leq \ell$, and G does not contain a directed path $o \rightsquigarrow o_2$, we have that $o \succ^{C_1} o_2$. As we argued above, this includes o_1 and so $o_1 \succ^{C_1} o_2$. We also proved above that $\succsim^{C_1}$ is consistent with G. Thus, $\succsim^{C_1}$ is consistent with G_1 by Proposition A.2.

□

Appendix F

Glossary – Additional Terminology

In this Appendix, we define additional, non-essential but relevant terms.

- **Genetic Algorithm:** A genetic algorithm is an optimisation technique inspired by the theory of evolution. The aim is to minimise (or maximise) some fitness function, f. Genetic algorithms are particularly suited to optimising f over discrete combinatorial domains. The domain over which one wants to optimise f is the total population and elements of this population are referred to as 'chromosomes'. Every chromosome is made up of the same sequence of k genes and each gene takes exactly one value (allele) in each chromosome. One can visualise a chromosome as a k length vector, each entry corresponding to a gene. Each gene takes values in some discrete set of alleles. The general template of a genetic algorithm is as follows.

  Start by selecting a (possibly random) initial set of chromosomes. This is the initial population.

  Until the termination condition is satisfied, we perform the below procedure repeatedly. The termination condition could be connected to the run time or complexity of the process. Alternatively, termination could be connected to population features such as diversity; for example, the algorithm terminates if the population has sufficiently converged in some aspect.

  Given the current population, (if certain conditions are satisfied) we first perform crossover. This process involves two steps. First, select a set of parents from the current population. This selection is usually based upon fitness (fitter chromosomes are more likely to be chosen). Second, apply some technique to combine these parents in order to generate children (new chromosomes).

We then perform mutation on the population (if certain conditions are satisfied). This involves randomly changing certain genes of some chromosomes in the population.

Crossover and mutation are repeated until sufficient new chromosomes have been produced. The next population is formed by replacing chromosomes in the previous population with these new chromosomes. (Reeves and Rowe, 2003)

- **Kripke Structure:** Let P be some set of propositional variables. A Kripke structure consists of four components, S, S_0, T, and L. S is a set of states defined by the values taken by P and $S_0 \subseteq S$ is the set of initial states. T is a binary transition relation over S, $T \subseteq S \times S$, such that every state, $s \in S$, is related to at least one other. That is, for every $s \in S$, there is some $s' \in S$ such that $(s, s') \in T$. We can consider T to represent directed edges between states. Finally, L is a labelling function, $L : S \mapsto 2^P$, that maps each $s \in S$ to the set of variables in P that are true in s. (Santhanam et al., 2010)

- **PAC-learner (for the case of CP-nets):** Defined by Chevaleyre et al. (2010), a polynomial time algorithm $\mathcal{A}$ is a PAC-learner (probably approximately correct learner) by a set of 'examples' $\mathcal{E} \subseteq \Omega \times \Omega$, for a class of CP-nets $\mathcal{C}$, if the following property holds:

  Let $\mathcal{D}$ be some distribution over $\mathcal{E}$ and let $\delta, \varepsilon \in (0, 1)$. There exists some polynomial, p, such that for any $N \in \mathcal{C}$ and any $\mathcal{D}, \varepsilon, \delta$ we have the following condition; if we let $\mathcal{A}$ have access to at least $p(|V|, 1/\delta, 1/\epsilon)$ many examples drawn randomly from $\mathcal{D}$, then with probability $\geq 1 - \delta$, the algorithm $\mathcal{A}$ returns a CP-net, M, such that, given an example (o_1, o_2) drawn randomly from distribution $\mathcal{D}$, we have

  $$\Pr((N \models o_1 \succ o_2 \text{ and } M \not\models o_1 \succ o_2) \vee (M \models o_1 \succ o_2 \text{ and } N \not\models o_1 \succ o_2)) \leq \varepsilon.$$

  Note that an example is an ordered pair of outcomes that is labelled either entailed or not entailed by N.

- **PSPACE-Complete:** PSPACE is the set of decision problems that can be solved using an amount of memory space that is polynomial in the input size. A problem, p, is PSPACE-complete if it is in PSPACE and all other PSPACE problems can be transformed into p in polynomial time. Intuitively, these are the 'hardest' problems in PSPACE. To give some perspective, PSPACE contains NP and so PSPACE-complete problems are at least as hard (or harder) than any NP or NP-complete problems.

- **Satisfiability (SAT) problem:** A SAT problem is the task of determining whether a given Boolean formula is satisfiable. In general, this is an NP-complete problem.

- **Transparent entailment:** Defined by Dimopoulos et al. (2009), an entailed preference, $o \succ o'$, is transparently entailed if the following condition holds for all $X \in V$ such that $o[X] \neq o'[X]$:

  Let $U = \text{Pa}(X)$ and $W = V \backslash \{X\} \cup U$. If $o[U] = \mathbf{u}$, $o'[U] = \mathbf{u}'$, $o[X] = x$, and $o'[X] = x'$, then there exists some $\mathbf{w} \in \text{Dom}(W)$ such that $\mathbf{wu}x \succ \mathbf{wu}x'$ or $\mathbf{wu}'x \succ \mathbf{wu}'x'$ is entailed. Note that this definition is equivalent to requiring that $\text{CPT}(X)$ contains at least one of the rules $\mathbf{u} : x \succ x'$ or $\mathbf{u}' : x \succ x'$. This means that either $\mathbf{wu}x \succ \mathbf{wu}x'$ holds for all $\mathbf{w} \in \text{Dom}(W)$ or $\mathbf{wu}'x \succ \mathbf{wu}'x'$ holds for all $\mathbf{w} \in \text{Dom}(W)$ (or both).

- **Universal set:** An (n, k) universal set is a set of n length binary vectors, such that, if you restrict to any k indices, all 2^k possible assignments are present in the set. Formally, let $U \subseteq \{0, 1\}^n$, then U is an (n, k) universal set if the following property holds:

  For any $S \subseteq \{1, 2, ..., n\}$, such that $|S| = k$, let $S = \{i_1, ..., i_k\}$ and define $U_{|_S}$ as follows:

  $$U_{|_S} = \{(u_{i_1}, u_{i_2}, ..., u_{i_k}) | (u_1, u_2, ..., u_n) \in U\}.$$

  Then we must have $\left|U_{|_S}\right| = 2^k$.

- **2-SAT problem:** A 2-SAT problem can be described as determining whether a formula, ϕ, can be satisfied, where ϕ is a conjunction of clauses that are all disjunctions between two variables. That is, ϕ has the following form:

  $$\phi = (x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge \cdots \wedge (x_k \vee x_{k+1}).$$

  Such problems can be solved in polynomial time.

References

Abbas, A. E. (2010). General decompositions of multiattribute utility functions with partial utility independence. *Journal of Multi-Criteria Decision Analysis*, 17:37–59. 2

Abbas, A. E. and Howard, R. A. (2005). Attribute dominance utility. *Decision Analysis*, 2(4):185–206. 2

Ahmed, S. and Mouhoub, M. (2019). A divide and conquer algorithm for dominance testing in acyclic CP-nets. In *Proc. of the* 31*st* *IEEE International Conference on Tools with Artificial Intelligence*, pages 392–399, OR, USA. IEEE. 37, 38

Alanazi, E. (2016). *Conditional Preference Networks: Learning and Optimization.* PhD thesis, University of Regina. 161, 164, 169

Alanazi, E., Mouhoub, M., and Mohammed, B. (2012). A preference-aware interactive system for online shopping. *Computer and Information Science*, 5(6):33–42. 5

Alanazi, E., Mouhoub, M., and Zilles, S. (2016). The complexity of learning acyclic CP-nets. In *Proc. of* 25*th* *International Joint Conference on Artificial Intelligence*, pages 1361–1367, NY, USA. 160, 161, 164, 169

Alanazi, E., Mouhoub, M., and Zilles, S. (2020). The complexity of exact learning of acyclic conditional preference networks from swap examples. *Artificial Intelligence*, 278. 161, 164

Allen, T. E. (2013). CP-nets with indifference. In *Proc. of the* 51*st* *Annual Allerton Conference on Communication, Control, and Computing*, pages 1488–1495, IL, USA. IEEE. 80, 86, 151, 152

Allen, T. E. (2016). *CP-nets: From Theory to Practice.* PhD thesis, University of Kentucky. 150, 155, 193

Allen, T. E., Goldsmith, J., Justice, H. E., Mattei, N., and Raines, K. (2017a). Uniform random generation and dominance testing for CP-nets. *Journal of Artificial Intelligence Research*, 59:771–813. 19, 33, 34, 59, 66, 69, 94, 261, 262, 263

Allen, T. E., Siler, C., and Goldsmith, J. (2017b). Learning tree-structured CP-nets with local search. In *Proc. of the 30th International Florida Artificial Intelligence Research Society Conference*, pages 8–13, FL, USA. 150, 155, 169, 193

Amor, N. B., Dubois, D., Gouider, H., and Prade, H. (2015). Possibilistic conditional preference networks. In *Proc. of the 13th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 36–46, Compiègne, France. <hal-01303854>. 2

Aydŏgan, R., Baarslag, T., Hindriks, K. V., Jonker, C. M., and Yolum, P. (2013). Heuristic-based approaches for CP-nets in negotiation. In Ito, T., Zhang, M., Robu, V., and Matsuo, T., editors, *Complex Automated Negotiations: Theories, Models, and Software Competitions*, Studies in Computational Intelligence (Volume 435), pages 113–123. Springer, NY, USA. 5

Bacchus, F. and Grove, A. (1995). Graphical models for preference and utility. In Besnard, P. and Hanks, S., editors, *Proc. of the 11th Conference on Uncertainty in Artificial Intelligence*, pages 3–10, Québec, Canada. Morgan Kaufmann. 2

Bigot, D., Fargier, H., Mengin, J., and Zanuttini, B. (2013). Probabilistic conditional preference networks. In *Proc. of the 29th Conference on Uncertainty in Artificial Intelligence*, pages 72–81, Washington, USA. 4, 57

Bistarelli, S., Fioravanti, F., and Peretti, P. (2007). Using CP-nets as a guide for countermeasure selection. In *Proc. of the 22nd Annual ACM Symposium on Applied Computing*, pages 300–304, Seoul, Korea. ACM. 4

Bistarelli, S., Montanari, U., and Rossi, F. (1997). Semiring-based constraint satisfaction and optimization. *Journal of the Association for Computing Machinery*, 44(2):201–236. 2

Booth, R., Chevaleyre, Y., Lang, J., Mengin, J., and Sombattheera, C. (2010). Learning conditionally lexicographic preference relations. In Coelho, H., Studer, R., and Wooldridge, M., editors, *Proc. of the 19th Biennial European Conference on Artificial Intelligence*, pages 269–274, Lisbon, Portugal. IOS Press. 1

Boubekeur, F., Boughanem, M., and Tamine-Lechani, L. (2007). Semantic information retrieval based on CP-nets. In *Proc. of the IEEE International Conference on Fuzzy Systems*, London, UK. IEEE. 4

Boutilier, C., Bacchus, F., and Brafman, R. I. (2001). UCP-networks: A directed graphical representation of conditional utilities. In *Proc. of the* 17^{th} *Conference on Uncertainty in Artificial Intelligence*, pages 56–64, WA, USA. 3, 21, 26

Boutilier, C., Brafman, R. I., Domshlak, C., Hoos, H. H., and Poole, D. (2004a). CP-nets: A tool for representing and reasoning with conditional *ceteris paribus* preference statements. *Journal of Artificial Intelligence Research*, 21:135–191. 1, 3, 5, 6, 9, 12, 14, 15, 19, 20, 27, 28, 29, 30, 31, 32, 33, 35, 41, 51, 52, 57, 58, 59, 61, 66, 69, 70, 79, 80, 83, 85, 86, 91, 92, 93, 96, 97, 102, 103, 104, 109, 117, 123, 142, 143, 147, 148, 149, 186, 236, 263, 293, 298

Boutilier, C., Brafman, R. I., Domshlak, C., Hoos, H. H., and Poole, D. (2004b). Preference-based constrained optimization with CP-nets. *Computational Intelligence*, 20(2):137–157. 6, 19, 20

Boutilier, C., Brafman, R. I., Hoos, H. H., and Poole, D. (1999). Reasoning with conditional ceteris paribus preference statements. In Laskey, K. and Prade, H., editors, *Proc. of the* 15^{th} *Conference on Uncertainty in Artificial Intelligence*, pages 71–80, Stockholm, Sweden. Morgan Kaufmann. 3

Bouveret, S., Endriss, U., and Lang, J. (2009). Conditional importance networks: A graphical language for representing ordinal, monotonic preferences over sets of goods. In *Proc. of the* 21^{st} *International Joint Conference on Artificial Intelligence*, pages 67–72, CA, USA. 1

Brafman, R. I. and Dimopoulos, Y. (2004). Extended semantics and optimization algorithms for CP-networks. *Computational Intelligence*, 20(2):218–245. 19

Brafman, R. I., Domshlak, C., and Shimony, S. E. (2006). On graphical modeling of preference and importance. *Journal of Artificial Intelligence Research*, 25:389–424. 4, 92, 148

Brafman, R. I. and Engel, Y. (2009). Directional decomposition of multiattribute utility functions. In Rossi, F. and Tsoukias, A., editors, *Proc. of the* 1^{st} *International Conference on Algorithmic Decision Theory, LNAI 5783*, pages 192–202, Venice, Italy. Springer. 2

Cafaro, M., Mirto, M., and Aloisio, G. (2013). Preference-based matchmaking of grid resources with CP-nets. *Journal of Grid Computing*, 11(2):211–237. 5

Châtel, P., Truck, I., and Malenfant, J. (2008). A linguistic approach for non-functional constraints in a semantic SOA environment. In Ruan, D., Montero, J., Lu, J., Martínez, L., D'hondt, P., and Kerre, E. E., editors, *Computational Intelligence in Decision and Control – Proc. of th 8th International FLINS Conference*, World Scientific Proceedings Series on Computer Engineering and Information Science, pages 889–894, Madrid, Spain. World Scientific. 4

Chevaleyre, Y., Koriche, F., Lang, J., Mengin, J., and Zanuttini, B. (2010). Learning ordinal preferences on multiattribute domains: the case of CP-nets. In Fürnkranz, J. and Hüllermeier, E., editors, *Preference Learning*, pages 273–296. Springer, Berlin, Germany. <hal-00944354>. 161, 164, 324

Cornelio, C., Goldsmith, J., Mattei, N., Rossi, F., and Venable, K. B. (2013). Updates and uncertainty in CP-nets. In Cranefield, S. and Nayak, A., editors, *AI 2013: Advances in Artificial Intelligence – Proc. of the 26th Australasian Joint Conference on Artificial Intelligence, LNAI 8272*, pages 301–312, Dunedin, New Zealand. Springer. 4

Coste-Marquis, S., Lang, J., Liberatore, P., and Marquis, P. (2004). Expressive power and succinctness of propositional languages for preference representation. In Dubois, D., Welty, C., and Williams, M.-A., editors, *Proc. of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR2004)*, pages 203–212, British Columbia, Canada. AAAI Press. 2

Dimopoulos, Y., Michael, L., and Athienitou, F. (2009). Ceteris paribus preference elicitation with predictive guarantees. In *Proc. of 21st International Joint Conference on Artificial Intelligence*, pages 1890–1895, CA, USA. 150, 151, 152, 154, 160, 163, 168, 325

Domshlak, C., Rossi, F., Venable, K. B., and Walsh, T. (2003). Reasoning about soft constraints and conditional preferences: complexity results and approximation techniques. In *Proc. of 18th International Joint Conference on Artificial Intelligence*, pages 215–220, Acapulco, Mexico. Available at `arXiv:0905.3766`. 22, 23, 25, 26, 27, 28, 46, 56, 91

Eckhardt, A. and Vojtáš, P. (2009). How to learn fuzzy user preferences with variable objectives. In *Proc. of the Joint International Fuzzy Systems Association World Congress and European Society of Fuzzy Logic and Technology Conference*, pages 938–943, Lisbon, Portugal. 162

Eckhardt, A. and Vojtáš, P. (2010). Learning user preferences for 2CP-regression for a recommender system. In *Proc. of the* 36*th* *Conference on Current Trends in Theory and Practice of Computer Science*, pages 346–357, Špindlerův Mlýn, Czech Republic. 162

Edwards, A. W. F. (1983). Pascal's problem: The 'gambler's ruin'. *International Statistical Review*, 51(1):73–79. 38

Eichhorn, C., Fey, M., and Kern-Isberner, G. (2016). CP- and OCF-networks – a comparison. *Fuzzy Sets and Systems*, 298:109–127. 2

Engel, Y. and Wellman, M. P. (2008). CUI networks: A graphical representation for conditional utility independence. *Journal of Artificial Intelligence Research*, 31:83–112. 2

Goldsmith, J., Lang, J., Truszczyński, M., and Wilson, N. (2008). The computational complexity of dominance and consistency in CP-nets. *Journal of Artificial Intelligence Research*, 33:403–432. 6, 14, 15, 19, 20, 57

Gonzales, C. and Perny, P. (2005). GAI networks for decision making under certainty. In *Proc. of the 2005 IJCAI Multidisciplinary Workshop on Advances in Preference Handling*, Edinburgh, Scotland. 2

Guerin, J. T., Allen, T. E., and Goldsmith, J. (2013). Learning CP-net preferences online from user queries. In *Proc. of the* 3*rd* *International Conference on Algorithmic Decision Theory*, pages 208–220, Brussels, Belgium. Springer. 156, 157, 160, 168, 193

Haqqani, M., Ashrafzadeh, H., Li, X., and Yu, X. (2018). Conditional preference learning for personalized and context-aware journey planning. In Auger, A., Fonseca, C. M., Lourenço, N., Machado, P., Paquete, L., and Whitley, D., editors, *Proc. of the* 15*th* *International Conference on Parallel Problem Solving from Nature*, pages 451–463, Coimbra, Portugal. Springer. 5, 155

Haqqani, M. and Li, X. (2017). An evolutionary approach for learning conditional preference networks from inconsistent examples. In Cong, G., Peng, W.-C., Zhang, W. E., Li, C., and Sun, A., editors, *Proc. of the International Conference* *on Advanced Data Mining and Applications 2017*, pages 502–515, Singapore. Springer. 150, 155, 169, 192, 193, 228

Khoshkangini, R., Pini, M. S., Rossi, F., and Tran, D. V. (2018). Constructing CP-nets from users past behaviors. In Cuzzocrea, A., Bonchi, F., and Gunopulos, D., editors, *Proc. of the CIKM 2018 Workshops Co-Located with* 27*th* *ACM International Conference on Information and Knowledge Management*, Torino, Italy. 5, 163, 165, 166, 170, 237

Koriche, F. and Zanuttini, B. (2010). Learning conditional preference networks. *Artificial Intelligence*, 174(11):685–703. 156, 159, 160, 161, 164, 169

Labernia, F., Yger, F., Mayag, B., and Atif, J. (2018). Query-based learning of acyclic conditional preference networks from contradictory preferences. *EURO Journal on Decision Processes*, 6(1–2):39–59. 150, 156, 157, 168, 193

Labernia, F., Zanuttini, B., Mayag, B., Yger, F., and Atif, J. (2017). Online learning of acyclic conditional preference networks from noisy data. In *Proc. of the* 17*th* *IEEE International Conference on Data Mining*, LA, USA. <hal-01619969>. 150, 155, 156, 157, 168, 193

Laing, K., Thwaites, P. A., and Gosling, J. P. (2019). Rank pruning for dominance queries in CP-nets. *Journal of Artificial Intelligence Research*, 64:55–107. 8, 9, 72

Lang, J. and Mengin, J. (2008). Learning preference relations over combinatorial domains. In Pagnucco, M. and Thielscher, M., editors, *Proc. of the* 12*th* *International Workshop on Non-Monotonic Reasoning*, pages 207–214, Sydney, Australia. 164, 169

Lang, J. and Mengin, J. (2009). The complexity of learning separable *ceteris paribus* preferences. In *Proc. of* 21*st* *International Joint Conference on Artificial Intelligence*, pages 848–853, CA, USA. 164

Li, M., Vo, Q. B., and Kowalczyk, R. (2011a). Efficient heuristic approach to dominance testing in CP-nets. In Tumer, K., Yolum, P., Sonenberg, L., and Stone, P., editors, *Proc. of* 10*th* *International Conference on Autonomous Agents and Multiagent Systems*, pages 353–360, Taipei, Taiwan. 25, 26, 27, 28, 29, 32, 33, 34, 35, 46, 56, 59, 69, 70, 71, 263

Li, M., Vo, Q. B., and Kowalczyk, R. (2011b). Majority-rule-based preference aggregation on multi-attribute domains with CP-nets. In Tumer, K., Yolum, P., Sonenberg, L., and Stone, P., editors, *Proc. of the* 10th *International Conference on Autonomous Agents and Multiagent Systems*, pages 659–666, Taipei, Taiwan. 5

Li, M., Vo, Q. B., and Kowalczyk, R. (2013). Penalty scoring functions for TCP-nets and its applicability in related areas. Working paper. Available at `http://www.ict.swin.edu.au/personal/myli/AI2013.pdf` [Accessed 8 March 2017]. 26, 33

Liu, J., Xiong, Y., Wu, C., Yao, Z., and Liu, W. (2014). Learning conditional preference networks from inconsistent examples. *IEEE Transactions on Knowledge and Data Engineering*, 26(2):376–390. 150, 152, 153, 154, 155, 169, 192, 193

Liu, J., Yao, Z., Xiong, Y., Liu, W., and Wu, C. (2013). Learning conditional preference network from noisy samples using hypothesis testing. *Knowledge-Based Systems*, 40:7–16. 150, 152, 153, 154, 155, 158, 192, 193

Liu, S. and Liu, J. (2019). CP-nets structure learning based on mRMCR principle. *IEEE Access*, 7:121482–121492. 150, 157, 159

Liu, X. and Truszczynski, M. (2014). Preference trees: A language for representing and reasoning about qualitative preferences. In *Proc. of the* 8th *Multidisciplinary Workshop on Advances in Preference Handling*, pages 55–60, Québec, Canada. 1

Liu, Z., Zhong, Z., Li, K., and Zhang, C. (2018a). Structure learning of conditional preference networks based on dependent degree of attributes from preference database. *IEEE Access*, 6:27864–27872. 150, 157, 158, 159

Liu, Z., Zhong, Z., Zhang, C., Yu, Y., and Liu, J. (2018b). Learning CP-nets structure from preference data streams. *IEEE Access*, 6:56716–56726. 150, 157, 158

McGeachie, M. and Doyle, J. (2004). Utility functions for ceteris paribus preferences. *Computational Intelligence*, 20(2):158–217. 24, 25, 26, 27, 28, 56

Michael, L. and Papageorgiou, E. (2013). An empirical investigation of ceteris paribus learnability. In *Proc. of* 23rd *International Joint Conference on Artificial Intelligence*, pages 1537–1543, Beijing, China. 151, 154, 168, 193

Mohammed, B., Mouhoub, M., and Alanazi, E. (2015). Combining constrained CP-nets and quantitative preferences for online shopping. In Ali, M., Kwon, Y. S., Lee, C.-H., Kim, J., and Kim, Y., editors, *Current Approaches in Applied Artificial Intelligence – Proc. of the 28th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, LNAI 9101*, pages 702–711, Seoul, Korea. Springer. 5

O'Neill, M., Vaziripour, E., Wu, J., and Zappala, D. (2016). Condensing steam: Distilling the diversity of gamer behavior. In *Proc. of the 16th Internet Measurement Conference*, pages 81–95, CA, USA. ACM. 231

Reeves, C. R. and Rowe, J. E. (2003). Genetic algorithms – principles and perspectives: A guide to GA theory. Operations Research/Computer Science Interfaces Series. Kluwer Academic Publishers, Dordrecht, Netherlands. 324

Robert, C. P. and Casella, G. (2004). Monte carlo statistical methods (second edition). Springer Texts in Statistics. Springer, NY, USA. 173, 279

Rossi, F., Venable, K. B., and Walsh, T. (2004). mCP nets: Representing and reasoning with preferences of multiple agents. In Cohn, A. G., editor, *Proc. of the 19th National Conference on Artificial Intelligence*, pages 729–734, CA, USA. AAAI Press. 3

Santhanam, G. R., Basu, S., and Honavar, V. (2010). Dominance testing via model checking. In *Proc. of 24th AAAI Conference on Artificial Intelligence*, pages 357–362, GA, USA. 35, 36, 69, 93, 324

Santhanam, G. R., Basu, S., and Honavar, V. (2016). *Representing and Reasoning with Qualitative Preferences: Tools and Applications*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, CA, USA. 19, 35

Schiex, T., Fargier, H., and Verfaillie, G. (1995). Valued constraint satisfaction problems: Hard and easy problems. In *Proc. of the 14th International Joint Conference on Artificial Intelligence*, volume 1, pages 631–637, Québec, Canada. 2

Siler, C. (2017). Learning conditional preference networks from optimal choices. Master's thesis, University of Kentucky. 162, 165, 166, 169, 170, 193

Sun, X., Liu, J., and Wang, K. (2017). Operators of preference composition for CP-nets. *Expert Systems With Applications*, 86:32–41. 27, 36, 69

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142. 151

Wang, H., Shao, S., Zhou, X., Wan, C., and Bouguettaya, A. (2016). Preference recommendation for personalized search. *Knowledge-Based Systems*, 100:124–136. 5

Wang, H., Tao, Y., Yu, Q., Tianjing, H., Xin, C., and Qin, W. (2019). Personalized service selection using conditional preference networks. *Knowledge-Based Systems*, 164:292–308. 5

Wang, H., Zhang, J., Sun, W., Song, H., Guo, G., and Zhou, X. (2012). WCP-nets: A weighted extension to CP-nets for web service selection. In Liu, C., Ludwig, H., Toumani, F., and Yu, Q., editors, *Service-Oriented Computing – Proc. of the* 10th *International Conference on Service Oriented Computing, LNCS 7636*, pages 298–312, Shanghai, China. Springer. 4

Wicker, A. W. (2006). Interest-matching comparisons using CP-nets. Master's thesis, North Carolina State University. 4

Wilson, N. (2004a). Consistency and constrained optimisation for conditional preferences. In López de Mántaras, R. and Saitta, L., editors, *Proc. of the* 16th *European Conference on Artificial Intelligence*, pages 888–892, Valencia, Spain. IOS Press. 4

Wilson, N. (2004b). Extending CP-nets with stronger conditional preference statements. In *Proc. of the* 19th *National Conference on Artificial Intelligence, AAAI*, pages 735–741, CA, USA. 3, 32, 59, 69, 93, 96, 102, 103, 109, 142