Evolving Graphs by Graph Programming

Timothy Atkinson

PhD

Computer Science
University of York

March 2020

Abstract

Graphs are a ubiquitous data structure in computer science and can be used to represent solutions to difficult problems in many distinct domains. This motivates the use of Evolutionary Algorithms to search over graphs and efficiently find approximate solutions. However, existing techniques often represent and manipulate graphs in an ad-hoc manner. In contrast, rule-based graph programming offers a formal mechanism for describing relations over graphs.

This thesis proposes the use of rule-based graph programming for representing and implementing genetic operators over graphs. We present the Evolutionary Algorithm Evolving Graphs by Graph Programming and a number of its extensions which are capable of learning stateful and stateless digital circuits, symbolic expressions and Artificial Neural Networks. We demonstrate that rule-based graph programming may be used to implement new and effective constraint-respecting mutation operators and show that these operators may strictly generalise others found in the literature. Through our proposal of Semantic Neutral Drift, we accelerate the search process by building plateaus into the fitness landscape using domain knowledge of equivalence. We also present Horizontal Gene Transfer, a mechanism whereby graphs may be passively recombined without disrupting their fitness.

Through rigorous evaluation and analysis of over 20,000 independent executions of Evolutionary Algorithms, we establish numerous benefits of our approach. We find that on many problems, Evolving Graphs by Graph Programming and its variants may significantly outperform other approaches from the literature. Additionally, our empirical results provide further evidence that neutral drift aids the efficiency of evolutionary search.

Contents

List of Figures

List of Tables

List of Acronyms

Acryonym	Definition
ADF	Automatically Defined Function
AFG	Acyclic Function Graph
ANN	Artificial Neural Network
CC	Copy/Collapse (rule-set)
CGP	Cartesian Genetic Programming
CPDAG	Completed Partially Directed Acyclic Graph
CPPN	Compositional Pattern Producing Network
DAG	Directed Acyclic Graph
DEAP	Distributed Evolutionary Algorithms in Python
DM	De Morgan's (rule-set)
DMN	De Morgan's and Negation (rule-set)
DPO	Double-Pushout
DNN	Deep Neural Network
EA	Evolutionary Algorithm
ECGP	Embedded Cartesian Genetic Programming
EGGP	Evolving Graphs by Graph Programming
ES	Evolution Strategies
FG	Function Graph
GA	Genetic Algorithm
GE	Grammatical Evolution
GNARL	GeNeralized Acquisition of Recurrent Links
GNP	Genetic Network Programming
GP	Genetic Programming
GRAPE	Graph Structured Program Evolution
GTS	Graph Transformation System
HFG	Hierarchical Function Graph

HGT	Horizontal Gene Transfer
ID	Identity (rule-set)
IQR	Interquartile Range
LGP	Linear Genetic Programming
MAD	Median Absolute Deviation
MAS	Median Average Size
MDL	Minimum Description Length
ME	Median Number of Evaluations
MF	Median Test Fitness
MIOST	Multiple Interactive Outputs in a Single Tree
MSE	Mean Square Error
NEAT	Neuroevolution of Augmenting Topologies
O-EGGP	Ordered Evolving Graphs by Graph Programming
P-GP 2	Probabilistic GP 2
PADO	Parallel Algorithm Discovery and Orchestration
PDGP	Parallel Distributed Genetic Programming
PGTS	Probabilistic Graph Transformation System
R-EGGP	Evolving Recurrent Graphs by Graph Programming
RCGP	Recurrent Cartesian Genetic Programming
RFG	Recurrent Function Graph
RNN	Recurrent Neural Network
SAAN	Subgraph Active-Active Node
SGTS	Stochastic Graph Transformation System
SGA	Simple Genetic Algorithm
SND	Semantic Neutral Drift
TFG	Typed Function Graph
TG	Type Graph
TGP	Tree-based Genetic Programming
TPG	Tangled Program Graph

Acknowledgements

I would like to acknowledge the funding provided by the Engineering and Physical Sciences Research Council (1789003), without whom this thesis would not have been possible.

I am immensely grateful for the insight, wisdom and support of my supervisors. The ideas I present in this thesis have benefited immeasurably from Detlef's intellectual rigour and Susan's apparently limitless knowledge. They have been both excellent supervisors and brilliant mentors. They have fundamentally shaped the way I view problem solving, computer science and academia. I hope that this thesis justifies their efforts.

I'd like to give thanks to my friend, Jerry Swan. He has inoculated me with healthy doses of scepticism for as long as I have known him. I am grateful for his support, sharp mind and general company, and for all the efforts he has put in to my future endeavours.

To my (mum and dad), I am thankful for the immense support and love with which you have raised me. My mum has instilled in me a life-long love of reading and knowledge, without which this would not have been possible. To her great credit, she has proof-read countless paper drafts and reports without any knowledge of computer science. My dad has taught me the value in hard work and shown me its purpose: love of family. No tribute to him would be complete without a pun; if I were to reference the two people that have encouraged me through my entire life, I would put them in Parent-Theses. I am grateful to my brother for filling my childhood with daydreams and creativity. Somewhere along the way our imagined worlds met with automata and this was the result.

Athena, words are not enough to describe how much I appreciate your love, support, patience and kindness. You, and our future, are why I work so hard. You make every day more enjoyable, and I cannot wait to see where life takes us next. Rapidash.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Some parts of this thesis have been published in journals and conference proceedings; where items were published jointly with collaborators, the author of this thesis is responsible for the material presented here. For each published item the primary author is the first listed author. These publications are:

> [7] T. Atkinson, D. Plump, and S. Stepney, "Probabilistic graph programming," in *Pre-Proc. Graph Computation Models, GCM 2017,* 2017.
>
> [8] T. Atkinson, D. Plump, and S. Stepney, "Evolving graphs by graph programming," in *Proc. European Conference on Genetic Programming, EuroGP 2018,* ser. LNCS, vol. 10781. Springer, 2018, pp.35–51.
>
> [9] T. Atkinson, D. Plump, and S. Stepney, "Probabilistic graph programs for randomised and evolutionary algorithms," in *Proc. International Conference on Graph Transformation, ICGT 2018,* ser. LNCS, vol. 10887. Springer, 2018, pp. 63–78.
>
> [10] T. Atkinson, D. Plump, and S. Stepney, "Evolving graphs with horizontal gene transfer," in *Proc. Genetic and Evolutionary Computation Conference, GECCO 2019,* ACM, 2019, pp. 968–976.
>
> [11] T. Atkinson, D. Plump, and S. Stepney, "Evolving graphs with semantic neutral drift," *Natural Computing,* 2019.
>
> [12] T. Atkinson, D. Plump, and S. Stepney, "Horizontal gene transfer for recombining graphs," *Genetic Programming and Evolvable Machines,* 2020.

1 Introduction

1.1 Motivation

Graphs, sets of nodes and interconnecting edges, are a ubiquitous data structure in computer science. They are a generalisation of many abstract ideas: circuits, computer programs, Artificial Neural Networks (ANNs), Bayesian networks, quantum circuits, various forms of automata, deep learning architectures and many other concepts that utilise underlying structure. While graphs are equivalent to bit-strings in their ability to universally represent data, it is their ability to directly and intuitively express *structure* that makes them powerful.

In many areas, it is desirable to find a graph that is an optimal solution to a given problem. However, discovering such graphs can be extremely difficult. For example, it is very likely that the problem of finding a minimal Boolean circuit that implements a given truth table cannot be solved in polynomial time [110]. Similarly, it is known that the problem of finding a Bayesian network which has a posterior probability (given data) greater than some constant value is NP-Complete [38]. With the computational intractability of finding global solutions to all instances of such problems, we often look towards methods that efficiently find approximate solutions, or efficiently find globally optimal solutions on many *real-world* problem instances.

Evolutionary Algorithms (EAs) are a family of algorithms well-suited to these tasks. An EA maintains a 'population' (set) of individuals, each representing an approximate solution to a given problem. In each iteration of the algorithm, the worst performing individuals from the population are deleted, and replaced with permutations or recombinations of the best performing individuals. The EA mimics Darwinian evolution, whereby the population takes on the role of a species and the given problem takes on the role of an environment applying selection pressure to said species thereby inducing a 'survival of the fittest' behaviour.

Taking these two notions together, it is unsurprising that a number of EAs have emerged that explicitly evolve graphs [157,189,222]. Several state-of-the-art results have been achieved

by evolutionary approaches evolving graphs. Examples of this include the optimisation of large-scale digital circuits [249], and the synthesis of deep learning architectures [153]. A particularly noteworthy case is found when considering the synthesis of graphs to optimise for multiple, often conflicting, objectives. For example, the graph-based Cartesian Genetic Programming (CGP) algorithm is a leading technique for synthesising digital circuits, also accounting for the produced solution's size, power consumption and delay [250].

However, existing techniques often use ad-hoc methods, both in the representation of graphs and in their manipulation. A meaningful example of this is found in CGP. It is often desirable to search for *acyclic* graphs, containing no loops, to discover circuits or programs which are stateless. To restrict the search to acyclic graphs, CGP imposes an ordering on nodes, such that an edge may only exist from a node to one earlier in the ordering. By modifying edges under this constraint, it is guaranteed that no cycle will be created. However, while it is quite clear that respecting a total ordering on nodes may imply acyclicity, the converse, that acyclicity implies respect for a *fixed* total ordering does not hold. Indeed, there are transformations of graphs which may preserve acyclicity without demanding such an ordering on nodes. This example, and its consequences for the efficiency of evolution, are expanded upon later.

It is the desire to design EAs working at the level of graphs with formal, correct transformations of graphs that leads us to rule-based graph programming. In graph programming, graphs are provided as inputs to programs and new graphs are produced as outputs; these programs are seen to *transform* graphs. When multiple transformations are chained together, it is possible to explicitly express the underlying manipulations of graphical structures that are conventionally understood in domain-specific ways. In particular, we work with the graph programming language GP 2 [182], which is known to be computationally complete in that it can express any computable partial function over graphs [183].

The motivation behind this thesis is, therefore, to discover new and better ways of expressing modifications over graphs through the use of graph programming, so that we can build more efficient EAs and thereby discover higher quality graphs faster. This general ambition is driven in turn by the fact that such advancements then lend themselves to searching over the various graphical domains we have already mentioned.

1.2 Thesis Aims

Taking together these outline motivations and findings of our review of literature in Chapter 2, this thesis has the following aims:

1. To extend the graph programming language GP 2 to a probabilistic variant capable of expressing probabilistic transformations of graphs necessary to implement genetic operators for evolution.
2. To investigate whether and how these probabilistic graph programs can be used to design genetic operators for learning graphs.
3. To establish the benefits of using probabilistic graph programs as genetic operators, through empirical comparisons and theoretical discussion.
4. To investigate how probabilistic graph programs can be used to implement complex domain-specific rewrites in the context of evolution.
5. To empirically study the benefits and costs of using such rewrites throughout an evolutionary process.
6. To investigate how graphs can be recombined through probabilistic graph programs.
7. To empirically study the benefits and costs of using such recombinations throughout an evolutionary process.

1.3 Thesis Contributions

Throughout this thesis, a number of contributions have been made regarding the implementation of EAs through graph programming:

1. This thesis presents P-GP 2, a probabilistic extension of the graph programming language GP 2 that allows the programmer to specify probability distributions over outcome graphs. This extension is core to the implementation of all EAs we present throughout this thesis. Further, we show that it is possible to implement several algorithms taken from graph theory, demonstrating that this contribution is more general than its use in this thesis.
2. This thesis identifies a class of graphs, named Function Graphs (FGs), which generalise digital circuits, symbolic expressions and ANNs in both stateful and stateless forms. We also discuss how these graphs can be evaluated in the context of an evolutionary

process. These graphs are sufficiently general to function as a domain in which we can design EAs.

3. This thesis presents the algorithm 'Evolving Graphs by Graph Programming' (EGGP), which is designed to evolve Acyclic Function Graphs (AFGs). This is the first EA that uses rule-based graph programming as a representation of genetic operators. We provide P-GP 2 implementations of initialisation of FGs and atomic mutation operators; transforming a function node, and redirecting an edge while preserving acyclicity. We argue that the algorithm strictly generalises the landscapes available in CGP.

4. This thesis rigorously evaluates EGGP by empirically comparing the approach to CGP and Tree-based Genetic Programming (TGP). We study the approach's ability to synthesise digital circuits across 16 problems, and the approach's ability to synthesise symbolic expressions across 14 problems. Digital circuit comparisons show that the approach significantly outperforms CGP under very similar conditions and that the difference in performance increases as problem difficulty increases. Symbolic expression comparisons are mixed, with each studied approach performing best on a subset of the studied problems. We set out a number of plausible explanations for this, and the difference between our symbolic expression comparisons and our digital circuit comparisons.

5. This thesis extends EGGP by presenting the algorithm 'Evolving Recurrent Graphs by Graph Programming' (R-EGGP), which is designed to evolve Recurrent Function Graphs (RFGs). We provide P-GP 2 implementations of initialisation of RFGs and atomic mutation operators; redirecting an edge while preserving acyclicity and redirecting an edge freely. We argue that the presented algorithm strictly generalises the landscapes available in Recurrent Cartesian Genetic Programming (RCGP).

6. This thesis rigorously evaluates R-EGGP by empirically comparing the approach to RCGP. We study the approach's ability to synthesise digital counters on 8 problems, mathematical sequences on 3 problems and generalising n-bit parity check circuits on 4 problems. We find that R-EGGP significantly outperforms RCGP on Digital counter and n-bit parity check problems under very similar conditions. We find fewer statistical differences on mathematical sequence problems.

7. This thesis extends EGGP by implementing Semantic Neutral Drift as a mechanism for accelerating search through the use of domain-specific graph rewrites. This approach builds upon existing theory in evolutionary computation that neutral drift, a process whereby individuals' genotypes can change over time without degrading fitness, aids

the evolutionary process. We design P-GP 2 programs that implement known logical equivalence laws and build these programs into our EA.

8. This thesis rigorously and extensively investigates the benefits and costs of Semantic Neutral Drift through empirical study of digital circuit synthesis. We find that in many cases, applying equivalence laws throughout an evolutionary process can significantly improve search efficiency. We also establish that there are circumstances where it is preferable to choose otherwise detrimental experimental parameters if that then facilitates the implementation of Semantic Neutral Drift.
9. This thesis extends EGGP by implementing Horizontal Gene Transfer as a mechanism for improving the performance of search through passive recombination of graphs. This concept is inspired by biological horizontal gene transfer, a natural phenomenon whereby members of a population share genetic material without reproducing. We argue that through Horizontal Gene Transfer and the mutation operators we have set out, it is possible for complex graph recombinations to arise as a by-product of the system.
10. This thesis rigorously evaluates the benefits of using Horizontal Gene Transfer throughout an evolutionary process. Experiments for synthesising symbolic expressions establish that Horizontal Gene Transfer is often beneficial, and never detrimental, in our observations. Experiments for synthesising neural networks confirm these findings, showing that EGGP and the Horizontal Gene Transfer mechanism readily extend to neuroevolution tasks.

1.4 Thesis Outline

The rest of the thesis is structured as an incremental development starting from the context in which we work, moving through the development of simple but effective EAs based on graph programming, and finally exploring complex extensions to our proposed approaches which improve performance of search. This thesis is broken down into the following chapters:

- **Chapter 2, Context**. We give overviews of graph programming and evolutionary computing. We discuss several graph-based EAs in detail and describe how the literature sets precedent and opens questions for the rest of the work in this thesis.
- **Chapter 3, Probabilistic Graph Programming**. Before the work in this thesis, the graph programming language GP 2 was exclusively non-deterministic; that is, when it was necessary to make a choice over program execution, the execution of that decision

was determined by the compiler, rather than the program. In this chapter, we set out a probabilistic extension to GP 2, named P-GP 2, which allows the programmer to explicitly specify probability distributions over outcomes. This extension is used to implement a number of classic randomised graph algorithms in order to demonstrate its general practicality. However, the main contribution of this is to have a mechanism whereby we can specify genetic operators over graphs for evolution, which we require to have reproducible probabilistic behaviours.

- **Chapter 4, Function Graphs**. This chapter is an extended discussion of representation. Here, we identify a class of graphs, FGs, which can express a variety of domains of interest, including digital circuits, symbolic expressions and ANNs. We discuss a number of examples of FGs and describe how they can be executed. We also give potential directions in which FGs could be extended to accommodate notions such as typed functions and modularity.
- **Chapter 5, Evolving Graphs by Graph Programming.** This chapter sets out an EA for evolving AFGs with mutation operators defined as P-GP 2 programs. An initialisation program combined with a start graph is used to generate the initial population. We give atomic edge and node mutations with arguments for their correctness. The landscape we are inducing is shown to generalise that of CGP. Finally, we suggest a number of extensions to EGGP, so that it would be able to handle the typed and modular FGs described in Chapter 4.
- **Chapter 6, Benchmarking EGGP.** To evaluate our approach, we use various benchmark problems drawn from the literature. On digital circuit synthesis problems, we see remarkable improvements in performance when using our proposed approach, EGGP, instead of CGP. Additional experiments suggest that this is a result of the generalised landscape described in Chapter 5. We see less significant differences on symbolic regression problems when comparing to both CGP and TGP. We set out plausible explanations for this. Finally, we propose further problems and experimental conditions for benchmarking the approach and its variants.
- **Chapter 7, Evolving Recurrent Graphs by Graph Programming.** This chapter sets out an extension to EGGP that supports RFGs, termed R-EGGP. An initialisation program combined with a start graph is used to generate the initial population. We give two atomic edge mutations; one which preserves acyclicity and one which mutates an edge freely. The landscape we are inducing is shown to generalise that of RCGP. We

perform a number of empirical comparisons with RCGP on digital counter synthesis, mathematical sequence synthesis and n-bit parity check synthesis tasks. We find that R-EGGP outperforms RCGP on many problems, and in particular on the harder digital counter tasks.

- **Chapter 8, Evolving Graphs with Semantic Neutral Drift.** Neutral drift is a well-studied field of evolutionary computation. In this chapter, we show that P-GP 2 can be used to effectively implement known equivalence transformations, thereby gaining access to a new type of neutral drift based on domain knowledge of semantic equivalence. This general idea is referred to as Semantic Neutral Drift. We demonstrate this concept by applying logical equivalence laws to digital circuits throughout evolutionary runs. Empirical studies demonstrate that this can often improve the efficiency of search, and there are even cases where it is preferable to choose detrimental evolutionary parameters if that then permits access to Semantic Neutral Drift.
- **Chapter 9, Evolving Graphs with Horizontal Gene Transfer.** Crossover of graphs is a difficult task, and despite much research, no universal answer has been found. In this chapter, we propose a new form of graph combination, inspired by biological horizontal gene transfer observed in nature, whereby the active components of FGs are shared *between* members of a population without producing a child. By copying active components of a donor into inactive components of recipients, we gain access to a passive form of recombination that does not disrupt elitism and is without detriment to the fitness of either the donor or the recipient. Complex recombinations can arise as combinations of this mechanism and the mutation operators we have described in earlier chapters. Empirical study on symbolic regression problems finds many instances where Horizontal Gene Transfer aids performance and none where it is detrimental to performance. In general, EGGP equipped with Horizontal Gene Transfer outperforms both TGP and CGP on symbolic regression problems, whereas 'vanilla' EGGP does not as discussed in Chapter 6. We further show that EGGP with Horizontal Gene Transfer can be readily extended to neuroevolution tasks, and consistently improves performance therein, demonstrating that this technique is useful across domains.
- **Chapter 10, Conclusions and Future Work.** In this chapter, we summarise the findings of this thesis. We consider the intersections of the findings of our technical contributions and reflect on our thesis aims. We propose several areas for future work, including various application domains, an approach to modularity based on hierarchical graphs and a potential avenue toward higher-order learning of genetic operators.

2 Context

Abstract

In this chapter, we review the context of the rest of this thesis. We provide a detailed tutorial on graph transformation and graph programming. We also describe probabilistic approaches to graph transformation in the literature as these are particularly relevant to our work on probabilistic graph programming. We give a broad overview of evolutionary computation, describing major families of Evolutionary Algorithms (EAs). Finally, we give a detailed review of EAs which use graphs as a representation.

2.1 Introduction

This chapter sets out the context of the rest of the thesis. Its intention is to provide sufficient background for the reader to understand both the evolutionary and graph programming aspects of later technical chapters. The author is aware that it is unlikely that the reader is familiar with *both* of these topics as they are generally distinct disciplines within computer science. Hence this chapter is written with that consideration in mind and is designed to introduce a reader to both topics under no assumptions of prior knowledge.

The reader with no experience in graph programming will find a tutorial progressively moving from basic definitions of graphs and graph transformations (the atomic computational unit of graph programming) to examples of simple graph programs. The reader unfamiliar with evolutionary computation will find a brief and concise literature review on evolutionary computation that rapidly moves from high-level concepts to the specific families of algorithms we concern ourselves within this thesis.

This chapter is divided into the following sections:

- Section 2.2: **Graph Programming**. In this section we, give an introduction to graph programming moving from the basic notions of graphs and graph morphisms to the graph programming language GP 2 and simple graph programs.
- Section 2.3: **Evolutionary Computation**. In this section, we give an overview of evolutionary computation.
- Section 2.4: **Graphs in Evolutionary Computation**. This section may be viewed as a general 'related work' section which we will refer back to throughout the thesis. We cover a number of EAs which utilize graphs as a representation.
- Section 2.5: **Conclusions and Direction for Research**. In this section, we summarise our findings and describe how the literature we have covered sets context for the rest of this thesis.

If the interested discipline-hopping reader desires to know more about a given topic, then there are a number of helpful resources available:

- *Fundamentals of Algebraic Graph Transformation* by Ehrig, Ehrig, Taentzer and Prange [61] explains graphs, graph transformation and the Double-Pushout (DPO) approach in great detail. Later chapters on category theory and confluence may be more technical but are perhaps out of scope of the content of this thesis regardless.

- *GP 2: Efficient Implementation of a Graph Programming Language* by Bak [15] offers a detailed discussion of the graph programming language GP 2 and its implementation.
- *Essentials of Metaheuristics* by Luke [141] is a gentle introduction to metaheuristics, which encompasses evolutionary computation, and describes a number of EAs.
- *A Field Guide to Genetic Programming* by Poli, Langdon and McPhee [191] is an intuitive textbook on genetic programming.

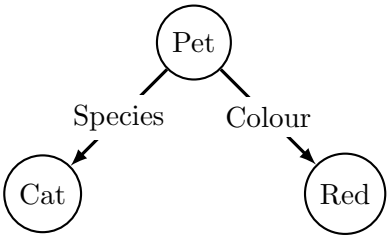


Figure 2.1: A simple string-labelled graph representing a database object of a red, pet cat; a particular pet is of the cat species and is coloured red.

2.2 Graph Programming

This section introduces the notion of graph programming and will primarily focus on the graph programming language GP 2 [182]. While detailed discussion of this is left for the dedicated Section 2.2.3, the reader may benefit from understanding *why* this section takes its given form. GP 2 uses a special construct, *rule schema*, which allows variable-based computation over graph labels. This is significant as, unlike in traditional graph transformation, it allows the programmer to access and manipulate data types with infinite possible values, such as integers. GP 2 depends on the DPO approach to graph transformation, and for this reason, the DPO approach is described in detail rather than other approaches.

This section is divided as follows. Firstly, in Section 2.2.1, a basic notion of graphs and graph rewriting is presented, leading to the DPO approach discussed in Section 2.2.2. Section 2.2.3 gives a description of the graph programming language GP 2, and finally, Section 2.2.4 describes probabilistic approaches to graph transformation that are absent from GP 2.

2.2.1 Graphs and Graph Transformation

Graph transformation is a computational abstraction that allows computing on graphs by matching and updating patterns within a graph. These computations are naturally non-deterministic: there may be multiple matches for a pattern within a graph, in which case graph transformation *does not specify* which path of computation should be taken. As will be seen, this setting allows the formulation of complex non-deterministic processes over data structures which would otherwise be difficult to traverse and manipulate.

A graph is a form of data representation consisting of a set of nodes and a set of edges which connect those nodes. Edges may connect any node to any other node, allowing complex data topologies, such as cyclic and connected graphs. Depending on context, nodes and edges may

then be labelled with data, creating a single structure for representing both concrete data and its topology, free from the more limiting topological constraints of structures such as trees or lists. The definition of a graph may vary, so here, graphs are assumed to be *labelled* and edges are *directed* (from a *source* node to a *target* node, indicated by the direction of each arrow in Figure 2.1). *Parallel* (multiple edges with common source and target nodes) and *looping* (where the source node is also the target node) edges are also allowed. Figure 2.1 shows a simple graph representing a database object of a pet cat. For intuition, a formal definition for a simple unlabelled graph is given in Definition 1. An unlabelled graph is a straightforward concept; it simply consists of a set of nodes and a set of edges between those nodes. An edge is described by its source node and its target node, and this is defined using a *source function* and a *target function*:

Definition 1 (Unlabelled Graph)**.** *An unlabelled graph $G = (V, E, s, t)$ consists of a finite set of nodes V, a finite set of edges E, as well as source function $s : E \to V$ and target function $t : E \to V$, which associate each edge with a source and target node respectively.*

The symbol $\mathcal{G}$ is used to represent the set of all possible graphs in this context, considered up to isomorphism. In this work, we only consider finite graphs, although should infinite graphs with potentially infinite node sets and edge sets be necessary they can be introduced.

With the ability to query and edit nodes and edges within a graph, a programmer may devise a problem specific transformation in an imperative language but this approach struggles to describe rewrites of complex subgraphs. In contrast, graph transformation offers an avenue towards intuitive and abstract graph rewriting. By updating graphs according to patterns, described using *rules*, it is possible to describe input-output relations on graphs that execute according to a given input graph's internal structure and data values. We consider here the DPO approach in Section 2.2.2 as this is the approach used in GP 2 and is the most commonly used approach more generally. However other approaches such as single-pushout [140] and sesqui-pushout [45] exist.

It is worth noting the obvious similarities between graph transformation and other rule-based approaches to computation, such as formal languages [151]. A particular relationship exists with L-systems [197], as both techniques use pattern-matching and rule-based rewrites to transform structure. However, L-systems are in general spacial, rather than relational, and are typically executed by applying all matches for all rules in parallel; a process that is not in general possible in graph transformation due to the existence of rule-sets with critical

pairs (see [62]).

2.2.2 Double-Pushout Approach

The DPO approach to graph transformation is an intuitive abstraction for manipulating graphs. This section is based on a number of works; the helpful monographs [61, 62], the dedicated chapter for the DPO approach in [46] and the original publication of the approach [63], but these are not referenced throughout the text as we intend this section to be a brief tutorial on the topic for a graph programmer, rather than a theoretician. For graph transformation with relabelling, this text uses the theory set out in [86] and developed for rule-based graph programming in [185, 224], although the underlying principles of natural pushouts and pullbacks are left for the reader. We firstly discuss this in the context of unlabelled graphs before introducing the concept of relabelling.

This section relies heavily on the following definitions of *graph morphisms* and particularly *injective graph morphisms*. An injective graph morphism can be understood as a function that maps elements of one graph to elements of another while preserving structure and without merging any two elements.

Definition 2 (Graph Morphism)**.** *A graph morphism $f : G \to H$ is a mapping from graph $G = (V_G, E_G, s_G, t_G)$ to graph $H = (V_H, E_H, s_H, t_H)$ that preserves structure. f consists of two functions: a node mapping $f_V : V_G \to V_H$ and an edge mapping $f_E : E_G \to E_H$. The following conditions must hold:*

1. *f preserves sources ($f_V \circ s_G = s_H \circ f_E$).*
2. *f preserves targets ($f_V \circ t_G = t_H \circ f_E$).*

Definition 3 (Injective Graph Morphism)**.** *A graph morphism $f : G \to H$ is an injective graph morphism when node mapping f_V and edge mapping f_E are injective.*

Additionally, we will use the notion of an inclusion: rules in the DPO approach consist of pairs of inclusions, where an inclusion maps each member of a subset to itself in a superset of that subset.

Definition 4 (Inclusion)**.** *A function $f : A \to B$ is an inclusion when $A \subseteq B$ and $\forall x \in A, f(x) = x$.*

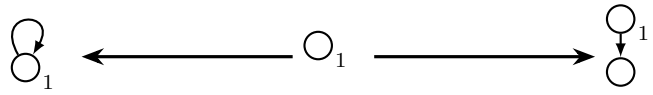


Figure 2.2: A simple DPO rule $r : L \leftarrow K \rightarrow R$. This rule matches a node with a looping edge, deletes that looping edge, and inserts a new node with an edge from the original node to that new node. The identifiers of nodes, indicated by integers to the bottom-right of each node, are used to visualise the inclusions $K \rightarrow L$ and $K \rightarrow R$.

DPO approach for unlabelled graphs

The key construct of the DPO approach is the rule. A rule r is given by

$$r = L \leftarrow K \rightarrow R \tag{2.1}$$

where L, K and R are unlabelled graphs and $K \rightarrow L$ and $K \rightarrow R$ are inclusions. The intuition is that L is a pattern to match which will then be rewritten to R. Once a match for L has been found, K describes the elements of that match which will not be deleted, and R describes new elements to add to the match. Informally, a rule is applied to a graph by matching a subgraph, deleting some elements of that subgraph and inserting some new elements to that subgraph. Figure 2.2 shows a simple rule for transforming unlabelled graphs.

A rule r is applied to a graph G using a graph morphism g to produce some new graph H. This graph transformation is denoted $G \Rightarrow_{r,g} H$. The graph morphism $g : L \rightarrow G$ is a mapping from r's L graph to the graph G, describing the match for L to apply r to. A rule r is applied as follows:

1. Choose an (injective) morphism $g : L \rightarrow G$ that satisfies the dangling condition (see below)
2. Delete elements according to r. This is done by deleting the elements of $g(L - K)$ from G.
3. Add elements according to r. This is done by adding elements of $R - K$ to G. Should an edge $e \in R - K$ be added that uses a previously existing node $n \in K$ as a source, the edge added to G uses $g(n)$ as its source; the same holds if n is used as a target.

A graph morphism g is typically injective in the DPO approach and consistently injective in the context of GP 2. One reason for this is that matching via injective morphisms is

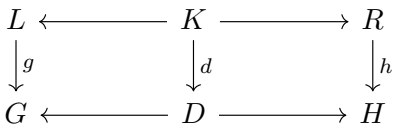


Figure 2.3: The commutative diagram formed by the application of a DPO rule $r = L \leftarrow K \rightarrow R$ applied to graph G producing intermediate graph D and then resultant graph H.

more expressive [85]; some behaviour describable with rules applied with injective morphisms cannot be simulated with non-injective morphisms, for example producing the set of all loop-free graphs. Conversely, the behaviour of a rule $r = L \leftarrow K \rightarrow R$ applied with non-injective morphisms can be simulated with a finite rule-set $Q(r)$, describing the possible merges of items in L, applied with injective morphisms.

This process of deletion and addition of graph elements according to a morphism can be shown to give rise to the commutative diagram of graph morphisms shown in Figure 2.3. The two squares are pushouts in the category of graphs in the sense of category theory. A direct derivation can be constructed by constructing a pushout complement of graph morphisms $K \rightarrow L$ and $L \rightarrow_g G$ (deleting elements) and then constructing a pushout of $K \rightarrow R$ and $K \rightarrow_d D$ (inserting elements). However, the reader may not need to understand the underlying category theory aspects to understand the intuition of the approach or the given diagram. Across the top, the component graphs of the rule applied are given. Morphism g maps of elements of L into G, and across the bottom, $G \leftarrow D \rightarrow H$ describes the transformation of G into H according to r and g.

As an example, consider the simple rule r given in Figure 2.2 applied to some graph, visualised as a commutative diagram in Figure 2.4. This rule matches a node with a looping edge, deletes that looping edge, and inserts a new node with an edge from the original node to that new node. Applying this to the given graph, there are two nodes with loops (with identifiers 2 and 4) where r could be applied, and so there are two possible morphisms to choose from. Choosing g, where $g(1) = 2$ such that node 2 is matched, gives a transformation where the loop on node 2 is removed and a new node is inserted with an edge from node 2 to that new node.

The example given in Figure 2.4 highlights the non-determinism of this approach. To apply

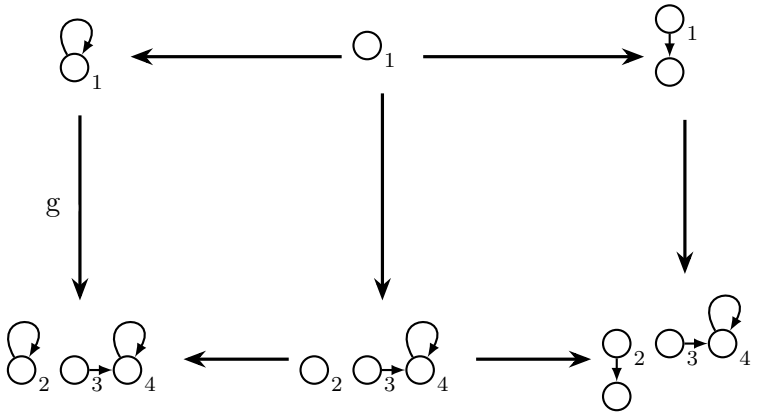


Figure 2.4: A simple DPO rule application $G \Rightarrow_{r,g} H$. Morphism g matches node 1 of r to node 2 of G, with the looping edge on 1 matched to the looping edge on 2.

r a choice must be made over the two possible matches. Should the possible matches share common elements, and some of those elements be deleted by either transformation, then the choice of match would then "destroy" the other match, so this choice can effectively decide a path of computation.

Additionally, the DPO approach uses the dangling condition described in Definition 5. In informal terms, the dangling condition guarantees that any node deleted is not the source or target or any edge that is not deleted; no edge is left "dangling" when its source or target is removed. This condition limits which morphisms are available, such that if a morphism g would act as a match for r except that it fails the dangling condition, g is not considered a valid match for r.

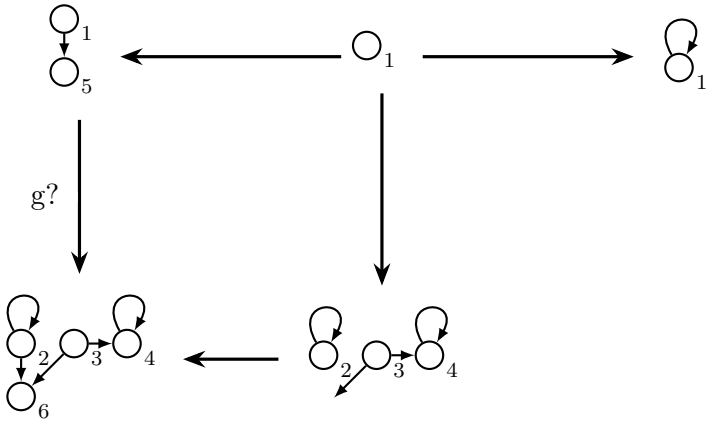


Figure 2.5: An attempted transformation that fails the dangling condition. For the rule $r = L \leftarrow K \rightarrow R$ and graph G, we have morphism g with $g(1) = 2$ and $g(5) = 6$. When the transformation is attempted, the deletion of node 6 leaves an edge with no target, and the resultant object clearly is not a graph.

Definition 5 (Dangling Condition)**.** *A graph morphism $g : L \rightarrow G$ for a rule $r = L \leftarrow K \rightarrow R$ and graph $G = (V, E, s, t)$ satisfies the dangling condition if no edge in $G - g(L - K)$ is incident to a node in $g(L - K)$.*

To see why the dangling condition is required, consider the diagram in Figure 2.5. The morphism g matches node 5 to node 6. If we attempt to construct the pushout complement thereby deleting node 6, we produce an object that is not a graph (there is an edge with no defined target), and clearly this cannot work as a valid transformation. Given a rule $L \leftarrow K \rightarrow R$ and graph morphism $g : L \rightarrow G$, graph D (as shown in Figure 2.3) only exists if g satisfies the dangling condition. Moreover, in this circumstance D is uniquely determined up to isomorphism.

Of course, these rules are not used alone or in single steps. A *Graph Transformation System* (GTS) is a finite set of rules $\mathcal{R} = \{r_1, r_2, ..., r_n\}$. GTSs are also referred to as *rule-sets*. A GTS $\mathcal{R}$ is applied to a graph G by firstly non-deterministically choosing some rule $r \in \mathcal{R}$ and then executing that rule according to the steps previously described.

As a final note, throughout this thesis we may use the term *confluent* when referring to rules and rule-sets. While we don't describe the underlying theory behind confluence here (see [61]), the reader may understand it as meaning deterministic. That is, when a confluent

rule-set is applied to an input graph until there are no more matches for any of its constituent rules then that process is guaranteed to terminate and will always produce the same result graph for a given input graph.

Double-Pushout approach for labelled graphs

With the basic notion of graph transformation by the DPO approach, the idea of labels can be introduced. A labelled graph is defined over some label set $\mathcal{L}$, the set of possible labels which can be associated with nodes and edges. Each node or edge in a labelled graph is associated with a label from $\mathcal{L}$, causing the graph to describe structured data. A labelled graph is defined:

Definition 6 (Labelled Graph)**.** *A labelled graph $G = (V, E, s, t, l_V, l_E)$ over some label set $\mathcal{L}$ consists of a finite set of nodes V, a finite set of edges E, source function $s : E \to V$, target function $t : E \to V$, node label function $l_V : V \to \mathcal{L}$ associating each node with a label and edge label function $l_E : E \to \mathcal{L}$ associating each edge with a label.*

An unlabelled graph, as given in Definition 1, can be considered as a labelled graph where all items have the same distinguished label.

With labelled graphs, there are implications for graph morphisms; they must be label-preserving. If l_V^L is L's node label function and l_V^G is G's label function, then for all nodes $n \in L$'s node set V_L, $l_V^L(n) = l_V^G(g(n))$: the same holds for edges.

In this context, rules of the form $r = L \leftarrow K \rightarrow R$ now consist of labelled graphs. From our current definitions, however, there is a possibility of relabelling preserved items. Additionally, simply treating relabelling as deletion of one node and creation of another is problematic as certain transformations would be forbidden by the dangling condition. To overcome this, we allow K to be partially labelled, that is, $K's$ label functions l_V and l_E are partial functions and some nodes and edges are unlabelled. In practical terms, when a node or edge in L is relabelled, it firstly has its label removed and then a new label is added.

Consider Figure 2.6 as an example of a rule with relabelling. Here, a rule describes a node labelled 1 being relabelled to 2 In the interface K and intermediate graph D, the node is unlabelled, but at the end of the rule execution, all nodes are labelled.

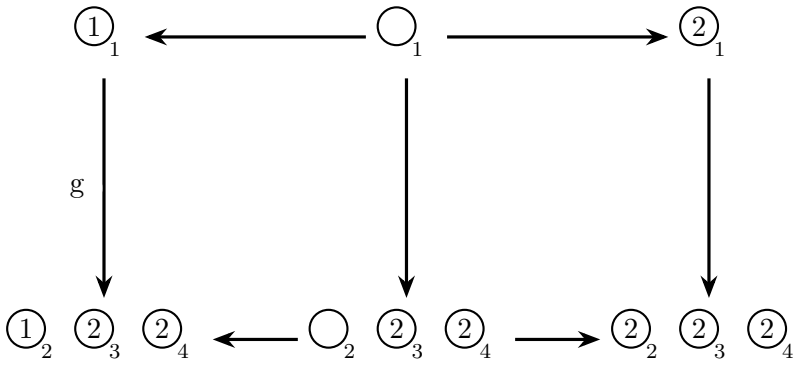


Figure 2.6: A simple DPO rule application $G \Rightarrow_{r,g} H$ with relabelling. Morphism g matches node 1 of r to node 2 of G, so that node's label is removed and updated to 2.

2.2.3 GP 2

GP 2 is a rule-based graph programming language. The original language definition, including an operational semantics, is given in [182]; an updated version can be found in [15]. There are currently two implementations of GP 2, a compiler generating C code [17] and an interpreter for exploring the language's non-determinism [16]. An introduction, rather than a full language definition, is offered here. For a full definition and more detail, refer to [15].

GP 2 programs transform input graphs into output graphs, where graphs are labelled and directed and may contain parallel edges and loops. The key construct for this is that of *conditional rule schemata*. These function as conventional DPO rules, except that we allow them to contain labels with expressions. This means that a rule schema may contain variables and produce new labels by performing computation (e.g. arithmetic) on those variables. GP 2 host graphs (see Definition 5), however, may only be labelled with lists of constant values such as strings and integers. The application of conditional rule schemata is controlled through a language that allows looping and branching execution. By allowing a program to chain together multiple rules and rule-sets, complex graph transformations become expressible which would be difficult or impossible to express with individual GTSs.

A few terms are commonly used when discussing GP 2:

- *Host graph*: The input graph of a GP 2 program. When the program is running, the intermediate results that it has produced so far are also referred to as host graphs.
- *Result graph*: The final host graph produced by a GP 2 program.

- *Match*: Where a match meant a morphism for conventional DPO rules, here it refers to a pair of items (g, α) where g is a pre-morphism and α is an assignment (see below).
- *Rule*: When it is clear that we are discussing GP 2, we use rule to refer to GP 2's conditional rule schemata.

Conditional rule schemata

Definition 7 (GP 2 Conditional Rule Schema)**.** *A conditional rule schema* $r = (L \leftarrow K \rightarrow R, c)$ *is a rule* $L \leftarrow K \rightarrow R$ *consisting of graphs* L*,* K *and* R *labelled with GP 2 expressions and a GP 2 application condition* c*.*

GP 2 rules may be labelled with any valid GP 2 expression. A grammar for GP 2 expressions is given in Figure 2.7. All expressions are *lists*, with an optional *mark*. Lists consist of *atoms*, *integers* and *strings*, and these can be expanded to integer and string expressions with operators such as basic arithmetic. There are constraints on the expressions used in the left-hand-side of a rule; all expressions must be *simple* (see [15]) so that variable assignments are unique.

Formally, there are 5 types in GP 2 expression: `int`, `char`, `string`, `atom` and `list`, and these have corresponding variables in Figure 2.7; `IVariable`, `CVariable`, `SVariable`, `AVariable` and `LVariable` respectively. In terms of type hierarchy, all types are lists, and int and string are both 'atomic' lists. Characters are seen as a subtype of string. This means that integers and strings are treated as lists of length 1 and that characters are treated of strings of length 1. This clarification is important for the matching of labels.

Conditions exist to allow the application of rule schema to be controlled and to forbid certain matches. A grammar is given for GP 2 conditions in Figure 2.8. These conditions may compare integers and variables or query notions such as variable type and the edge degree of a node. A match is only considered valid for a GP 2 rule if its assigned variables satisfy its condition.

Unlike conventional rules, matching for rule schema $r = (L \leftarrow K \rightarrow R, c)$ to some graph G has two phases. First, a *pre-morphism* is found that acts as a graph morphism but does not check for label preservation (as the domain, L, of a pre-morphism $g : L \rightarrow G$ contains unevaluated expressions). Once a pre-morphism has been found, an assignment is constructed for variables in L using the labels of mapped nodes and edges in g. This assignment α provides

⟨*Label*⟩ ::= ⟨*List*⟩ [⟨*Mark*⟩]

⟨*List*⟩ ::= `empty` | ⟨*Atom*⟩ | `LVariable` | ⟨*List*⟩ `:` ⟨*List*⟩

⟨*Mark*⟩ ::= `red` | `green` | `blue` | `grey` | `dashed` | `any`

⟨*Atom*⟩ ::= ⟨*Integer*⟩ | ⟨*String*⟩ | `Avariable`

⟨*Integer*⟩ ::= Digit {Digit} | `Ivariable` | `-` ⟨*Integer*⟩
| ⟨*Integer*⟩ ⟨*ArithOp*⟩ ⟨*Integer*⟩ | (`indeg` | `outdeg`) `(` Node `)`
| `length(` (AVariable | SVariable | LVariable) `)`

⟨*ArithOp*⟩ ::= `+` | `-` | `*` | `/`

⟨*String*⟩ ::= `"` {Character} `"` | `CVariable` | `SVariable` | ⟨*String*⟩ `.` ⟨*String*⟩

⟨*Char*⟩ ::= `"` Character `"` | `CVariable`

Figure 2.7: Abstract syntax of GP 2's label expressions [15]. `LVariable`, `AVariable`, `IVariable`, `SVariable` and `CVariable` represent the sets of variables for each type provided with the rule schema.

⟨*Condition*⟩ ::= ⟨*Type*⟩ `(` ⟨*List*⟩ `)` | ⟨*List*⟩ (`=` | `!=`) ⟨*List*⟩ | ⟨*Integer*⟩ ⟨*RelOp*⟩ ⟨*Integer*⟩
| `edge(` Node `,` Node [`,` ⟨*Label*⟩] `)`
| `not` ⟨*Condition*⟩ | ⟨*Condition*⟩ (`and` | `or`) ⟨*Condition*⟩

⟨*Type*⟩ ::= `int` | `char` | `string` | `atom`

⟨*RelOp*⟩ ::= `>` | `>=` | `<` | `<=`

Figure 2.8: Abstract syntax of GP 2's rule conditions [15]. The Label, Integer and List nonterminals refers back to Figure 2.7.

each variable in L a value, meaning that L^α now contains concrete values and a check can be made to ensure that g is label preserving for the rule induced by this assignment, $r^{g,\alpha}$. Once α is known, the condition can then also be evaluated.

The complete rule application process is:

1. Find a pre-morphism $g : L \to G$ that satisfies the dangling condition.
2. Check if there is an assignment α of variables in L such that $g : L^\alpha \to G$ is a valid morphism.
3. Check if the condition holds under α.

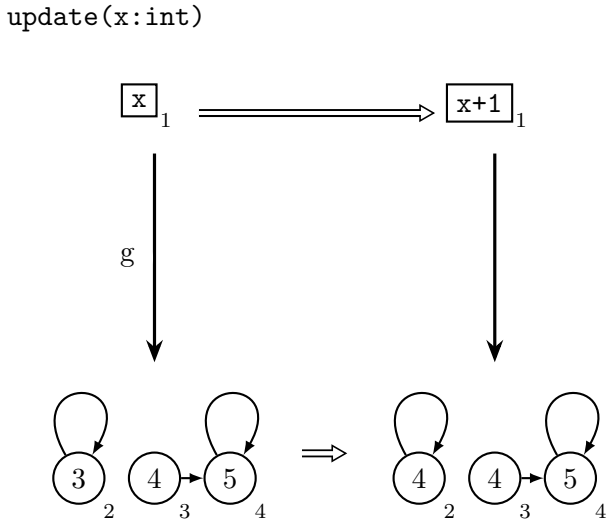


Figure 2.9: A simple GP 2 rule application of rule `update` to Host Graph G, $G \Rightarrow_{\texttt{update}^{g,\alpha},g} H$. Pre-morphism g matches node 1 of `update`'s left graph L to node 2 of G such that $\alpha(\texttt{x}) = 3$.

4. Create concrete rule instance $r^{g,\alpha}$ by evaluating expressions in R with respect to α. Execute $r^{g,\alpha}$ with pre-morphism g using the DPO approach to produce result graph H: $G \Rightarrow_{r^{g,\alpha},g} H$.

An example rule and its application is given in Figure 2.9. In the rule `update` across the top, a node labelled with an integer `x` is found, and its value is incremented by 1. Across the bottom, the host graph G has node 2 matched for r's node 1 and node 2's value is updated from 3 to 4. The K interface and D intermediate graph are excluded for simplicity.

The reader should note the use of marks. Marks group nodes and edges into different "colours", and a node with a certain mark in a rule's left hand graph L can only be matched to a node with that same mark in the host graph G. This effectively allows the programmer to split their graph into subgraphs by colour, and exclusively compute on those subgraphs. Additionally, programmers may use the `any` mark which permits a node in L to be matched to any marked node, but not unmarked nodes.

Syntax and semantics

GP 2 uses syntax to control the application of conditional rule schemata. A program executes according to its `Main` procedure. This procedure may then call other procedures. Overall, a

⟨*Prog*⟩ ::= ⟨*Decl*⟩ {⟨*Decl*⟩}

⟨*Decl*⟩ ::= ⟨*MainDecl*⟩ | ⟨*ProcDecl*⟩ | ⟨*RuleDecl*⟩

⟨*MainDecl*⟩ ::= `Main =` ⟨*ComSeq*⟩

⟨*ProcDecl*⟩ ::= `ProcId =` [⟨*LocalDecl*⟩] ⟨*ComSeq*⟩

⟨*LocalDecl*⟩ ::= (⟨*RuleDecl*⟩ | ⟨*ProcDecl*⟩) { ⟨*LocalDecl*⟩ }

⟨*ComSeq*⟩ ::= ⟨*Com*⟩ {`;` ⟨*Com*⟩}

⟨*Com*⟩ ::= ⟨*RuleSetCall*⟩ | ⟨*ProcCall*⟩
- | `if` ⟨*ComSeq*⟩ `then` ⟨*Comseq*⟩ [`else` ⟨*ComSeq*⟩]
- | `try` ⟨*ComSeq*⟩ [`then` ⟨*Comseq*⟩] [`else` ⟨*ComSeq*⟩]
- | ⟨*ComSeq*⟩ '!'
- | ⟨*ComSeq*⟩ or ⟨*ComSeq*⟩
- | `(` ⟨*ComSeq*⟩ `)`
- | `break` | `skip` | `fail`

⟨*RuleSetCall*⟩ ::= `RuleId` | `{` [`RuleId` { `,` `RuleId`}] `}`

⟨*ProcCall*⟩ ::= `ProcId`

Figure 2.10: Abstract syntax of GP 2 programs [15].

GP 2 program takes a graph G as input and either produces some result graph H, diverges, or produces a `fail` result. The `fail` result is a legitimate result that may occur when there are no matches for a given rule-set; this is distinct from run time errors, such as division by 0, that may occur.

The construct for calling rules is the rule-set; a rule-set $\{r_1, r_2, ...r_n\}$ when called may apply any rule $r_i, i \in 1, 2, \ldots, n$ and the choice of rule is made non-deterministically. If no rule in a rule-set is executable, by having no matches, then the rule-set call fails. Single rules can also be called without brackets, although this is simply shorthand for a rule-set containing one rule: $\{r_1\}$. A rule-set may also be applied as long as possible (until there are no more matches for any of its component rules) using the ! operator, but calls made in this way will not produce a `fail` result once terminated.

Commands are executed sequentially, and a program may branch to different sequences of commands using the `if` and `try` statements. These statements attempt to execute a command sequence, and should it succeed, follow one path of execution, and should it fail

follow another path of execution. The distinction between the two statements, then, is that `if` reverts any changes made by the conditional sequence of statements, whereas `try` keeps them when the condition succeeds; a programmer can view this as a normal `if` statement (checking a condition) in comparison to attempting a change to the host graph and keeping that change should it succeed with `try`.

Procedures can be declared and re-used, although these are effectively macros that must be non-recursive. The ! may also be used for a Procedure, applying a Procedure as long as possible, and terminating the execution of that command when that Procedure produces a `fail` result.

Full syntax is given as a grammar in Figure 2.10, for a full formal semantics of this syntax, rather than the informality offered here, refer to [15].

Example program: graph copying

In Figure 2.11 we present a GP 2 algorithm, `CG`, for copying a graph. The input graph is assumed to be unmarked and contain no loops, and the output graph contains two subgraphs: the original graph, now marked `blue`, and a copy of that graph, marked `red`.

`CG` consists of 3 conditional rule schemata: `copy_node`, `copy_edge` and `disconnect`. Each of these is applied as long as possible, after each other in that order. The idea of the algorithm is to apply `copy_node` as long as possible with the ! operator to duplicate all nodes in the original graph, marking the original nodes `blue` and the newly created nodes `red`, and maintaining an auxiliary edge between the original and its copy to identify the origin of each copy. The label `x` is a `list`, which as discussed earlier is the most general type in GP 2's type hierarchy, so this rule will copy nodes with *any* unmarked label. When a node is marked `blue`, once it has been copied, it cannot be a match for the unmarked node in the left hand graph of `copy_node` so cannot be copied again, causing `copy_node!` to be a terminating statement.

After all nodes have been copied, all edges are copied by applying `copy_edge` as long as possible. Each unmarked edge between `blue` marked (original) nodes is copied by creating an edge, marked `red`, with the same label between those `blue` nodes' copies (which are identified by the auxiliary edges between `blue` nodes and `red` nodes created by `copy_node`). Once an edge is copied, it is also marked `blue` to prevent it being copied again by matching the unmarked edge (labelled `z`) in `copy_edge`'s left hand graph, causing `copy_edge!` to also be a terminating statement. As the edge to be copied is between nodes marked `blue`, auxiliary

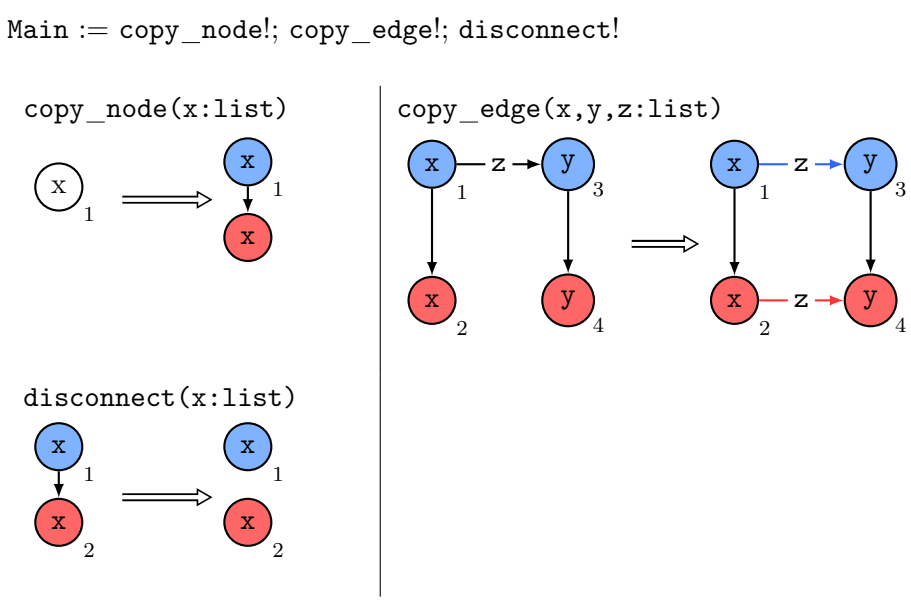


Figure 2.11: `CG`: A GP 2 program for copying an unmarked graph. The `Main` statement consists of calling `copy_node`, `copy_edge` and `disconnect` rules each as long as possible, in that order, to copy a graph, marking the original blue and the duplicate graph red. An equivalent program can also be given as `Main := {copy_node, copy_edge}!; disconnect!`

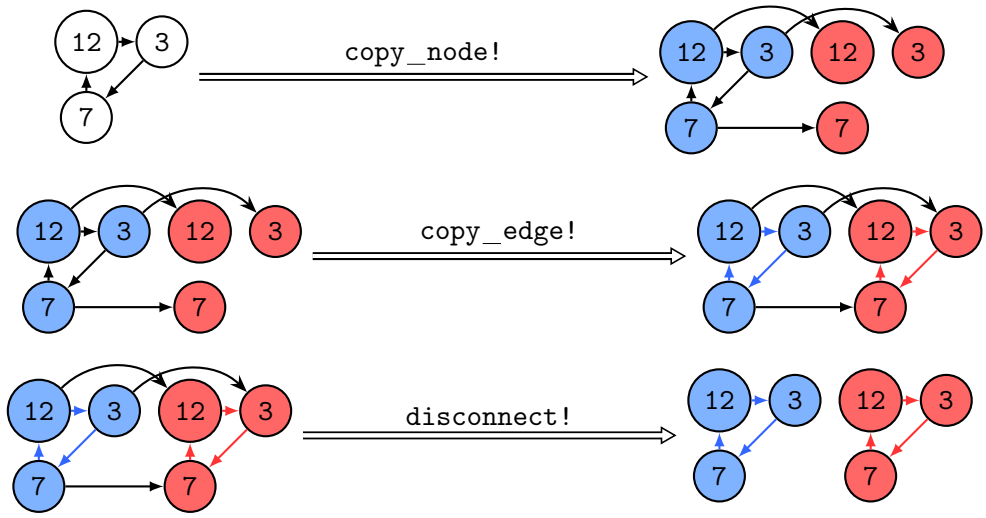


Figure 2.12: `CG` applied to a simple cyclic graph. The execution of the program is visualised according to the execution of each of its 3 as long as possible statements, `copy_node!`, `copy_edge!` and `disconnect!`.

edges identifying node copies (which are between a `blue` marked node and a `red` marked node) and copied edges (which are between `red` marked nodes) cannot be copied by this rule.

Once all nodes and edges are copied, the final step is to disconnect the two graphs so that there are no auxiliary edges between `blue` marked nodes and `red` marked nodes. The rule `disconnect` is applied as long as possible to remove the auxiliary edges. As the edge is between a `blue` marked node and a `red` marked node, the rule cannot delete any of the original or copied edges as these are exclusively between pairs of `blue` nodes or pairs of `red` nodes. Trivially, each auxiliary edge can only be deleted once, so `disconnect!` is a terminating statement.

Figure 2.12 shows the application of `CG` to a simple cyclic graph with 3 nodes labelled with integers. The produced graph contains the original nodes and edges of the input graph, now marked `blue`, and a duplicate of that graph, marked `red`.

2.2.4 Probabilistic Approaches to Graph Transformation

This thesis focuses on evolutionary computation, an inherently probabilistic approach to problem solving. However, although it has been repeatedly identified that graph transformation creates non-deterministic computation capable of taking multiple routes, that non-determinism is not typically associated with concrete probabilistic decisions. GP 2's definition leaves open how to resolve the non-determinism of choosing rules from rule-sets and matches. This is seen in the existing C-generating GP 2 compiler [15], where decisions are made on a first-come first-served basis and there is no probabilistic guarantee of any given outcome. Here, we briefly review approaches to introducing probabilities to the graph transformation domain.

There are two main approaches to the probabilistic operation of GTSs; stochastic and probabilistic. The former describes a Stochastic Graph Transformation System (SGTS), where each rule in a GTS's rule-set is associated with a real positive value, known as a rule's application rate [95, 97]. The probability of each match for a given rule occurring in continuous time is typically described according to an exponential distribution parameterised by the rule's application rate. An implication of using SGTSs is that the probability of a rule from the rule-set being applied in any given step is dependent on the number of matches for that rule. SGTSs have been generalised to a model where each match for each rule, referred to as an *event*, is associated with some continuous probability distribution, inducing generalised semi-Markov schemes describing the operation of the entire system [98, 123, 234].

Model	Deciding Rule & Match Choice	Time Model	Probabilistic Rule Execution
Classical Graph Transformation	non-deterministic	—	No
Probabilistic Graph Transformation [132]	non-deterministic	D	Yes
Stochastic Graph Transformation [97]	Probabilistic	C	No

Table 2.1: Different approaches to probabilistic decision making in graph transformation. D indicates discrete time, C indicates continuous time.

A Probabilistic Graph Transformation System (PGTS) walks a middle ground between classical graph transformation's freedom of choice and more probabilistic notions. PGTSs are GTSs where the choice of rule and match are open non-deterministic decisions but rules have different possible executions (based on a common left hand side) which occur with different probabilities [132]. This mixture of non-determinism and probabilities over discrete space is shown to induce Markov decision processes. The Markovian processes induced by the stochastic and probabilistic notions are distinct; PGTSs cannot be encoded in SGTSs and the converse is also true [132]. Table 2.1 explains the distinction between conventional graph transformation, stochastic graph transformation and probabilistic graph transformation.

Other approaches to probabilistic graph rewriting include [21], where a rule-algebra framework is proposed for the study of stochastic rewrite systems, and [49], where stochastic rewrite systems are used in the simulation and study of molecular biological systems. However, these works focus on the modelling, simulation and analysis of stochastic systems, whereas we are interested in generic probabilistic approaches to graph transformation.

2.3 Evolutionary Computation

This section gives a broad review of the field of evolutionary computation, describing at a high-level a number of 'standard' families of Evolutionary Algorithms (EAs). For this

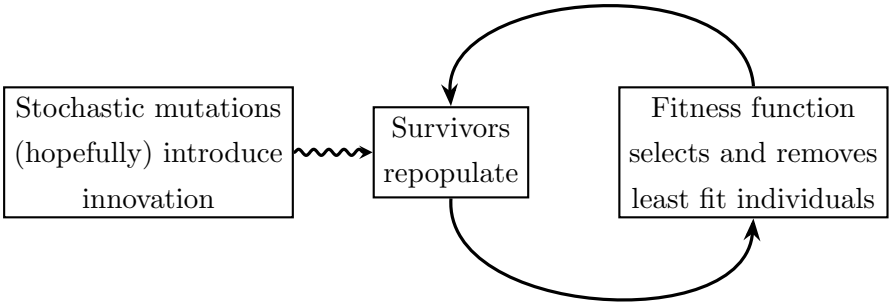


Figure 2.13: A simple model of Evolutionary Algorithms

reason, this section generally refers to older source material, rather than the state-of-the-art techniques built upon these. By introducing concepts in this manner, we hope that the reader will acquire a general notion of what EAs are and how they work, independent of specific representations and concrete algorithms.

Evolutionary computation is a family of algorithms inspired by Darwinian evolution. Typically these hold a population of potential solutions in memory, evaluate their ability to solve a given problem, and then generate a new population as recombinations and stochastic variations of those potential solutions in the previous population that best solved the problem. This is analogous to differential evolution in nature; the beings most suited to their environment are most likely to survive and reproduce and thus, over time, a species as a whole becomes more adapted to its environment. The basic model of this analogy is outlined in Figure 2.13. This might be characterised in pseudo-code as given in Algorithm 1.

Algorithm 1 Generic EA

```
1: procedure EA
2:     population ← randomised set of individuals
3:     while True do
4:         survivors ← selected from population according to quality
5:         population ← re-populated from survivors
6:     end while
7: end procedure
```

This family of "EAs" can be characterised as meta-heuristics: strategies for guiding a search process (heuristic) through a search space to produce (near) optimal solutions to a

given problem [25]. This is a splintered family of closely related algorithms separated by application domain. For example; Genetic Programming (GP) [130] and high-level neuroevolution techniques such as Neuroevolution of Augmenting Topologies (NEAT) [222] have common elements of population, mutation, cross-over and fitness-based selection and yet there is currently no discernible way to represent both in a common concrete framework. The root of this issue is complex, but a key point of divergence is the use of different representations; both for potential solutions and mutation operators. This divergence manifests itself in the application of EAs to some new domain; a state-of-the-art domain-specific algorithm may require substantial work to be applicable elsewhere.

That is not to claim that the field is entirely divided, as a number of attempts to unify known algorithms exist. For example, [229] presents a model, Adaptive Memory Programming, which attempts to unify meta-heuristics (Genetic Algorithms (GAs), as well as others such as Ant Colony optimisation and Tabu Search) using memory that is updated with provisional solutions. Taking a different view, [25] argues that meta-heuristics are searches in a 3D plane (called the I&D Frame) of *intensification* (exploitation), *diversification* (exploration) and *randomness*. However, these attempts typically rely on abstract concepts of individuals, rather than a common representation. For a search algorithm to be considered general purpose it must free its user from the demands of providing a representation for solutions and mutations, as specifying such parameters effectively means specifying a domain-specific algorithm. See [227] for more thoughts in this direction. This motivates the search for a common representation of these parameters in evolutionary computation, but it should be noted that this is a separate, if related, problem to unification, which attempts to put EAs (and other meta-heuristics) within a common framework.

Historically, this domain might have been split up into two approaches; GAs and Evolution Strategies (ES) which are discussed in Sections 2.3.1 and 2.3.2, respectively. These techniques use fixed encodings to represent individuals and then rely on some decoder to translate genomes into potential solutions which can be evaluated. GP, discussed in Section 2.3.3, emerged later and typically uses a direct tree representation of a computer program. A discussion is also given to the domain-specific field of neuroevolution, where EAs are used to evolve ANNs, in Section 2.3.4.

2.3.1 Genetic Algorithms

GAs are EAs which manipulate a population of strings of variables, often bit-strings, using mutation operators and crossover. The field can draw its history from many lines of simulated evolution in the 1950s and 1960s, although Holland's 1975 book on the subject [101] is considered a landmark that offered formality and commonly agreed GA design. The GA proposed by Holland is commonly referred to as the Simple Genetic Algorithm (SGA) [50, 150, 252]; a classic survey by Srinivas [216] describes this SGA using pseudo-code given in Algorithm 2.

Algorithm 2 Simple GA [216]

```
1: procedure SGA
2:     initialise population;
3:     evaluate population;
4:     while termination criterion not reached do
5:         select solutions for next population;
6:         perform crossover and mutation;
7:         evaluate population;
8:     end while
9: end procedure
```

Srinivas also notes several component parts of the SGA that complete the notion given in Algorithm 2:

- A population of strings of variables, corresponding to the *population* variable.
- Control parameters which describe behaviour such as the rate of mutations, population size, the maximum number of generations before termination or the minimum fitness required for termination.
- A fitness function, which evaluates each solution within the population according to its ability to solve a given problem. This is used to evaluate the entire population in lines 3 and 7.
- Genetic operators. These are operators which describe crossover and mutations on members of the population.
- A selection mechanism to select solutions for the next population in line 5. This is separate from though driven by the fitness function; it is not simply a matter of choosing the very best performing solutions as that may eliminate solutions which show potential

novelty.

- A mechanism to encode/decode solutions as strings of variables. This allows the SGA to be applied to domain-specific problems, by translating between string *genotypes* and concrete solution *phenotypes* which can then be evaluated by the fitness function.

In contrast, Harvey describes a minimal version of the GA, the Microbial GA [92], which is more specific than the SGA. In the Microbial GA, a population of bit-strings is updated by picking pairs of individuals from the population and replacing the worst performing with a child of the two. This simplified approach may be useful when introducing graph transformation to EAs.

A number of optimisations exist for GAs beyond the SGA, for example, parallelisation [32]. Whereas a master-slave model that allocates work to sub-processes offers possible efficiency savings [33], the island model of parallel GAs proposes a new paradigm. In this model, there are several populations which are evolved separately, although some individuals are passed between populations to allow useful genes to spread [256]. The intuition is that by maintaining multiple populations, more diversity is achieved as the algorithm searches multiple spaces simultaneously, with more diversity assumed to improve the chance of finding novelty and better optima. Further optimisations include elitism [51] and adaptive GAs [146], although these are not discussed in further detail here.

While suited to a diverse range of problems, for example, general assignment [139] and flowshop sequencing [194], GAs are not without flaws. If an EA has low locality, a measure which describes how similar new individuals are to their parent individuals, then it may struggle [196]. This is apparent in GAs when using a poorly chosen encoding/decoding mechanism. More importantly, in the context of graphs, we see no benefit to representing GAs as graph programs, as this would mean graph transformations over strings of variables - structures with simple linear topologies. For this reason, GAs are not discussed further.

2.3.2 Evolution Strategies

ES is a numerical approach to EAs where individuals are treated as vectors of real values and mutations alter these vectors according to, typically Gaussian, distributions [13]. ES was first implemented as a simple algorithm where each generation has one parent individual and one child individual; this is known as $(1 + 1)$-ES. This algorithm, outlined in Algorithm 3, serves as an intuitive introduction to ES.

Algorithm 3 $(1+1)$-ES, a simple evolution strategy [13]. *parent* corresponds to the parent individual's real vector representation and θ is a vector of standard deviations for mutations. $\mathcal{N}(j)$ takes a real vector j as input and produces a vector of real values k where $k(i)$ is drawn from a normal distribution with mean 0 and standard deviation $j(i)$.

```
1: procedure (1 + 1)-ES(θ)
2:     initialise parent
3:     child = parent + N(θ);
4:     evaluate parent, child;
5:     while termination criterion not reached do
6:         parent = highest fitness solution from parent, child;
7:         child = parent + N(θ);
8:         evaluate child;
9:     end while
10: end procedure
```

Here a number of components, such as fitness functions and control parameters are required to provide a concrete $(1+1)$-ES implementation, as was the case with the SGA discussed in Section 2.3.1.

Iterating upon $(1 + 1)$-ES, there are several ES variants to address different concerns. Multimembered ES introduces multiple parents for each generation, allowing the notion of crossover to be introduced [152]. By dividing a population into several sub-populations which are evolved in parallel [198], it is possible to parallelise ES in a similar manner to the island model for GAs.

As with GAs, we see no benefit to representing ES with graph programs, as this would mean graph transformations over vectors - again, these are structures with simple topologies. For this reason, ES is not discussed further.

2.3.3 Genetic Programming

GP was initially introduced as an application of EAs to computer programs. In its earliest iteration, individuals were represented as either strings of integers or trees that were then parsed to produce simple programs [48]. However, this tree representation rapidly became the convention [130] and to some readers it may even be synonymous with general GP. The tree-based approach is discussed below, followed by discussion of another approach,

Linear Genetic Programming (LGP), which attempts to apply EAs to imperative programs. Koza [130] outlines the general notion of GP free from representations as given in Figure 2.14, which does not significantly differ from the general approach of EAs given in Algorithm 1.

Tree-based Genetic Programming (TGP) In the tree-based representation, a tree is used to represent the structure and operation of a program. Nodes represent functions, variables or constants, and child nodes represent inputs to their parent. As an example, Figure 2.15 shows a tree representation of a program for the formula $(y + (z - 1)) \times (3 \times x)$.

Crossover is usually achieved by exchanging subtrees between two parent solutions [130]. Nodes within each parent are selected as cross-over points; once a cross-over point has been selected in each parent tree, the entire subtree of the crossover nodes and their children can be swapped into the other tree. A number of mutation operators are viable, for example, headless chicken crossover involves crossover between a solution and a randomly generated tree [3]. Point mutations, where a node is relabelled with a new function, constant or variable, incorporate more fine-grained changes to a given solution. Regardless of the approach taken, there is evidence that crossover alone is not sufficient for exploration, as Koza initially postulated [130], and that random mutations offer additional avenues of exploration [3, 143].

As with GAs, many optimisations are possible for TGP; the island model described in Section 2.3.1 is applicable here [68]. In Strongly Typed GP, each variable and constant is associated with a type, allowing crossover to target subtrees of the same data type and prevent the production of invalid individuals [160]. Meta GP describes a context in which a GP system can be evolved through GP [59].

As a field, TGP has been plagued by the issue of bloat. As trees have unbounded size individuals often grow in size without meaningful gains upon reaching a local optimum [142]. This growth can then slow the GP system down and wastefully consume additional memory. Several techniques exist to overcome bloat; larger-than-average individuals can have their fitness artificially reduced [190], destructive mutations can be favoured for larger individuals [122] or the total size of the population can be limited [208].

As this tree-based representation is inherently a graph, considering TGP as graph-based GP appears straight-forward; individual representations can remain as they are, while mutations and crossover operators can be represented as relatively simple graph programs.

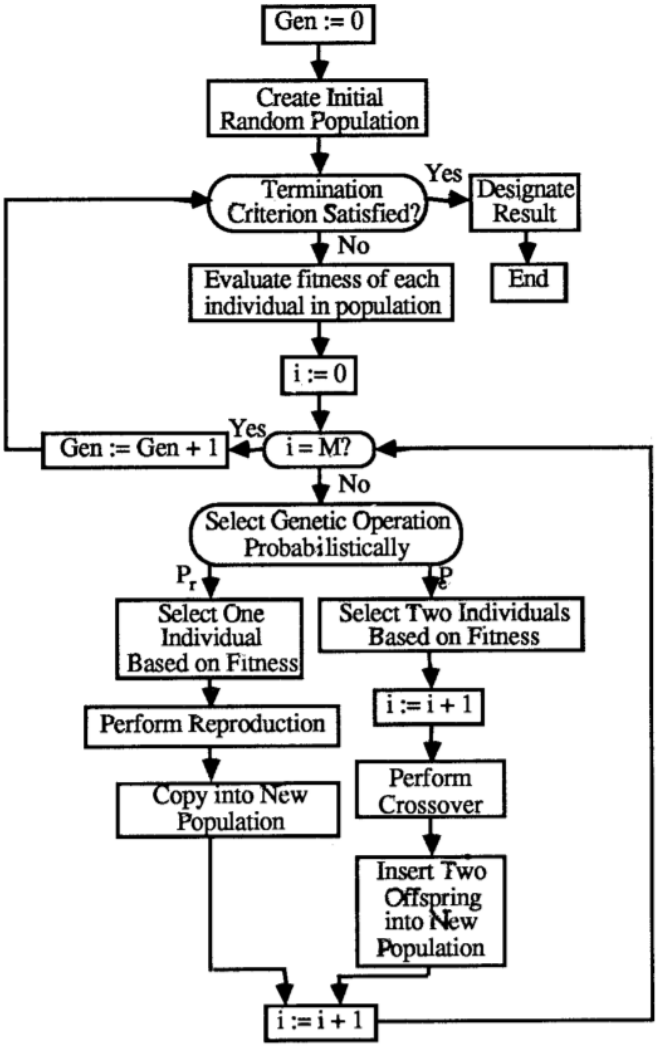


Figure 2.14: A model of GP, figure taken from [130]. M is the number of reproductions to perform in a given generation. P_r is the probability of a reproduction being a conventional mutation, whereas P_c is the probability of a reproduction being a crossover operation.

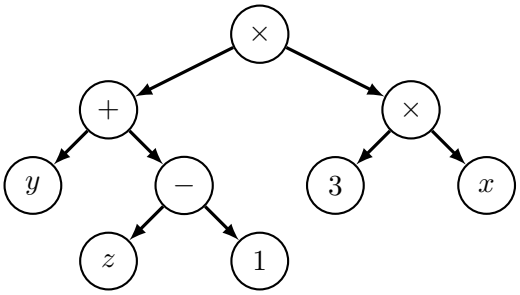


Figure 2.15: A tree representation of the formula $(y + (z - 1)) \times (3 \times x)$.

Linear Genetic Programming (LGP) LGP [18, 26] techniques evolve linear sequences of instructions; these programs typically have a shallow or flat structure which is not intuitively described by a deep tree. There are a number of reasons for evolving linear sequences of instructions, for example: by evolving machine code the cost of evaluating programs may be several orders of magnitude faster than interpreting trees [171] and it becomes possible to evolve machine code specifically targeting some physical device [135] or virtual machine [60]. In conventional LGP, each instruction in an individual may access and manipulate the contents of a globally available set of registers. Unlike TGP, LGP may produce intronic code which offers no functional purpose but remains within an individual's genome. There are cases where this intronic code may improve the overall performance of the system [103]. LGP offers an intuitive approach to evolved concurrency, with concurrent programmings represented as multiple sequences of linear instructions [36]. There are also attempts to implement meta-learning with LGP, by evolving instruction sequences which describe ES [174].

In a graph setting LGP appears to show the same lack of advantage as GAs discussed in Section 2.3.1 due to the linear representation of individuals. However, we could view evolved programs as graphs in the sense of data-flow diagrams with nodes representing instructions and edges representing the flow of information from a previously executed instruction e.g. by accessing the same register. Additionally, there are works which introduce additional structure to LGP: Linear-Tree GP [113] represents individuals as trees where each node contains a sequence of instructions, and Linear-Graph GP [114] extends this notion to graphs.

2.3.4 Neuroevolution

Neuroevolution is a somewhat unique sub-section of this contextual review; rather than focusing on a common algorithmic approach, this field is unified by its problem domain: Artificial

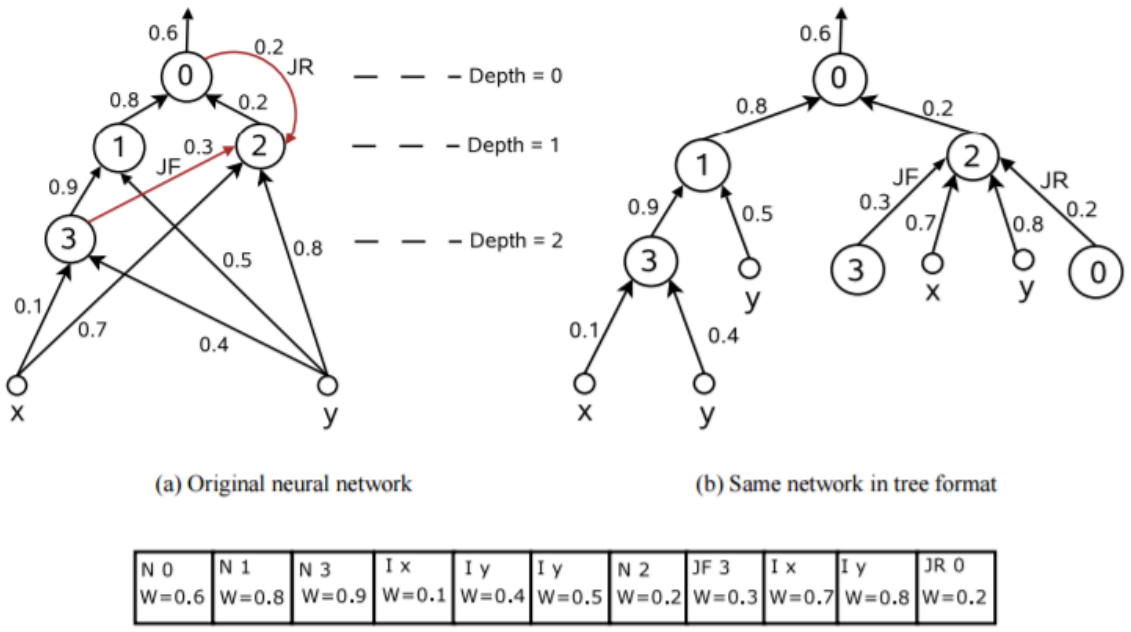
Neural Networks (ANNs). Here, the reader is assumed to be familiar with ANNs, but if this is not the case, refer to [24].

The first division one can make in neuroevolution is which aspects of a given ANN are evolved. An ANN can be seen as a topology with weights attached to edges; the topology can be evolved, the weights can be evolved, and these two aspects can potentially be evolved in conjunction.

A number of works focus on ANNs with fixed topologies. As the topology is fixed, the number of connections is fixed and as such, the number of weights which must be evolved is fixed. This lends itself to GAs; each weight can be associated with a fixed number of bytes meaning that the entire neural network's connection weights can be represented as a fixed-length bit string [22, 161, 257]. Some domain-specific approaches focus on evolving weights directly rather than through some encoding [81]. These approaches, on the surface, appear well justified: conventional training of ANNs through the back-propagation technique also keeps topologies fixed [255] and this evolutionary approach appears analogous. However, there is evidence suggesting that evolution of ANN topology contributes significantly to the learning process [222] and, in particular, topology evolution *alone* may in some circumstances out-perform weight evolution alone [239].

In contrast, ICONE [195] evolves neural networks by treating neurons and connections as separate entities that describe an overall architecture. By grouping elements according to tags and groups, cross-over is achieved and a network can be broken down into modular components. In EANT and EANT2 [117, 207], neural networks are encoded as a linear sequence of genes that describe neurons and connection that construct an individual network. Figure 2.16 shows the comparison between a neural network, a tree interpretation of that network, and a linear sequence of genomes representing that network as used in EANT2; this representation is evolved using ES for weights and dedicated structural mutations. In a similar manner to EANT, NEAT [222] constructs neural networks from linear sequences of genes. As the authors view this as manipulating a graph structure, this algorithm and its variations are discussed in further detail in Section 2.4.3, although graphical interpretations could be provided for others such as EANT and ICONE.

Neuroevolution has found success for complex real-time problems, particularly where recurrence is relevant. For example, some applications of NEAT focus on evolution of agents for games such as racing [34, 35] and strategy and intelligent behaviour [219]. Cartesian Genetic Programming (CGP), an approach discussed in Section 2.4.1, similarly developed recurrent



(a) Original neural network

(b) Same network in tree format

N 0 W=0.6	N 1 W=0.8	N 3 W=0.9	I x W=0.1	I y W=0.4	I y W=0.5	N 2 W=0.2	JF 3 W=0.3	I x W=0.7	I y W=0.8	JR 0 W=0.2

Figure 2.16: An example encoding of a neural network in EANT2, figure taken from [207]. (a) shows the neural network in full, and (b) shows a tree format representation of this neural network; recurrence is represented as a node appearing in its own subtree as with node 0. Across the bottom, a sequence of genomes describe the construction of (b), with N genes adding nodes, I genes adding connections from inputs, JF genes adding feed-forward connections and JR genes adding recurrent connections. W in each gene represents a corresponding connection weight.

game agents for a real-time pole balancing problem [126]. Because of the natural interpretation of a neural network as a graph, with neurons as nodes and connections as edges, and the apparent application to problems that conventional machine learning struggles to address, this is a highly motivating field for further study.

2.4 Graphs in Evolutionary Computation

Graphs are a commonly used structure in evolutionary computation. This section addresses work where graphs have been explicitly used as a representation in evolutionary computation and is effectively a general purpose 'related work' section of this thesis that we refer back to. Due to the universality of graphs as a data structure, this section could bloat without some restrictions, so we will make the following constraints:

1. The graph representation used in a given work should be sufficiently complex. It would be possible to include TGP [129] here as trees are a well-defined subset of graphs, but this is perhaps not the most fruitful direction of thought due to trees' structural simplicity.
2. The graph representation should be either direct or close to direct. As an example, various works attempt to learn quantum circuits (which *are* in some sense graphs) via TGP and a domain-specific language that can be *decoded* into a circuit [213]. But from the perspective of the EA, it is the tree that is assigned the fitness value, so in some sense, this falls under our first constraint. In contrast, the work in [6] uses ant colony optimisation to directly generate quantum circuits as subgraphs of a Cartesian graph, which would be more relevant with respect to our discussion of representation.

Several we discuss, such as CGP [157] and NEAT [222], effectively describe their individuals as lists that directly encode nodes and edges, but this does not fall under our second constraint as this is simply another way of notating a graph under Definition 1.

In the following sections, we describe in detail 3 significant graph-based EAs. In Section 2.4.1 we describe CGP, a generic EA for learning Directed Acyclic Graphs (DAGs) with many graph-based applications. In Section 2.4.2 we describe Parallel Distributed Genetic Programming (PDGP), an extension to TGP with many similarities to CGP. In Section 2.4.3 we describe NEAT, a neuroevolution system which explicitly learns a graph-based representation of ANNs. Finally, in Section 2.4.4 we will cover a number of other graph-based EAs in less detail, discriminating between those approaches which learn graphs, and those approaches which learn graph-like solutions to domain-specific problems *without* the use of an obfuscating encoding.

2.4.1 Cartesian Genetic Programming

CGP is a graph-based EA with a wide variety of applications. In this section, we describe the basic approach and a number of extensions to CGP that have been proposed. For more detail and an up-to-date survey on the status of CGP, refer to [158].

The main principle (from our graph-based perspective) of CGP is that an individual is represented by a graph with a fixed and ordered set of nodes [157]; a node may only be the source of edges that target nodes with an earlier position in the ordering. Each node represents a function that is applied to its inputs (given by its outgoing edges). In many modern works, this is a total ordering where one can imagine nodes arranged in a line with connections allowed only to the left. However, earlier works [154, 157] often used a partial ordering where one can imagine nodes arranged in a 2-dimensional grid, again with connections only allowed to the left. For our discussion, we use the former notion, as this is more prevalent in modern CGP usage e.g. [165, 226, 245]. In CGP publications such as [154, 157], this is often implemented via an encoding where solutions are linear sequences of integers that are decoded into DAGs. We give a typical genotype/phenotype mapping in CGP in Figure 2.17.

The most commonly used mutations take two forms; they can either re-label a node with a different function, or they can redirect edges while respecting the given ordering. A number of other mutation operators have been proposed in the literature which have various benefits and costs [76, 77]. The current advice is to use CGP with mutations only, rather than with crossover [155, 243], alongside the $1 + \lambda$ EA. This approach[1] sees a single individual survive in each generation which is then copied and mutated to generate λ children. However, a number of works have explored the use of crossover in CGP, including uniform crossover [154], arithmetic crossover on a vector representation [41], and subgraph crossover [112]. Empirical comparison [106] shows that crossover operators do not always aid performance and that CGP with mutation only can sometimes be the best performing approach.

Figure 2.17 highlights that a node (in this case, the AND node) may be the input of no other node in the graph, and therefore contribute nothing to the output. These inactive nodes can build substructures which can effectively undergo random walks, allowing a property referred to as neutral drift to occur, which is believed to allow CGP to escape local optima by exposing the search algorithm to new neighbourhoods [244, 251, 266]. The claim that this process always aids performance has been contested [43]. There is evidence suggesting that the performance of CGP can be further accelerated by increasing the amount of redundant

[1] Which I have heard called 'glorified hillclimbing', but works remarkably well.

Genotype:

Node 1			Node 2			Node 3			Output
0	0	1	1	0	2	0	1	0	2

Phenotype:

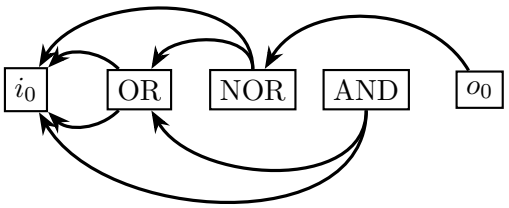


Figure 2.17: The genotype-phenotype mapping of a simple CGP individual consisting of 1 input, 3 nodes and 1 output and arity 2. Each node is represented by 3 genes; the first 2 describe the indices of the node's inputs (starting at index 0 for the individual's input i_0) and the third describing the node's function. Function indices 0, 1 and 2 correspond to AND, OR and NOR respectively. The final gene describes the index of the node used by the individual's output o_0.

material present [156].

Many extensions to CGP exist in the literature. A modular variant of CGP named Embedded CGP has been proposed where modules (subgraphs) are automatically acquired and reused throughout the evolutionary process, often accelerating the search [253, 254]. Recurrent CGP [241, 242] allows the existence of recurrent connections which can target any node in a graph, facilitating the induction of recursive solutions to problems such as generating the Fibonacci sequence or time-series forecasting. Self-modifying CGP [90] facilitates the inclusion of nodes that can create and delete other nodes, thereby allowing a graph to develop to solve a class of problems, such as computing π or e to arbitrary precision [91].

Using a graphical structure, CGP has demonstrated its capability as a cross-domain optimisation algorithm. Initially, it was proposed as a means to evolve circuits for Boolean functions [154], but this was generalised to the concepts described in [157]. A number of works have extended its application to search over *approximate* circuits by employing a multi-objective EA and exploring the tradeoff between accuracy, with respect to a target truth table, and cost, with respect to power consumption, circuit delay and size [165, 249, 250]. With the introduction of edge weights, CGP has been shown to effectively evolve ANNs [124, 126, 238]. By relaxing the requirement for a CGP individual to be feed-forward, it is possible to evolve

Recurrent Neural Networks (RNNs) capable of solving real-time problems [125, 245]. An interesting application can be found in [137], where a two-tiered version of CGP was used to evolve an abstract sequence of instructions with repetition of instructions occurring as a function of numerical parameters. Other application areas include (but are not limited to) convolutional neural network architecture design [226], multi-step forecasting [58], cryptographic circuit design [179, 180] and image processing [88, 89, 201].

To summarise, CGP is a graph-based EA that uses a direct[2] encoding of graphs as a representation. Since its inception, it has spawned a broad field of research with many extensions both in terms of genetic operators and representation. The empirical observations of neutral drift in CGP and related theory have since spilt over into more general evolutionary research and should be a consideration in any discussion of neutral drift, e.g. [72]. Perhaps the most relevant observation in relation to our work is the generality shown by CGP, with a wide range of application areas *unified by the common representation of solutions as graphs.*

2.4.2 Parallel Distributed Genetic Programming

PDGP is an extension of TGP that bears much resemblance to CGP with respect to representation and emerged independently at around the same time [187]. The main representation concept in PDGP is that, rather than single output trees, an individual is a multiple output program existing as a 'graph on a grid' [186, 188, 189]. Figure 2.18 shows an example PDGP solution.

In PDGP, nodes may connect to nodes one layer previous in the grid. To allow deeper connections, inputs may be passed through layers via 'wire' functions which compute identities of their inputs, and an example of this (a node with a vertical line through it) is given in Figure 2.18. Unlike mainstream CGP thought, most genetic operations in PDGP are done through crossover. In particular, crossover operators such as Subgraph Active-Active Node (SAAN) crossover are used to recombine parts of solutions while attempting to minimise disruption to the rest of the solution. A diagram visualising SAAN crossover is given in Figure 2.19. Here the subgraph induced by an active node in parent 1 replaces the subgraph induced by an active node in parent 2. To ensure that the solution still fits on the grid, it is wrapped around as shown. A particular difference between PDGP and CGP is the lack of atomic mutations in PDGP; instead, mutation is achieved by crossover with randomly generated solutions [186].

PDGP has been shown empirically to perform favourably in comparison to TGP [188]

[2] Or, very close to direct, depending on perspective.

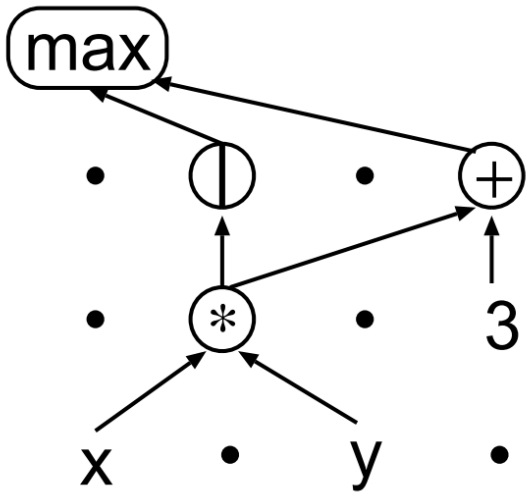


Figure 2.18: A PDGP individual representing the solution $max(x \times y, 3 + x \times y)$, placed on a 2 dimensional grid. Figure taken from [186]. The node with a vertical line represents a 'wire' function, computing an identity function on its input.

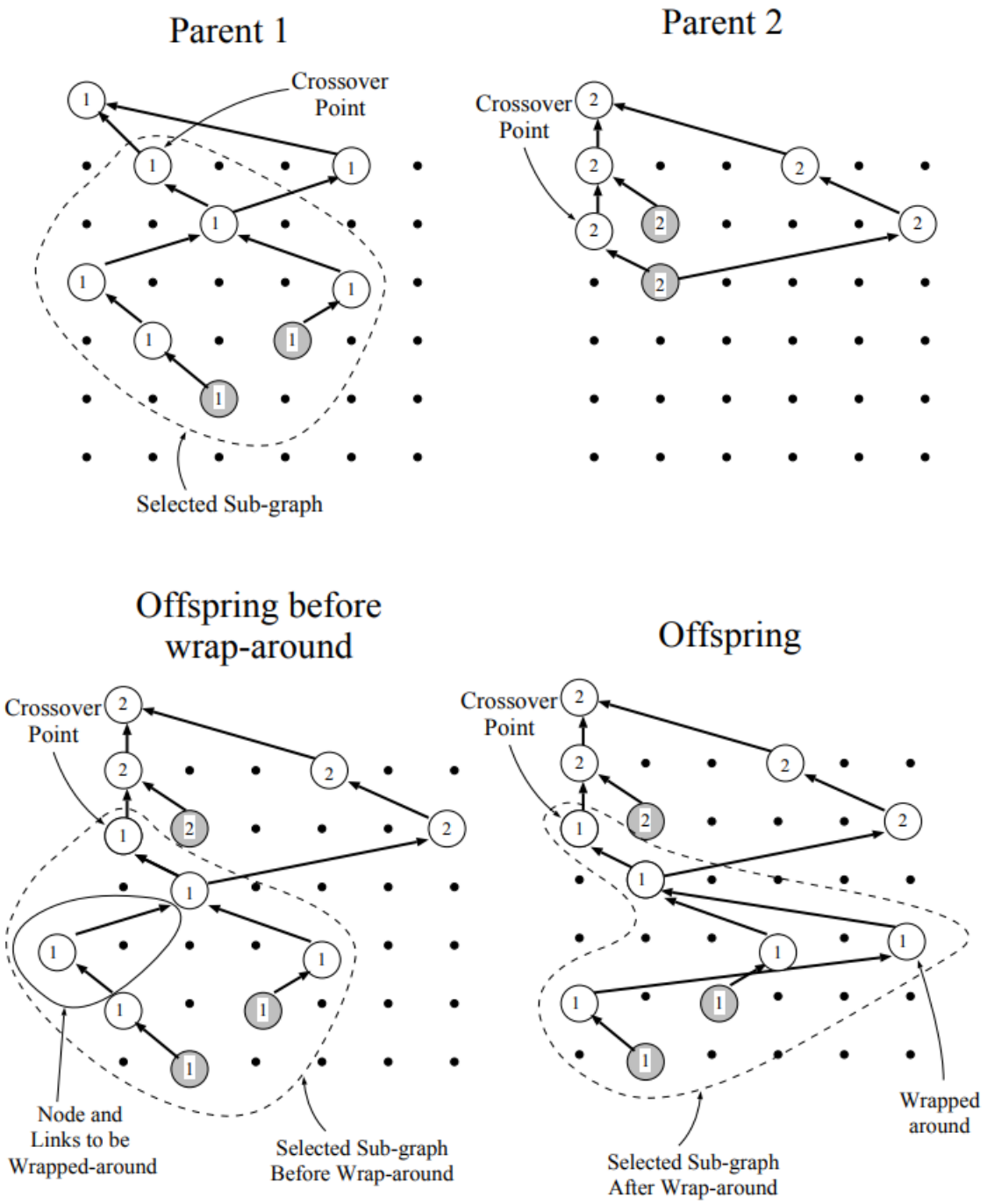


Figure 2.19: SAAN crossover in PDGP, figure taken from [186]. An active subgraph induced by a single node (the crossover point) in parent 1 replaces an active subgraph in parent 2. The inserted content is wrapped around to ensure that it still correctly fits on the 2D grid.

and has been used to evolve ANNs [192]. However, despite its apparent generality and efficiency and notable similarity to CGP (which the creators of CGP have often pointed out [155, 158]), PDGP has not seen the same degree of wide-spread research interest. While the reasons for this are debatable, it is quite clear that due to the shared representation, many of the applications of CGP would also be target applications for PDGP. PDGP is interesting, from our perspective, because it in many ways represents an open question in graph-based evolution; what are good crossover operators for graphs, and do these improve upon simple mutation operators? There is a lack of direct empirical comparison between CGP and PDGP in the literature.

2.4.3 Neuroevolution of Augmenting Topologies

NEAT is a form of neuroevolution explicitly constructing a graph representation of a neural network. The philosophy of the algorithm is to keep networks minimal and use historical changes to find points of crossover [222]. Figure 2.20 shows the relationship between a NEAT genome representation, a linear sequence of genes, and its corresponding neural network.

In NEAT, structural mutations are additive, instead of varying or deleting existing components. Structural mutations take two forms; adding a node, and adding an edge. Adding a node is done by disabling a previous connection gene, and inserting a connection from that previous connection's source to a new node, labelled with weight 1, and a connection from that new node to the previous connection's target, with the previous connection's weight. This is done in this way to minimise disruption to the network behaviour. Adding a connection is done simply by choosing a new source and target node. These structural changes are recorded as 'innovations' which are used to track common history of two individuals; if they share an innovation number then they have a common ancestor up until that innovation, and so crossover can take place by lining up these common genes and thereby avoiding using expensive topological analysis while attempting to establish coherent crossover points.

NEAT makes extensive use of 'speciation'. Rather than having the entire population of networks compete with each other, networks instead compete within 'niches' which are defined by the distance between networks. The distance between networks is, in part, computed using the historical markings we have already discussed. By doing this, NEAT protects topological innovations, giving new topologies time to adapt before discarding them.

NEAT has the noteworthy ability to solve hard, often real-time, neuroevolution problems. For example, beyond classic control problems such as pole balancing [222], NEAT has also

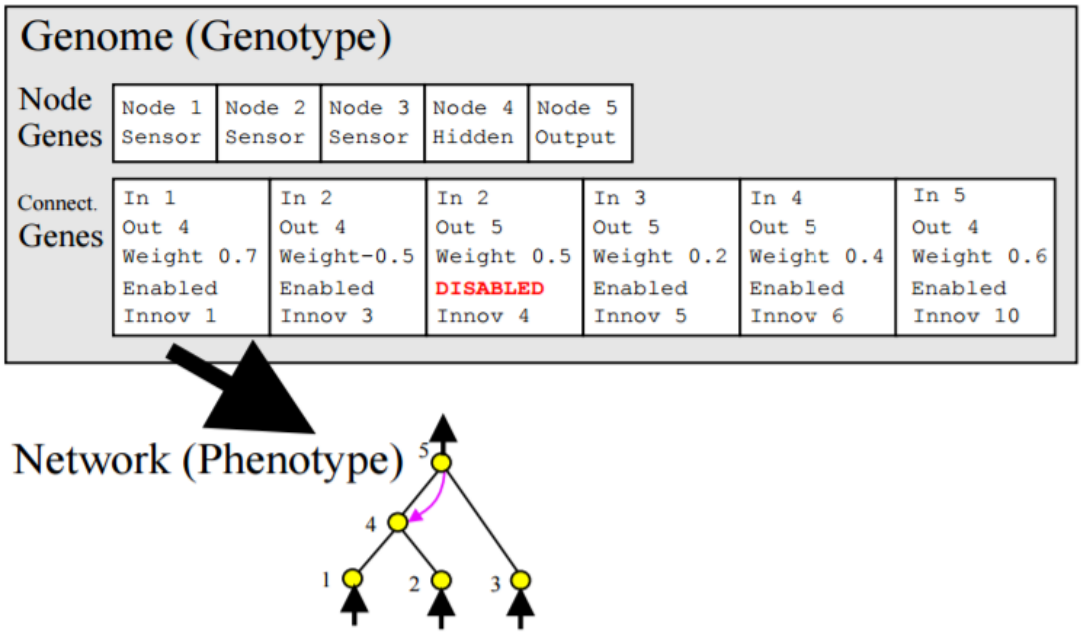


Figure 2.20: An example encoding of a neural network in NEAT, figure taken from [222]. The network is described by a set of node and connection genes which are used to construct a neural network that can then be evaluated.

been used to discover car controllers [34, 35]. A real-time version of NEAT has been used in coordination with a custom game NERO so that agents could be trained to play against a human player in real-time [219]. NEAT is also used extensively in fields such as evolutionary robotics [52, 53] and swarm robotics [57, 78].

A number of notable variants of NEAT have been proposed. HyperNEAT [73, 220] is a variant of NEAT where the networks evolved are more complex Compositional Pattern Producing Networks (CPPNs). The main distinction between these CPPNs and ANNs is the use of activation functions [218]. These CPPNs can then be used to generate the weights of a larger neural network, by effectively generating a hyper-cube of real values. By doing this, it becomes possible to use HyperNEAT as a reinforcement learning algorithm for training large Convolutional Neural Networks which can perform complex tasks such as playing video games from raw pixel information [94] or playing GO [74]. However, there is evidence to suggest that traditional TGP can replace NEAT in HyperNEAT and perform similarly [29].

Another variant, CoDeepNEAT [153], substitutes neurons with Deep Neural Network (DNN) layers. By evolving architectures with the NEAT methodology, alongside hyper-parameters, it is possible to evolve architectures capable of competing with human-designed architectures [136,153]. However, several other deep neural architecture search techniques have been set out [5,193,226], and without empirical comparisons available, it is unclear whether CoDeepNEAT is particularly effective.

NEAT is capable of evolving a wide variety of neural-inspired graphs; ANNs [222], CPPNs [220] and DNNs [153]. It is not unimaginable, then, that it would be possible to extend NEAT to evolve the range of graph-like structures studied with CGP, such as digital circuits [157] or forecasting solutions [58]. This direction of thought is encouraged by the fact that CGP has indeed been used to evolve neural networks [126] and DNNs [226]. A major distinction between CGP and NEAT is that CGP uses a fixed size representation and a highly elitist EA, whereas NEAT grows solutions while attempting to maintain diversity. If the comparison between CGP and PDGP is one of mutation vs. recombination of graphs, then the comparison between NEAT and CGP may be one of fixed size graphs vs. growing graphs, and elitism vs. diversity.

2.4.4 Other Graph-Based Evolutionary Algorithms

In this section we discuss various other graph-based EAs from the literature. This section is not exhaustive; many works could be viewed as relevant and compiling such an exhaustive list would be a research endeavour in itself. Instead, the intention is to give the reader an overview of the varied and extensive use of graphs in evolutionary computation.

Graph-based Genetic Programming. There have been a number of other extensions to GP that utilise a graph-like structure.

Multiple Interactive Outputs in a Single Tree (MIOST) [71, 138] proposes using trees with multiple output nodes and sharing to extend traditional GP to domains where problems have multiple, related outputs. Sharing is created via 'p' function nodes, which may point to other nodes in the graph. All other nodes are structured as trees as in conventional TGP. Then traditional tree-based genetic operators (with small modifications to account for p nodes) may be used such as crossover or subtree mutation. While the representation used in MIOST appears to approach that of CGP or PDGP, the use of p nodes as an ad-hoc extension of TGP and the reuse of tree-based genetic operators suggest that this approach is further from

our interests.

Linear-graph GP [114] can be viewed as a graph-based extension of LGP where individuals are represented as graphs. Each node in the graph contains a linear sequence of instructions which are executed when the node is reached. After these instructions are executed, an *if-then-else* branching instruction is evaluated which selects which node should be evaluated next. Individuals may then be recombined by exchanging linear sequences of instructions between nodes, or by identifying and exchanging entire subgraphs. Experimentally, Linear-graph GP was shown to outperform an LGP system [114].

Tangled Program Graphs (TPGs) [120] is a modular extension of GP whereby nodes in a graph represent cooperating programs which, collectively, constitute an agent capable of interacting with an environment. Each of these nodes is labelled with a sequence of instructions, in the manner of LGP, and therefore the representation of TPGs appears quite similar to Linear-Graph GP. However, as TPGs are generally used to represent agents, it is therefore often necessary for them to be stateful; it then follows TPGs may contain cycles, unlike Linear-Graph GP. TPGs has been used extensively to learn agents capable of playing video games [120, 121, 210].

Evolution of neural network topology As we have discussed in Sections 2.3.4 and 2.4.3 there are various approaches to neuroevolution which incorporate evolution of topology, such as NEAT [222], ICONE [195] and EANT2 [117]. There are further works which could be considered relevant here.

GeNeralized Acquisition of Recurrent Links (GNARL) [4] instantiates a population of recurrent networks by choosing a random number of hidden units chosen from a user-defined range and then adding a random number of connections chosen from a user-defined range[3]. GNARL primarily modifies networks via mutation, rather than crossover, and distinguishes between weight mutation, which permutes connection weights, and structural mutation. Structural mutations may add or remove hidden units and connections, which is a relatively straightforward process due to the fact that purely RNNs have very few structure constraints.

EPNet [264] is a hybrid method which combines structural mutations of topology with backpropagation training of weights. In each iteration of the evolutionary process, network weights are trained by backpropagation. Networks are then replaced with their children,

[3] In some sense, initialisation of these networks resembles sampling from a variety of $\mathcal{D}(n, p)$ directed random graph models as described in [82].

which are structurally mutated with hidden unit addition/deletion and connection addition/deletion. In some sense, EPNet utilises a form of 'Lamarckian' evolution, whereby the population develops (via backpropagation), and their developed features, e.g. connection weights, are passed to the next generation.

Evolution of automata A number of works have investigated the evolution of automata with a view of individual solutions as graphs. In fact, some of the earliest work on evolutionary computation was on the evolution of automata [69]. Some of these works could have been placed in the earlier 'Graph-Based Genetic Programming' section as the authors themselves view their contributions as extensions of GP. However, as they evolve domain-specific forms of automata, rather than arbitrary programs, they are described here instead. In the context of the work undertaken here, they are perhaps less useful as points of inspiration as they work with specific forms of automata rather than more generic graph structures, but there is clearly value in examining how they represent and modify graphs.

Graph Structured Program Evolution (GRAPE) [206] combines a graph-like structure with a custom branching functions to evolve automata capable of accessing and modifying an internal register. When a node is evaluated, the register is used in a computation, and then a decision is made over which node should be evaluated next. Special functions are added to the function set which determine branching behaviour based on the contents of the register. GRAPE has been used to induce sorting algorithms [203] and various recursive functions such as factorials [205]. It has also been extended to support Automatically Defined Nodes [204] which take on a similar role as Automatically Defined Functions (ADFs) in TGP [129].

Genetic Network Programming (GNP) [118] proposes an EA over graph-like automata consisting of 'judgement' nodes (which operate as *if-then-else* statements) and 'processing' nodes which perform some action. Mutation in GNP is quite similar to that of Recurrent CGP [241], in that with a certain probability, each edge is redirected to point anywhere in the structure. Crossover in GNP bears some resemblance to early work on uniform crossover in CGP [154], where nodes are selected from each parent independently of the rest of the structure. GNP is often used in a reinforcement learning context, to induce agents capable of reading and interacting with some environment; see for example its application to elevator controls [100] and stock-market trading [37].

Parallel Algorithm Discovery and Orchestration (PADO) [231, 232] proposes an EA over graph-like automata which are very similar to those used in GNP. In PADO, nodes have both functional behaviour and branching behaviour, both of which are governed by a stack and

indexed memory. The main application of PADO is to synthesise object recognition systems.

Various works in the literature have considered the evolution of Turing machines e.g. [144, 167, 230], although many of these fall under the second constraint we placed on relevant literature. A particular work of interest is found in [177], where the authors represent Turing machines as graphs encoded in a linear genome and develop a crossover operator based on the structure of the underlying graph. Here, crossover is achieved by picking a node in each graph as a crossover point and then exchanging subgraphs reachable within a certain number of connections from the crossover points.

Evolution of Bayesian networks Various approaches have been set out that target the optimisation of Bayesian network structure. For a more detailed review and discussion, see [133]. A number of these approaches focus on modifying the connectivity matrix of a Bayesian network via a GA [66, 134, 148], which, while a valid representation of a graph, is perhaps less interesting in our context. However, several works directly modify the Bayesian network as a graph and therefore are of interest.

In [261], Bayesian networks are treated as DAGs. In each iteration of the EA, offspring are produced by mutation. Mutation operators include edge addition/deletion, edge reversal, edge relocation and a knowledge-guided mutation which adds and deletes edges based on minimising the Minimum Description Length (MDL). If an offspring is produced which contains a cycle, it is corrected by deleting the set of edges which induce that cycle.

Bayesian networks have been also been evolved as Completed Partially Directed Acyclic Graphs (CPDAGs) [166]. In this circumstance, individuals in the population represent not only one solution but entire equivalence classes of solutions. Mutation operators may insert and delete both directed and undirected edges, and reverse directed edges. A special mutation operator used appears to resemble a rewrite rule, where a structure of the form $X - Y - Z$ is rewritten to $X \rightarrow Y \leftarrow Z$. In this work, acyclicity is maintained by checking if a mutation would introduce a cycle before it is applied, rather than the correction process used in [261]. Similarly, [47] proposes a variety of EAs learning Bayesian networks represented both as CPDAGs and DAGs, also utilising this operator.

2.5 Conclusions and Directions for Research

In this chapter, we have described the general notion of graph transformation and the rule-based graph programming language GP 2. We have covered GP 2's syntax and semantics and described a simple GP 2 program. We have also given an overall view of evolutionary computation, covering the significant areas of GAs, ES, GP, and Neuroevolution. We have paid particular attention to graph-based EAs, giving detailed descriptions and comparative discussions of CGP, PDGP and NEAT. We have also described a number of other graph-based EAs.

This chapter is a discussion of the context in which this thesis operates. We have seen that graph-based EAs can be used to solve problems in a broad and varied set of domains:

1. Digital circuits [154, 157, 253, 254].
2. Approximate digital circuits [165, 250].
3. Cryptographic circuits [179, 180].
4. Symbolic expressions [155, 188, 189, 253].
5. Forecasting [37, 58, 241, 264].
6. Various forms of automata [118, 177, 206, 232].
7. Sequences of instructions [114, 137].
8. Image processing [88, 89, 201].
9. Video-game agents [120, 121, 210].
10. ANNs [4, 117, 124, 126, 192, 195, 222, 238, 245].
11. CPPNs [73, 74, 94, 220].
12. DNN architectures [136, 153, 226].
13. Bayesian networks [47, 166, 261].

This list is not exhaustive, and there are further domains which have been approached from the perspective of graph-based evolution.

We have also seen that, in many of these works, much of the contribution of the research is to propose new genetic operators over graphs. See, for example, the representation and modification of CGP solutions [157], the history-based genetic crossover used in NEAT [222], or the proposed crossover operator in [177]. We can, therefore, see a clear precedent in the

literature of interest in the general evolution of graphs and therefore the design of genetic operators over graphs. Further, we often see that correctness of evolution is achieved through constraints on the representation [138, 157, 189], or even correction of the phenotype [261].

It is here that we find the intersection between our literature on graph-based evolution and our description of rule-based graph programming. On one hand, we have a clear problem-driven desire to express correct functions over graphs to use as genetic operators. On the other hand, the graph programming language GP 2 provides a concise and formal paradigm to describe relations over graphs. This brings us back to the motivations and aims we set out in Chapter 1, with the literature we have covered justifying our ambitions to design EAs using graph programs as a paradigm for describing genetic operators.

3 Probabilistic Graph Programming

Abstract

To implement probabilistic genetic operators as rule-based graph programs, we require access to a probabilistic variant of graph programming. In this chapter, we describe an extension of GP 2, termed Probabilistic GP 2 (P-GP 2), which supports both probabilistic rule choice and probabilistic matching. We outline the implementation of this extension as a modification of the existing GP 2 compiler. A number of probabilistic graph programs are given: probabilistic graph colouring, Karger's algorithm for graph cutting and two models of random graphs. We stress that these examples are independent of our motivating application to Evolutionary Algorithms (EAs), highlighting the versatility of the work undertaken.

Relevant Publications

Content from the following publications is used in this chapter:

[7] T. Atkinson, D. Plump, and S. Stepney, "Probabilistic graph programming," in *Pre-Proc. Graph Computation Models, GCM 2017,* 2017.

[9] T. Atkinson, D. Plump, and S. Stepney, "Probabilistic graph programs for randomised and evolutionary algorithms," in *Proc. International Conference on Graph Transformation, ICGT 2018*, ser. LNCS, vol. 10887. Springer, 2018, pp. 63–78.

3.1 Introduction

The semantics of GP 2 are non-deterministic in two respects: to execute a rule-set $\{r_1, \ldots, r_n\}$ on a host graph G, any of the rules applicable to G can be picked and applied; and to apply a rule r, any of the valid matches of r's left-hand side in the host graph can be chosen. GP 2's compiler [17] has been designed by prioritising speed over completeness, thus it simply chooses the first applicable rule in textual order and the first match that is found.

For some algorithms, compiled GP 2 programs reach the performance of hand-crafted C programs. For example, [17] contains a 2-colouring program whose run-time on input graphs of bounded degree matches the run-time of Sedgewick's program in Graph Algorithms in C. Clearly, this implementation of GP 2 is not meant to produce different results for the same input or make random choices with pre-defined probabilities.

However, probabilistic choice is a powerful algorithmic concept which is essential to both randomised and Evolutionary Algorithms (EAs). Randomised algorithms take a source of random numbers in addition to input and make random choices during execution. There are many problems for which a randomised algorithm is simpler or faster than a conventional deterministic algorithm [164]. EAs, on the other hand, can be seen as randomised heuristic search methods employing the generate-and-test principle. They drive the search process by variation and selection operators which involve random choices [64]. The existence and practicality of these probabilistic algorithms motivates the extension of graph programming languages to the probabilistic domain. Note that our motivation is different from existing simulation-driven extensions of graph transformation [97, 132]: we propose high-level programming with probabilistic constructs rather than specifying probabilistic models.

To cover algorithms on graphs that make random choices, we define Probabilistic GP 2 (P-GP 2) by extending GP 2 with two constructs: (1) choosing rules according to user-defined probabilities and (2) choosing rule matches uniformly at random.

We present four case studies in which we use P-GP 2 to implement randomised algorithms. The first algorithm is a probabilistic program which produces graph colourings. Empirical data shows that the effectiveness of this program at finding globally optimal solutions decays rapidly as input graphs grow in size. The second example is Karger's randomised algorithm for finding a minimum cut in a graph [115]. The algorithm comes with a probabilistic analysis, which guarantees a high probability that the cut computed by the program is minimal. The third example is sampling from Gilbert's $G(n, p)$ random graph model [75]. The program generates random graphs with n vertices such that each possible edge occurs with probability

p. The final example is sampling from the $D(n, E)$ random directed graph model. The program generates random directed graphs with n vertices and E edges.

This chapter is organised as follows. In Section 3.2 we describe the syntax and semantics of P-GP 2. We also discuss the probabilistic models induced when using P-GP 2, and the implementation of P-GP 2. In Section 3.3 we describe our four example probabilistic graph programs. We draw comparisons with other approaches to probabilistic behaviour in graph transformation in Section 3.4. Finally, we conclude our findings and set out directions for future work in Section 3.5.

3.2 Probabilistic Graph Programming

3.2.1 Syntax and Semantics

We present a conservative extension to GP 2, P-GP 2, where a rule-set may be executed probabilistically by using additional syntax. Rules in the set are picked according to probabilities specified by the programmer, while the match of a selected rule is chosen uniformly at random. When the new syntax is not used, a rule-set is treated as non-deterministic and executed as in GP 2's implementation [17]. This is preferable when executing confluent rule-sets where the discovery of all possible matches is expensive and unnecessary.

To formally describe probabilistic decisions in P-GP 2, we consider the application of a rule-set $\mathcal{R} = \{r_1, \ldots, r_n\}$ to some host graph G. The set of all possible rule-match pairs from $\mathcal{R}$ in G, denoted by $G^{\mathcal{R}}$, is given by

$$G^{\mathcal{R}} = \{(r_i, g) \mid r_i \in \mathcal{R} \text{ and } G \Rightarrow_{r_i,g} H \text{ for some graph } H\}. \tag{3.1}$$

We make separate decisions for choosing a rule and a match. The first decision is to choose a rule, which is made over the subset of rules in $\mathcal{R}$ that have matches in G, denoted by $\mathcal{R}^G$, given by

$$\mathcal{R}^G = \{r_i \mid r_i \in \mathcal{R} \text{ and } G \Rightarrow_{r_i,g} H \text{ for some match } g \text{ and graph } H\}. \tag{3.2}$$

Once a rule $r_i \in \mathcal{R}^G$ is chosen, the second decision is to choose a match with which to apply r_i. The set of possible matches of r_i in G, denoted by G^{r_i}, is given by

$$G^{r_i} = \{g \mid G \Rightarrow_{r_i,g} H \text{ for some graph } H\}. \tag{3.3}$$

We assign a probability distribution (defined below) to $G^{\mathcal{R}}$ which is used to decide particular rule executions. This distribution, denoted by $P_{G^{\mathcal{R}}}$, has to satisfy

$$P_{G^{\mathcal{R}}} : G^{\mathcal{R}} \to [0, 1], \quad \text{such that} \quad \sum_{(r_i,g) \in G^{\mathcal{R}}} P_{G^{\mathcal{R}}}(r_i, g) = 1, \tag{3.4}$$

where $[0, 1]$ denotes the real-valued (inclusive) interval between 0 and 1.

P-GP 2 allows the programmer to specify $P_{G^{\mathcal{R}}}$ by rule declarations in which the rule can be associated with a real-valued positive weight. This weight is listed in square brackets after the rule's variable declarations, as shown in Figure 3.1. This syntax is optional and if a rule's weight is omitted, the weight is 1.0 by default. In the following, we use the notation $w(r)$ for the positive real value associated with any rule r in the program.

```
grow_loop(n:int) [3.0]
```

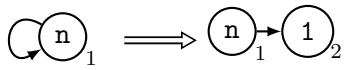


Figure 3.1: A P-GP 2 declaration of a rule with associated weight 3.0. The weight is indicated in square brackets after the variable declaration.

To indicate that the call of a rule-set $\{r_1, \ldots, r_n\}$ should be executed probabilistically, the call is written with square brackets:

$$[r_1, \ldots, r_n]. \tag{3.5}$$

This includes the case of a probabilistic call of a single rule r, written $[r]$, which ignores any weight associated with r and simply chooses a match for r uniformly at random. Given a probabilistic rule-set call $\mathcal{R} = [r_1, \ldots, r_n]$, the probability distribution, $P_{G^{\mathcal{R}}}$, is defined as follows;

The summed weight of all rules with matches in G is

$$\sum_{r_x \in \mathcal{R}^G} w(r_x), \tag{3.6}$$

and the weighted distribution over rules in $\mathcal{R}^G$ assigns to each rule $r_i \in \mathcal{R}^G$ the probability

$$\frac{w(r_i)}{\sum\limits_{r_x \in \mathcal{R}^G} w(r_x)}. \tag{3.7}$$

The uniform distribution over the matches of each rule $r_i \in \mathcal{R}^G$ assigns the probability $1/|G^{r_i}|$ to each match $g \in G^{r_i}$. This yields the definition of $P_{G^{\mathcal{R}}}$ for all pairs $(r_i, g) \in G^{\mathcal{R}}$ given by

$$P_{G^{\mathcal{R}}}(r_i, g) = \frac{w(r_i)}{\sum\limits_{r_x \in \mathcal{R}^G} w(r_x)} \times \frac{1}{|G^{r_i}|}. \tag{3.8}$$

In the implementation of P-GP 2, the probability distribution, $P_{G^{\mathcal{R}}}$, decides the choice of rule and match for $\mathcal{R} = [r_1, \ldots, r_n]$ (based on a random-number generator). Note that this is correctly implemented by first choosing an applicable rule r_i according to the weights and then choosing a match for r_i uniformly at random. The set of all matches is computed at run-time using the existing search-plan method described in [15]. Note that this is an implementation decision that is not intrinsic to the design of P-GP 2.

⟨*Com*⟩ ::= ⟨*RuleSetCall*⟩ | ⟨*ProbRuleSetCall*⟩ | ⟨*GlobalProbRuleSetCall*⟩| ⟨*ProcCall*⟩
| `if` ⟨*ComSeq*⟩ `then` ⟨*Comseq*⟩ [`else` ⟨*ComSeq*⟩]
| `try` ⟨*ComSeq*⟩ [`then` ⟨*Comseq*⟩] [`else` ⟨*ComSeq*⟩]
| ⟨*ComSeq*⟩ '!'
| ⟨*ComSeq*⟩ `or` ⟨*ComSeq*⟩
| `(` ⟨*ComSeq*⟩ `)`
| `break` | `skip` | `fail`

⟨*ProbRuleSetCall*⟩ ::= `[` `RuleId` `]` | `[` [`RuleId` { `,` `RuleId`}] `]`

⟨*GlobProbRuleSetCall*⟩ ::= `[[` `RuleId` `]]` | `[[` [`RuleId` { `,` `RuleId`}] `]]`

Figure 3.2: The modified abstract syntax of P-GP 2's programs (see Figure 2.10). *ProbRuleSetCall* denotes a probabilistic rule-set call, to be executed as we have outlined. *GlobalProbRuleSetCall* denotes a global probabilistic rule-set call, also to be executed as we have outlined.

We also add special syntax to allow a programmer to specify that a uniform distribution should be used across all matches for all rules of a rule-set. If the programmer uses the double square bracket syntax

$$\texttt{[[}r_1, \ldots, r_n\texttt{]]} \tag{3.9}$$

then we ignore rule weights and instead assign $P_{G^{\mathcal{R}}}$ as

$$P_{G^{\mathcal{R}}}(r_i, g) = \frac{1}{|G^{\mathcal{R}}|}. \tag{3.10}$$

We refer to this as a 'global' probabilistic rule-set call.

If a rule-set $\mathcal{R}$ is called using GP 2 curly-brackets syntax, execution follows the GP 2 implementation [17]. Hence our language extension is conservative; existing GP 2 programs will execute exactly as before because probabilistic behaviour is invoked only by the new syntax.

P-GP 2 modifies GP 2's syntax grammar. Figure 3.2 gives the modified parts of the program grammar to include new probabilistic rule-set calls and global probabilistic rule-set calls.

As one final and relatively minor probabilistic extension to GP 2, we also introduce a new integer operator `rand_int(a,b)`. This is called with integer arguments `a` and `b` and returns a random integer drawn from the (inclusive) interval (`a`, `b`). This also requires a modification of GP 2's grammar; in this case, the integer aspects of GP 2's expression grammar. Figure 3.3 shows the updated integer grammar.

⟨*Integer*⟩ ::= Digit {Digit} | `Ivariable` | '-' ⟨*Integer*⟩
| ⟨*Integer*⟩ ⟨*ArithOp*⟩ ⟨*Integer*⟩ | (`indeg` | `outdeg`) `(` Node `)`
| `length(` (AVariable | SVariable | LVariable) `)`
| `rand_int(` ⟨*Integer*⟩ `,` ⟨*Integer*⟩ `)`

Figure 3.3: The modified abstract syntax of P-GP 2's expressions (see Figure 2.7). `rand_int` allows a programmer to sample a uniform distribution over the inclusive range of its 2 input integers.

3.2.2 Existence of a Markov Chain

In this section, we describe how a P-GP 2 program can be interpreted in the context of a Markov chain. We assume a discrete time model for P-GP 2 as we are only concerned with the step-wise operation of a graph program, rather than a specific modelling domain.

It then becomes clear that a rule-set applied to a graph using probabilistic syntax induces a first-order Markov chain. A Markov chain is a model in probability theory where there are transitions between states in a countable set S occurring with fixed probabilities [173, 202]. This is viewed as a Markov process, see Definition 8, over a discrete, countable state space.

Definition 8. ***(Markov process) [173, 202].***
A Markov process is a stochastic process $\mathbb{X} = (X_0, X_1, X_2, ...X_n)$ *consisting of a sequence of random variables where for each random variable* X_i *at time* i*, all future states are conditionally dependent on the current state and independent from previous states:*

$$Pr(X_{i+1} = x \mid X_0 = x_0, X_1 = x_1, X_2 = x_2, ..., X_i = x_i) = Pr(X_{i+1} = x \mid X_i = x_i). \quad (3.11)$$

Definition 9. ***(Markov chain) [173, 202].***
A Markov chain is a Markov process $\mathbb{X} = (X_0, X_1, X_2, ...X_n)$ *on a countable state space* S*, such that each random variable* X_i *at time* i *is a probability distribution over* S*.*

Fixed probabilities mean that the probability of transitioning from one state to another depends only on the current state. The transition probabilities can be represented as a $|S|\times|S|$ transition matrix Q where for any two states $s, s' \in S$, $Q(s, s')$ is the probability of transitioning from state s to state s'. The behaviour of the process can then be simulated by repeatedly multiplying initial distribution X_0, a vector of size $|S|$ describing a probability distribution of the process's initial state, by Q. After n transitions (time steps) this produces

the vector X_n containing as elements the probabilities $X_n(s)$ of being in a state $s \in S$. If S is countable but infinite, there may be no natural representation for Q.

For a rule-set $\mathcal{R}$ applied probabilistically to graph G, the induced Markov chain's state space S is every graph reachable by repeatedly applying $\mathcal{R}$ to G given by

$$S = \{H \mid G \Rightarrow^* H\}. \tag{3.12}$$

For any probabilistic call to rule-set $\mathcal{R}$ and input graph G, the implied state space must be a subset of the set of all possible host graphs considered up to isomorphism: $S \subset \mathcal{G}$. As $\mathcal{G}$ is countable, it entails that S must always be countable. The induced transition matrix Q is defined according to the possible transitions between pairs of graphs $A, B \in S$ and their associated fixed probabilities given by $P_{A^{\mathcal{R}}}$ given by

$$Q(A, B) = \sum_{(r,g) \in A^{\mathcal{R}} \mid A \Rightarrow_{r,g} B', B' \cong B} P_{A^{\mathcal{R}}}(r, g). \tag{3.13}$$

Informally speaking, the transition matrix entry for the transition between graphs A and B is the total probability of A being transformed into B in a single step by probabilistically executing $\mathcal{R}$ on A using any of the matches in $A^{\mathcal{R}}$.

The initial distribution X_0 is a trivial case; the probability of being in initial state G, the host graph, when $\mathcal{R}$ is called, is 1. This means that the initial distribution is defined, for any graph $G' \in S$, as

$$X_0[G'] = \begin{cases} 1 & \text{if } G' = G \\ 0 & \text{otherwise} \end{cases}. \tag{3.14}$$

In special cases, it may be possible to consider transition matrix Q explicitly for a probabilistic rule-set call and find probabilities of its resultant graph accordingly, but more generally the input graph to a program is not known before run-time, preventing pre-computation of state space S and therefore Q. In this case, a step-wise execution of probabilistic rule-set call to produce a result graph can be seen as sampling from the Markov chain induced by the rule-set and host graph. The execution of a probabilistic single rule-set call in P-GP 2 corresponds to a single step of the corresponding induced Markov chain, whereas the as-long-as-possible call $\mathcal{R}!$ corresponds to simulation of the induced Markov chain until reaching some absorbing state (see [173] for more information).

More generally we can consider P-GP 2 programs, rather than single probabilistic rule-set calls. The following sufficient conditions can be used to characterise a P-GP 2 program's behaviour:

1. If a program is terminating and all rule-sets called by the program are (a) called as long as possible, and (b) confluent then the program is deterministic.
2. If all rule-sets called by the program are either (a) called probabilistically, or (b) confluent and called as long as possible then the program forms a Markov chain. The deterministic sub-components of the program form may be treated as part of probabilistic transitions of some previous probabilistic step.

If some rule-sets called by the program are called probabilistically but there are other rule-sets called non-deterministically which are not confluent, then the program forms a Markov Decision Process (see [202]) with non-deterministic sub-components executed according to the implementation of the compiler. If there are no probabilistic rule-set calls in the program and some rule-sets called non-deterministically which are not confluent, the program is in general non-deterministic.

To see that these conditions are sufficient, but not necessary:

1. Consider a program where there are non-confluent rule-sets called non-deterministically, but before each such rule-set call, a confluent rule-set is applied as long as possible which prevents any possible critical pairs of the non-confluent rule-set. Then the program is deterministic despite not meeting the above condition.
2. Consider a program with a loop `(r1; [r2])!`. Then there are examples of `r1`, `r2` where the loop induces a Markov chain when considering resultant graphs up to isomorphism despite containing a non-deterministic rule-set call (the single call to `r1`) which is not executed as long as possible. See Figure 3.9 for such an example.

3.2.3 Implementation of P-GP 2

Our implementation of P-GP 2 is a modification of the existing GP 2 compiler generating C code described in [15]. In this section, we outline how the new features are implemented.

Probabilistic rule-set calls

The existing compiler uses the searchplan method for matching rules. As the specification of GP 2 has that rule-sets and rules are executed non-deterministically, the existing compiler chooses the first valid rule and match found for efficiency. We retain the existing implementation of this searchplan method and instead simply continue the search until the graph has been exhaustively explored. Note that this method retains the same worst case time complex-

Algorithm 4 Pseudocode for probabilistically picking a rule when a probabilistic rule-set call is made.

```
1: procedure PickRule(R: rule-set, w: weight function)
2:     valid_rules ← []
3:     for r ∈ R do
4:         if searchplan(r) → first ≠ NULL then
5:             append r to valid_rules
6:         end if
7:     end for
8:     if |valid_rules|= 0 then
9:         return NULL
10:    else
11:        rule ← probabilistic weighted choice over valid_rules according to w
12:        return rule
13:    end if
14: end procedure
```

ity as the original implementation, but will always cost that complexity. In the pseudocode listings used In this section, we will refer to the search plan method for a given rule `r` as an iterable linked list searchplan(`r`) where `match` ← searchplan(`r`).`first` gives the first match found of `r` in the host graph and each `match` returned has `match.next` giving the next match found. If there are no matches of `r`, searchplan(`r`).`first` returns `NULL` and if a given match `match` is the final match found in the host graph, `match.next` also returns `NULL`.

The pseudocode of the implementation of probabilistically picking a rule is given in Algorithm 4. Here the procedure is passed a rule-set $\mathcal{R}$ and weight function `w` and identifies the set of rules with valid matches, denoted `valid_rules`. Then the procedure picks one such valid rule according to the weighted distribution given by `w`. If there are no valid rules, then the procedure returns `NULL`.

Once a rule `r` has been chosen (or the size of the rule-set is 1), then the method for probabilistically choosing a match for `r` is implemented according to the pseudocode given in Algorithm 5. The procedure is passed a rule `r` and proceeds to iterate over its searchplan searchplan(`r`) until no more matches are found. Each match found is stored in `valid_matches`, which is drawn from uniformly at random when the host graph has been exhaustively searched. If no valid matches are found, then the procedure returns `NULL`.

Algorithm 5 Pseudocode for probabilistically picking a match for a rule.

```
1:  procedure PickMatch(r: rule)
2:      valid_matches ← []
3:      match ← searchplan(r).first
4:      while match ≠ NULL do
5:          append match to valid_matches
6:          match ← match.next
7:      end while
8:      if |valid_matches|= 0 then
9:          return NULL
10:     else
11:         match ← probabilistic uniform choice over valid_matches
12:         return match
13:     end if
14: end procedure
```

The implementation of the global probabilistic rule-set calls (called with double square brackets) follows the implementation of the single rule's matching algorithm given in Algorithm 5. However, the list of valid matches is constructed over all rules in the rule-set, rather than just a single rule `r`.

3.3 Example Probabilistic Graph Programs

3.3.1 Probabilistic Vertex Colouring

In this section, we discuss a probabilistic version of a very simple non-deterministic vertex colouring program `VC` (taken from [183]). Computing a vertex colouring that uses the minimal number of colours is an NP-complete problem [209], the program `VC` only guarantees to compute some colouring but does this in polynomial time. We discuss the behaviour of `VC` under P-GP 2 on members of a problem set, grid graphs, that have known optimal colourings.

Grid graphs

In a grid graph, nodes are organised in a square lattice. We give direction to grid graphs by allocating one node as a source with all edges directed outwards from that source. Figure 3.4

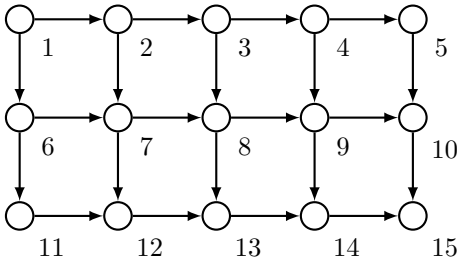


Figure 3.4: $GG_{5,3}$: a 5×3 grid graph

```
Main := mark!; init!; [inc]!
```

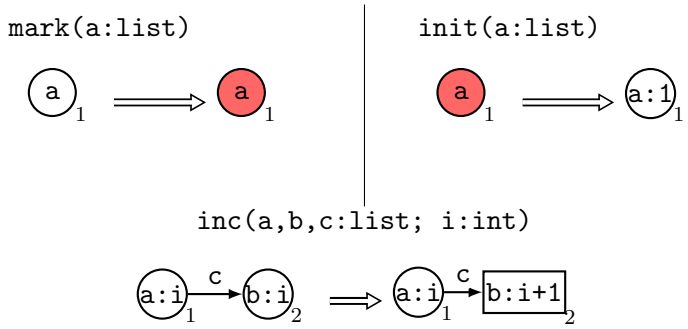


Figure 3.5: The probabilistic vertex colouring program `VC`.

shows a 5×3 grid graph with node 1 as its source. Let GG be the family of all unlabelled directed grid graphs and $GG_{x,y}$ specifically refer to an $x \times y$ unlabelled directed grid graph.

In this case study, we discuss the likelihood of `VC` producing an optimal colouring over GG in terms of parameters x and y. We choose GG as a motivating example as each of its members has a known optimal colouring. Using 2 colours, a grid graph can be coloured in a checkerboard fashion, and GG is, therefore, a family of bipartite graphs.

Vertex colouring program `VC`

Figure 3.5 shows our vertex colouring program `VC`. The colour assigned to a particular node is an integer which is appended to the node's existing label. The first part of the program, `mark!;init!`, is deterministic, assigning to each node the colour `1`. The second part of the program, the loop `inc!`, is terminating but highly probabilistic, matching adjacent pairs of nodes that are identically coloured and incrementing the colour of the target node of the

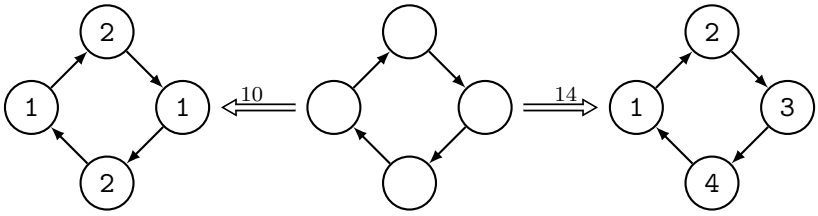


Figure 3.6: Applying the probabilistic vertex colouring program `VC` to a simple 4-node cycle. The outcome on the left is an optimal colouring, produced after 10 rule applications. The outcome on the right is the worst possible colouring, produced after 14 rule applications.

matched connecting edge.

The time complexity of the initial part is quadratic in the number of host graph nodes; both `mark` and `init` are applied to each node once and finding a match for either rule requires a single search over all nodes. It can be shown that the number of `inc` applications is quadratic in the number of host graph nodes [183]. Moreover, the compiled P-GP 2 code will find a match of `inc` in linear time in the worst case by searching once over all edges in the host graph. Therefore the run-time of the loop `inc!` is cubic in the size of the host graph and hence `VC`'s overall time complexity is also cubic.

To highlight the effect of program derivation on outcome, consider Figure 3.6 which shows two executions of `VC` on a small host graph. Whereas the left execution produces an optimal colouring with 10 rule applications, the right execution returns the worst colouring after 14 steps.

Behaviour of `VC` on grid graphs

We study the *likelihood of optimal colouring* for a set of inputs, given as the cumulated probability of samples producing an optimal colouring, which in this scenario corresponds to samples producing a 2 colouring. Table 3.1 shows the observed behaviour of `VC` over grid graphs with width in the integer interval $[1, 3]$ and height in $[1, 5]$. Each result is given as a real which describes the observed probability of samples for that input which returned 2-coloured result graphs.

As an observation, the likelihood of generating a 2 colouring for a grid graph appears greatly reduced as the graph grows in width and height. This is perhaps unsurprising, given

		width		
		1	2	3
height	1	1.0	1.0	0.5
	2	1.0	0.25	0.16
	3	0.5	0.16	0.04
	4	0.5	0.05	5e-3
	5	0.25	0.02	1e-4

Table 3.1: Results from sampling the vertex colouring program on grid graphs for derivations producing optimal colourings. Each entry represents the proportion of samples observed returning optimal colourings.

that vertex colouring is an NP-hard problem. But as it is known that grid graphs are trivially 2-colourable, this result highlights that a naive probabilistic approach to vertex colouring is highly ineffective on certain classes of input graphs.

3.3.2 Karger's Minimum Cut Algorithm

Karger's contraction algorithm [115] is a randomised algorithm that attempts to find a minimum cut in a graph G, that is, the minimal set of edges to delete to produce two disconnected subgraphs of G. The contraction procedure repeatedly merges adjacent nodes at random until only two remain. As this algorithm is designed for undirected multi-graphs (without loops), we model an edge between two nodes as two directed edges, one in each direction. For visual simplicity, we draw this as a single edge with an arrow head on each end. We assume that input graphs are unmarked, contain only simulated directed edges, and are connected. We also assume that edges are labelled with unique integers, as this allows us to recover the cut from the returned solution.

Figure 3.7 shows a P-GP 2 implementation of this contraction procedure. This program repeatedly chooses an edge to contract at random using the `pick_pair` rule, which marks the surviving node `red` and the node that will be deleted `blue`. The nodes' common edges are deleted by `delete_edge` and all other edges connected to the `blue` node that will be deleted are redirected to connect to the `red` surviving node by `redirect`. In the final part of the loop, `cleanup` deletes the `blue` node and unmarks the `red` node. This sequence is applied as long as possible until the rule `three_node` is no longer applicable; this rule is an identity rule ensuring that a contraction will not be attempted when only 2 nodes remain. The final

`Main` := (`three_node`; [`pick_pair`]; `delete_edge`!; `redirect`!; `cleanup`)!

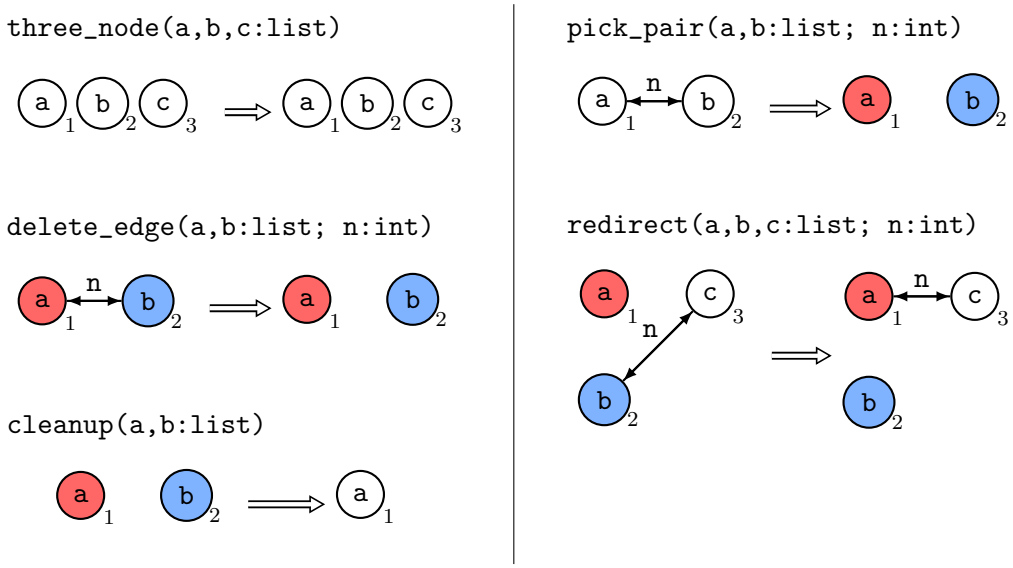


Figure 3.7: The contraction procedure of Karger's algorithm implemented in P-GP 2.

graph produced by this algorithm represents a cut, where the edges between the 2 surviving nodes are labelled with integers. The edges with corresponding integer labels in the input graph are removed to produce a cut.

Karger's analysis of this algorithm finds a lower bound for the probability of producing a minimum cut. Consider a minimum cut of c edges in a graph of n nodes and e edges. The minimum degree of the graph must be at least c, so $e \geq \frac{n.c}{2}$. If any of the edges of the minimum cut are contracted, that cut will not be produced. Therefore the probability of the cut being produced is the probability of not contracting any of its edges throughout the algorithm's execution. The probability of picking such an edge for contraction is

$$\frac{c}{e} \leq \frac{c}{\frac{n.c}{2}} = \frac{2}{n}, \tag{3.15}$$

and therefore the probability p_n of never contracting any edge in c is given by

$$p_n \geq \prod_{i=3}^{n} 1 - \frac{2}{i} = \frac{2}{n(n-1)}. \tag{3.16}$$

For example, applying Karger's algorithm to the host graph G shown in Figure 3.8 can produce one possible minimum cut (cutting 2 edges), which happens with probability greater than or equal to $\frac{1}{28}$. By using rooted nodes (see [17]) it is possible to design a P-GP 2 program

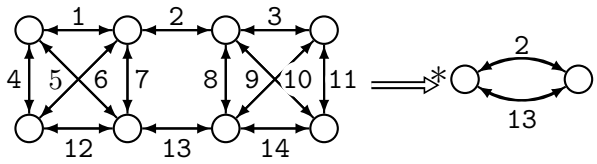


Figure 3.8: Karger's contraction algorithm applied to a simple 8-node graph to produce a minimal cut. The probability of producing this minimal cut is at least $\frac{1}{28}$; our implementation generated this result after seven runs.

that executes this algorithm on a graph with edges E in $O(|E|^2)$ time, with `pick_pair` being the limiting rule taking linear time to find all possible matches, applied $|E|-2$ times.

3.3.3 $G(n,p)$ model for Random Graphs

The $G(n,p)$ model [75] is a probability distribution over graphs of n vertices where each possible edge between vertices occurs with probability p. Here we describe an algorithm for sampling from this distribution for given parameters n and p. This model is designed for simple graphs and so we model an edge between two nodes, in a similar manner to that used in Karger's algorithm, as two directed edges, one in each direction.

As we are concerned with a fixed number of vertices n, we assume an unmarked input graph with n vertices and for each pair of vertices v_1, v_2 exactly one edge with v_1 as its source and v_2 as its target – effectively a fully connected graph with two directed edges simulating a single undirected edge. Then $G(n,p)$ can be sampled by parameterising the GP 2 algorithm given in Figure 3.9 by p. In this algorithm, every undirected edge in the host graph is chosen non-deterministically by `pick_edge`, marking it `red`. Then this edge is either kept and marked `blue` by `keep_edge` with probability p or it is deleted by `delete_edge` with probability $1-p$. After all edges have either been deleted or marked `blue`, `unmark_edge` is used to remove the surviving edges' marks. By applying this algorithm, each possible edge is deleted with probability $1-p$ and hence occurs with probability p, sampling from the $G(n,p)$ model.

Sampling from the $G(n,p)$ model yields a uniform distribution over graphs of n nodes and M edges and each such graph occurs with probability

$$p^M(1-p)^{\binom{n}{2}-M}. \tag{3.17}$$

Figure 3.10 shows a possible result when applying this algorithm to a 4-node input with $p = 0.4$.

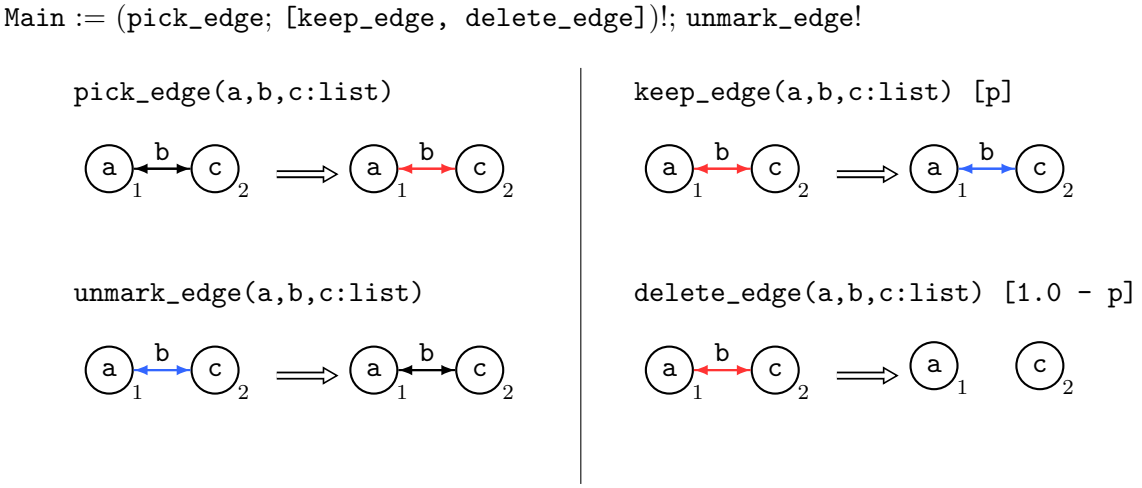


Figure 3.9: P-GP 2 program for sampling from the $G(n, p)$ model for some probability `p`. The input is assumed to be a connected unmarked graph with n vertices.

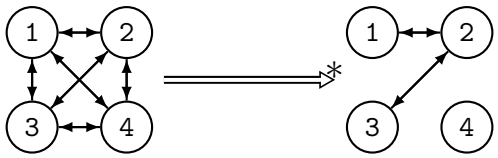


Figure 3.10: The $G(n, p)$ program applied to a complete 4-node graph with $p = 0.4$. The probability of producing this result is 0.0207.

`Main` := (`continue`; `[[add_edge, add_loop]]`)!; `clean`

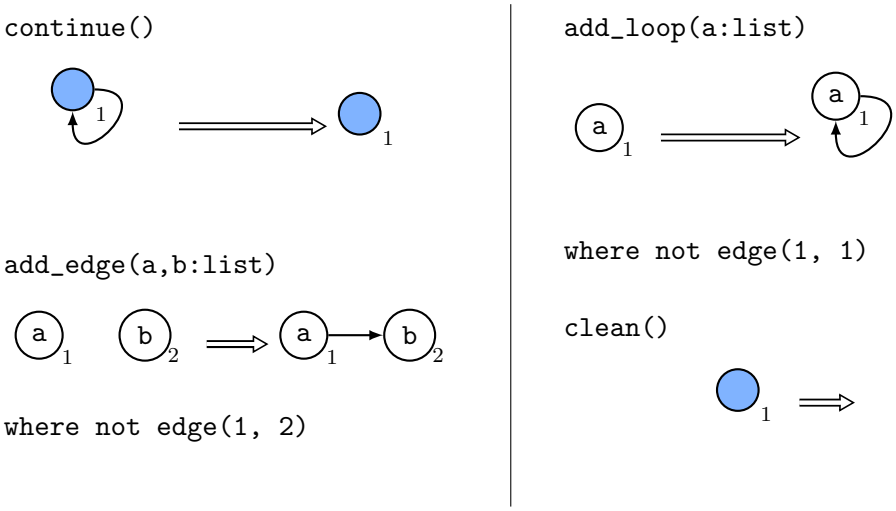


Figure 3.11: P-GP 2 program for sampling from the $D(n, E)$. The input is assumed to be an input graph with n unmarked vertices and a single `blue` marked vertex with E loops.

3.3.4 $D(n, M)$ model for Directed Random Graphs

The $D(n, M)$ model is a directed random graph model [82] giving a probability distribution over graphs of n vertices with M randomly distributed edges. Here we describe an algorithm for sampling from this distribution for given parameters n and M.

As we are concerned with a fixed number of vertices n and edges M, we assume an input graph with n unmarked vertices and a single `blue` marked vertex with M loops. Then $D(n, M)$ can be sampled by using the P-GP 2 program given in Figure 3.11. In the main loop of the algorithm, first, the rule `continue` is applied. This rule deletes a loop from the single `blue` marked node. This ensures that the program terminates once all M edges have been added. Then the rule-set `[[add_edge, add_loop]]` is applied, inserting a new edge uniformly at random over all places where an edge does not exist. Here we use a global probabilistic rule-set call (called with double square brackets) to ensure that the distribution is uniform over both loops and edges. Once all edges have been added and `continue` fails, the rule `clean` is applied to remove the `blue` node. The resultant graph has been sampled from the $D(n, M)$ model.

Sampling from the $D(n, E)$ model yields a uniform distribution over graphs of n nodes and

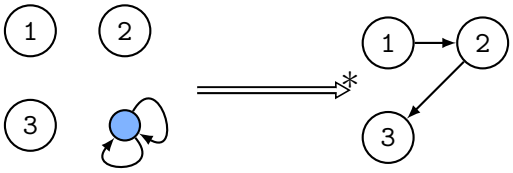


Figure 3.12: The $D(n, E)$ program applied to an input graph with $n = 3$ and $M = 2$. The probability of producing this result is $\frac{1}{36}$.

Model	**Rule and Match Choice**	**Time Model**	**Probabilistic Rule Execution**
Classical Graph Transformation, Rule-set Calls in GP 2	Non-deterministic	—	No
Probabilistic Graph Transformation [132]	Non-deterministic	Discrete	Yes
Stochastic Graph Transformation [97]	Probabilistic	Continuous	No
Probabilistic Rule-set Calls in P-GP 2, `ppick` in Porgy	Probabilistic	Discrete	No

Table 3.2: Different approaches to decision making in graph transformation.

M edges and each such graph occurs with probability equal to

$$\binom{n^2}{M}^{-1}. \tag{3.18}$$

Figure 3.12 shows a possible result when applying this algorithm to an input graph with $n = 3$ and $M = 2$. The probability of producing this result is $\frac{1}{36}$.

3.4 Related Work

In this section, we address three other approaches to graph transformation which incorporate probabilities. All three aim at modelling and analysing systems rather than implementing algorithms by graph programs, which is our intention. Table 3.2 gives a concise description of our comparison.

The port graph rewriting framework PORGY [67] allows to model complex systems by

```
probability_edge(a,b,c:list)
```

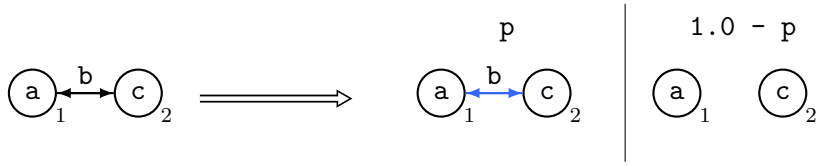


Figure 3.13: A PGTS rule with multiple right-hand sides. The probability of each right-hand side is the value given above it.

transforming port graphs according to strategies formulated in a dedicated language. Probability distributions similar to those in this paper can be expressed in PORGY using the `ppick` command which allows probabilistic program branching, possibly through external function calls.

Stochastic Graph Transformation Systems [97] (SGTSs) are an approach to continuous-time graph transformation. Rule-match pairs are associated with continuous probability functions describing their probability of executing within a given time window. While the continuous time model is clearly distinct to our approach, the application rates associated with rules in SGTSs describe similar biases in probabilistic rule choice as our approach.

Closest to our approach are Probabilistic Graph Transformation Systems (PGTSs) [132]. This model assumes non-deterministic choice of rule and match as in conventional graph transformation, but executes rules probabilistically. In PGTSs, rules have single left-hand-sides but possibly several right-hand sides equipped with probabilities. This mixture of non-determinism and probabilistic execution gives rise to Markov decision processes. There are clear similarities between our approach and PGTSs: both operate in discrete steps and both can express non-determinism and probabilistic behaviour. However, PGTSs are strict in their allocation of behaviour; rule and match choice is non-deterministic and rule execution is probabilistic. In our approach, a programmer may specify that a rule-set is executed in either manner. It seems possible to simulate (unnested) PGTSs in our approach by applying a non-deterministic rule-set that chooses a rule and its match followed by a probabilistic rule-set which executes one of the right-hand sides of this rule. For example, the first loop in the $G(n, p)$ program in Figure 3.9 simulates a single PGTS rule; `pick_edge` non-deterministically chooses a match, and `[keep_edge, delete_edge]` probabilistically executes some right-hand side on the chosen match. Figure 3.13 visualises this single PGTS rule.

Other approaches to probabilistic graph rewriting include [21], where a rule-algebra frame-

work is proposed for the study of stochastic rewrite systems, and [49], where stochastic rewrite systems are used in the simulation and study of molecular biological systems. However, direct comparison with these works is difficult due to differences in the representation of rewriting and design decisions driven by the intended application areas.

3.5 Conclusions and Future Work

In this chapter we have described the probabilistic graph programming language P-GP 2. P-GP 2 is an extension to GP 2 that allows a programmer to specify probability distributions over the outcomes of rule-set calls through the use of rule weights and new syntax. A programmer can specify a weighted decision over rule-choice followed by a uniform decision over match-choice, or simply a uniform distribution over all matches for all rules. We have demonstrated the versatility of P-GP 2 by implementing 4 randomised graph algorithms.

There are a number of possible directions for future work on P-GP 2. We would like to explore which algorithms from the areas of randomised graph algorithms and random graph generation can be described in P-GP 2. Obvious examples include randomised algorithms for checking graph connectedness [164], generating minimum spanning trees [116] and generating random graphs according to the model of [65]. Additionally, it would be interesting to investigate the efficiency of using incremental pattern matching [23] in the implementation as an alternative method for identifying all matches. Incremental pattern matching stores all matches of a rule in a table which is edited every time the host graph is modified. As rules naturally entail small local rewrites, there are likely many cases where incremental pattern matching improves the performance of P-GP 2 as an alternative to finding all matches for a rule in each probabilistic rule call. An additional area of potential research is investigating whether some randomised graph algorithms cannot be readily expressed in which case we can ask: what further extensions to P-GP 2's syntax are necessary to make them expressible?

4 Function Graphs

Abstract

In this chapter, we identify a class of graphs, Function Graphs (FGs), which can represent a number of application domains of interest: digital circuits, symbolic expressions and Artificial Neural Networks (ANNs). These graphs concisely and directly describe acyclic (feed-forward) and cyclic (recurrent) programs with arbitrary numbers of inputs and outputs. Alongside example graphs in each of the listed domains, we discuss FGs' general semantics.

4.1 Introduction

Representation is crucial in computer science, and an important specific representation is the graph. Graphs are used in a wide range of applications and algorithms, see for example [44, 109, 209]. In Evolutionary Algorithms (EAs), graphs are used in some applications, but are usually encoded in a linear genome, with the genome undergoing mutation and crossover, and a later "genotype to phenotype mapping" used to decode the linear genome into a graph structure. For example in Cartesian Genetic Programming (CGP) [155, 157], the connections of feed forward networks are encoded in a linear genome. Neuroevolution of Augmenting Topologies (NEAT) [221, 222] provides a linear encoding of Artificial Neural Networks (ANNs) which are seen as graph structures. Trees (a subset of more general graphs) are also used in EAs. Grammatical Evolution [175, 199] uses a linear genome of integers to indirectly encode programs. Genetic Programming (GP) [129, 131] is unusual for an EA: rather than using a linear genome, it typically uses direct manipulation of abstract syntax trees. Poli [188, 189] uses a 'graph on a grid' representation: the underlying structure is a graph, but the nodes are constrained to lie on discrete grid points. Multiple Interactive Outputs in a Single Tree (MIOST) [138] proposes using trees with multiple output nodes and sharing to extend traditional GP to domains where problems have multiple, related outputs. Pereira et al [177] represent Turing machines as graphs encoded in a linear genome, and develop a crossover operator based on the structure of the underlying graph.

There are arguments for and against linear genomes representing graphs. While standard in EAs, able to exploit the knowledge about evolutionary operators, they can hide the problem's underlying structure and can have biases in the effect of evolutionary operators. There may be advantages in evolving graphs directly, rather than via linear genome encodings or 2D grid encodings, and defining mutation operators that respect the graph structure.

To this end, Function Graphs (FGs) are utilised as a generic representation of programs for the purposes of evolution throughout the rest of this thesis. This chapter describes our notion of FGs and discusses how they can be encoded in P-GP 2. We stress the obvious similarity between FGs and term graphs [181]. It would, therefore, be possible to evaluate FGs with term graph rewriting for both Acyclic FGs, see [181], and Recurrent FGs, see [14], but for simplicity we will in general describe FGs in the sense of data-flow diagrams.

In Section 4.2 we discuss FGs informally and give numerous example graphs in multiple domains. Section 4.3.1 gives a description of FGs and their semantic behaviour. Finally, in Section 4.4, we conclude our discussion and propose possible extensions to FGs.

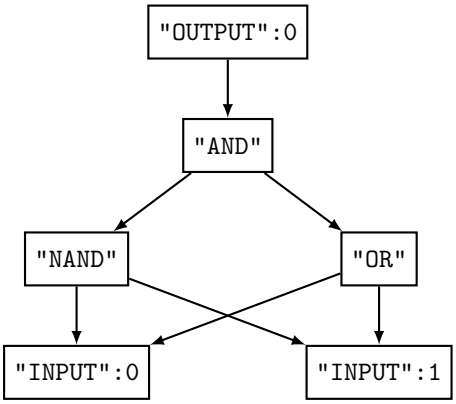


Figure 4.1: An example FG implementing an XOR gate from NAND, OR and AND gates.

4.2 Intuition and Example Function Graphs

An FG is a graph consisting of:

- Input nodes. These are explicitly ordered nodes which allow global inputs to be loaded into the FG. All input nodes have no outgoing edges.
- Function nodes. These are nodes which compute functions on their local inputs. Their local inputs are given by their outgoing edges. The number of outgoing edges of a function node is equal to the arity of its associated function.
- Output nodes. These are explicitly ordered nodes which return global outputs from the FG. All output nodes have no incoming edges.

Each function node is associated with a function from some predefined function set. When an FG is presented with an input, that input is loaded into the input nodes. Then each function node is evaluated by applying its associated function to its inputs. Finally, the FG returns an output according to the values of its output nodes. In general an FG need not be acyclic nor connected.

Consider, for example, the simple FG shown in Figure 4.1. This FG implements an XOR ($\oplus$) gate with truth table:

i_0	i_1	$o_0 = i_0 \oplus i_1$
0	0	0
0	1	1
1	0	1
1	1	0

(4.1)

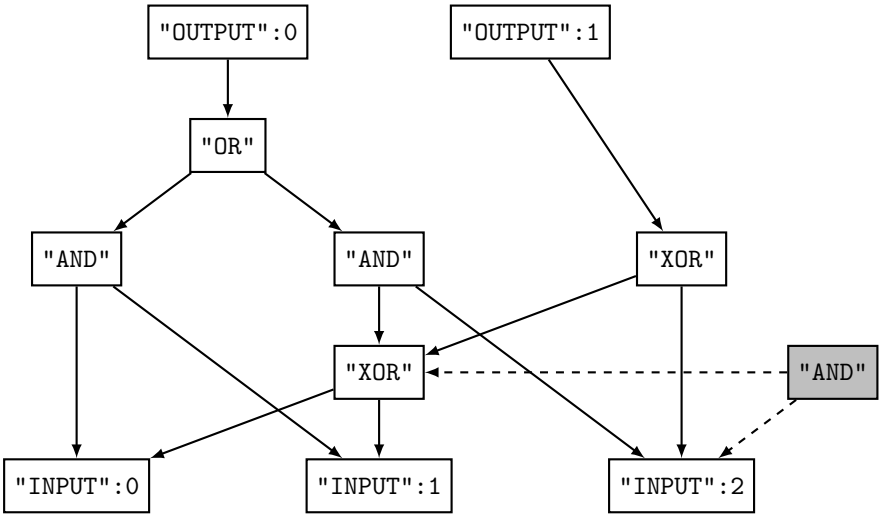


Figure 4.2: An example FG implementing a 1-bit adder from XOR, AND and OR gates.

The single output o_0 is given by the node labelled `"OUTPUT":0`. This node's single outgoing edge indicates that o_0 returns the value associated with the node labelled `"AND"`. This node computes AND ($\wedge$) of its two inputs, which are given by the node's two outgoing edges. The first of these inputs is the node labelled `"NAND"` which computes NAND ($\uparrow$) of its two inputs, which are given by the two input nodes labelled `"INPUT":0` and `"INPUT":1` which correspond to inputs i_0 and i_1. The second of these inputs is the node labelled `"OR"` which computes OR ($\vee$) of these same input nodes. Hence the overall semantics of the FG is given as

$$o_0 = (i_0 \uparrow i_1) \wedge (i_0 \vee i_1) = i_0 \oplus i_1. \tag{4.2}$$

It is worth stressing the meaning of edge direction in our function graphs. Where an edge exists from node v_1 to node v_2, we take that to mean that node v_1 uses node v_2 as input. This is in-line with convention in term graph rewriting [181]. In contrast, other approaches such as PDGP [186] and NEAT [222], understand an edge from node v_1 to v_2 to mean a flow of data from node v_1 to node v_2 in the manner of a data-flow diagram.

In the following subsections we use examples to demonstrate various features of FGs. First, we see a 1-bit adder demonstrating the use of multiple outputs and intronic material. We then describe an FG implementing Newton's Law of Gravitation, highlighting the ordering on edges that prevents ambiguity with non-commutative functions. An FG implementing the Fibonacci sequence introduces the notion of recursive edges, allowing the FG representation to describe stateful programs. Finally, we see a simple neural network, highlighting the existence

of weighted edges that determine the strength of connections throughout the graph.

4.2.1 1-Bit Adder: Multiple Outputs and Intronic Material

Figure 4.2 shows an FG implementing a 1-bit adder from XOR, AND and OR gates. The purpose of a 1-bit adder is to take in input bits a and b and carry bit c and compute $a+b+c$ represented as a pair of output bits o_0, o_1. The truth table of a 1-bit adder is:

$i_1 = a$	$i_2 = b$	$i_3 = c$	o_0	o_1	Description
0	0	0	0	0	$0+0+0=0$
0	0	1	0	1	$0+0+1=1$
0	1	0	0	1	$0+1+0=1$
0	1	1	1	0	$0+1+1=2$
1	0	0	0	1	$1+0+0=1$
1	0	1	1	0	$1+0+1=2$
1	1	0	1	0	$1+1+0=2$
1	1	1	1	1	$1+1+1=3$

(4.3)

For our FG, shown in Figure 4.2, we have that

$$o_0 = ((i_0 \oplus i_1) \wedge i_2) \vee (i_0 \wedge i_1), \tag{4.4}$$

i.e., output o_0 returns 1 when at least 2 inputs are equal to 1. We also have that

$$o_1 = (i_0 \oplus i_1) \oplus i_2, \tag{4.5}$$

i.e. output o_1 returns 1 when either 1 or 3 inputs are equal to 1. Hence our FG correctly implements the 1-bit adder function; when 0 inputs are equal to 1, the FG returns 0 (00). When 1 input is equal to 1, the FG returns 1 (01). When 2 inputs are equal to 1, the FG returns 2 (10). And when 3 inputs are equal to 0, the FG returns 3 (11).

There are a few noteworthy features in this example FG. Firstly, an FG can have multiple outputs; in this case each output corresponds to a different output bit of a digital circuit. Secondly, an FG can have *intronic* material. Referring to Figure 4.2, consider the node labelled `"AND"` and coloured grey towards the right-hand-side of the diagram. There is no path from either output node to this node, and it is therefore impossible for this node to contribute to the semantics of the overall graph. It is therefore possible to remove or relabel this node, or redirect its edges while preserving the semantics of the FG. Similarly, it is possible to insert new nodes for which there is no path from either output node, again preserving semantics.

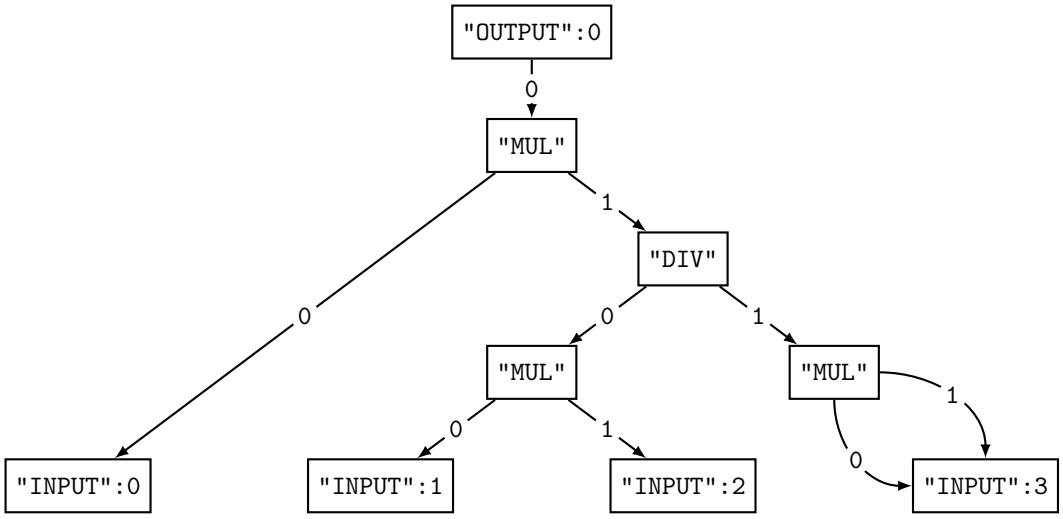


Figure 4.3: An example FG using multiplication (`"MUL"`) and division (`"DIV"`) nodes implementing Newton's Law of Gravitation $F = Gm_1m_2/r^2$ with $i_0 = G$, $i_1 = m_1$, $i_2 = m_2$ and $i_3 = r$.

In general, any node for which there is no incoming path from an output node is referred to as *neutral*, *intronic* or *inactive*. Similarly, any such node's outgoing edges are also described as neutral, inactive or intronic. This neutral material corresponds to 'garbage' in term graph rewriting [181].

4.2.2 Newton's Law of Gravitation: Ordered Edges

In the previous examples, all of the functions associated with function nodes are commutative. That is, treating each 2-input logic gate we have used as a function $F(a, b)$ of input bits a, b it always holds that

$$F(a, b) = F(b, a). \tag{4.6}$$

However there are many functions of interest which are non-commutative. A classic example is the logical implication operator $\Rightarrow$. For input bits a, b we have that

a	b	$a \Rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

(4.7)

where $a \Rightarrow b \neq b \Rightarrow a$ for $a = 0$, $b = 1$. We want the execution of our FGs to be unambiguous, and the diagrams in Figures 4.1 and 4.2 do not provide clear orderings on edges. Consider, for example, the node labelled `"AND"` in Figure 4.1; if this node were instead associated with a logical implication $\Rightarrow$ then the semantics of the program would not be clearly defined, with the interpretation of the ordering of the node's outgoing edges determining the behaviour of the program.

In general, edges in FGs *are* ordered. Consider the FG shown in Figure 4.3. This FG implements Newton's Law of Gravitation

$$F = G\frac{m_1 m_2}{r^2}, \tag{4.8}$$

using multiplication and division nodes. Here, the outgoing edges of each node are ordered according to integer labels. For example, the sole division node labelled `"DIV"` takes as inputs the results of computations $m_1 m_2$ and r^2. The edge labelled `0` indicates that $m_1 m_2$ is the first input presented to the function node, whereas the edge labelled `1` indicates that r^2 is the second input presented to the function node. Hence the division node computes $m_1 m_2 / r^2$ rather than $r^2 / m_1 m_2$.

In general, if a function node has n outgoing edges, $e_0, \ldots, e_{n-1}$, then these edges are explicitly ordered by labelling each edge with a unique integer, $0, \ldots, n-1$. Note that in the previous examples, this ordering information was implicitly present. However, as all functions used were commutative, it was unnecessary to explicitly present this information. In general, edges in FGs are always ordered, but the integers encoding the orderings are only shown when at least one function used is non-commutative.

4.2.3 Fibonacci Sequence: Recurrent Edges and Stateful Programs

Every FG covered so far has been *acyclic*. That is, in each graph covered, for every path $v_1 \to v_2$, there does not exist a path $v_2 \to v_1$. This conceptually works for problems where the task is to learn or represent some mapping of inputs to outputs. However, in general, FGs may be considered *stateful*, and are able to access their internal states via *recurrent* edges. The edges we have seen so far are *non-recurrent* in that they provide their associated function nodes with the values produced by their targets in the current execution step of the FG. A recurrent edge instead returns that value produced by its target in the previous execution step of the FG.

Consider the FG given in Figure 4.4. When this FG is driven with input $i_0 = 1$ for n time

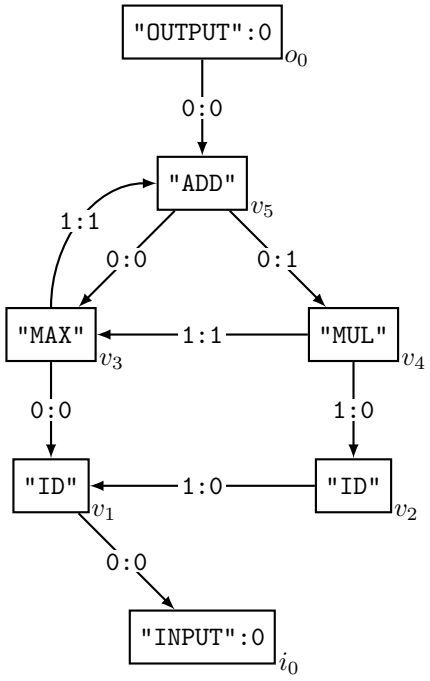


Figure 4.4: An example FG generating the Fibonacci sequence using 1-input identities (`"ID"`), a 2-input max function (`"MAX"`), addition (`"ADD"`) and multiplication (`"MUL"`). The input i_0 is assumed to be fixed at 1 across all execution steps.

steps, it generates the Fibonacci sequence, $Fib(0), Fib(1), \ldots, Fib(n)$ with

$$Fib(x) = \begin{cases} 1, & \text{if } x < 2 \\ Fib(x-1) + Fib(x-2), & \text{otherwise.} \end{cases} \tag{4.9}$$

We can now see that each edge is associated with a *pair* of integers represented as a list. For a given edge, the label $a : b$ can be read as a indicating that the edge is recurrent ($a = 1$) or non-recurrent ($a = 0$) and b indicating the order of the edge (as described in the previous section. Although all functions used in the example are commutative, we here include the ordering information to avoid confusion about the role of the integers encoding the recurrence property.

In this example each node has been given an identifier (shown to the bottom right of each node). This is not part of the representation, and simply serves as a reference for the following description of the program. Now consider the 2-input max function v_3. This function takes inputs from v_1 and v_5. The edge $v_3 \to v_1$ is labelled `0:0`. The first `0` indicates that this edge is non-recurrent. The second `0` indicates that this edge is treated as the first input to the

node. The edge $v_3 \to v_5$ is labelled `1:1`. The first `1` indicates that this edge is recurrent, and therefore returns the *previous* value associated with v_5. The second `1` indicates that this edge is treated as the second input to the node.

Hence by using recurrent edges, it becomes possible for an FG to use values computed in previous execution steps. If we assume that at all time steps, $i_0 = 1$ and the initial value associated with each node is 0, the behaviour of the example FG can be mapped through time as follows:

$Time$	i_0	v_1	v_2	v_3	v_4	v_5	o_0	$Description$
-1	$-$	0	0	0	0	0	$-$	*Initial state.*
0	1	1	0	1	0	1	1	$Fib(0)$
1	1	1	1	1	0	1	1	$Fib(1)$
2	1	1	1	1	1	2	2	$Fib(2)$
3	1	1	1	2	1	3	3	$Fib(3)$
4	1	1	1	3	2	5	5	$Fib(4)$
5	1	1	1	5	3	8	8	$Fib(5)$
6	1	1	1	8	5	13	13	$Fib(6)$
					$\dots$			

(4.10)

In initial state, every node's value is 0. In the first execution step, $v1$'s and hence v_3's values are updated to 1, causing $o_0 = v_5$ to return $1 = Fib(0)$. By the third executions step, both v_3 and v_4 return 1, causing $o_0 = v_5$ to return $2 = Fib(2)$. From then on, the FG simply computes the addition of its previous 2 outputs, with nodes v_4 and v_3 with recurrent connections $v_4 \to v_3$ and $v_3 \to v_5$ serving as form of memory.

As before with the ordering edges, if all edges in a graph are non-recurrent (or indeed, recurrent) then this information may not be visually shown. In all 3 previous examples (Figures 4.1, 4.2 and 4.3), all edges were non-recurrent. Implicitly, this information was present, but it was unnecessary to show it explicitly.

4.2.4 A Simple Neural Network: Weighted Edges and Biased Nodes

In each of the previous examples, edges were assumed to be performing as a form of 'identity' where they would present exactly the value of their targets to their associated function node. In general, edges may be associated with *weights*. The weight associated with an edge describes the strength (and sign) of the connection between two function nodes. We refer to the weight of an edge e as $w(e)$.

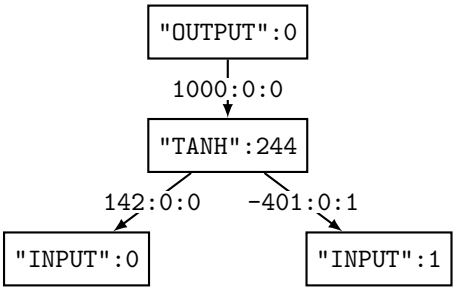


Figure 4.5: An example FG implementing a simple neural network. Nodes labelled `"TANH"` : x are tanh neurons with a bias of $\frac{\texttt{x}}{1000}$.

Similarly, nodes were assumed to have no associated local constants. In general, nodes may be associated with *biases*. The bias of a function node provides a constant value for use within the node's computation. We refer to the bias of a function node v as $b(v)$.

We give an example FG with weights and biases in Figure 4.5. This FG consists of 2 input nodes, 1 function node and 1 output node. The single function node has an edge to each input. Consider the edge labelled `142:0:0`; here, the `0:0` corresponds to the information we have already seen; the first `0` indicates that this is a non-recurrent edge and the second `0` indicates that this is the first input to the function node. The value `142` is the weight of the edge. As P-GP 2 does not have native support for floats, we take a simple encoding of a subset of rationals by storing our weights as integers and converting them to reals by dividing by 1000. So in this case, the value `142` corresponds to a weight of $\frac{142}{1000} = 0.142$.

Similarly, the single function node is labelled `"TANH":244`, where `"TANH"` refers to a tanh neuron whose behaviour follows

$$tanh\_neuron(x_1, \ldots, x_k) = \tanh\left(\Sigma_{i=1}^{k} x_i\right), \tag{4.11}$$

which is well defined for any number of inputs k. The second value, `244` is the bias of the function node. We use the same encoding, so the value `244` corresponds to a bias of $\frac{244}{1000} = 0.244$.

The single edge from the output node to the single function node has a weight of 1, meaning that the behaviour of the FG is determined by the single function node. Treating the function node's bias as one of the inputs to $tanh\_neuron$, we can see that the overall behaviour of the FG is given by

$$o_0 = \tanh\left(\frac{142}{1000} i_0 + \frac{-401}{1000} i_1 + \frac{244}{1000}\right), \tag{4.12}$$

such that when $i_0 = -1$ and $i_1 = 1$, $o_0 = -0.2904\ldots$. The introduction of weights and biases means that we can in general describe ANNs by treating neurons as function nodes with the summation component of their behaviour contained within their associated functions. By combining weights, biases and the recurrent edges we have seen in Section a 4.2.3, FGs can describe Recurrent Neural Networks (RNNs).

4.3 Semantics of Function Graphs

To describe the semantics of FGs more formally, we begin with a summary definition of FGs in Section 4.3.1. We then give the behaviour of FGs in Section 4.3.2.

4.3.1 Definition of Function Graphs

To summarise what we have observed from our examples, FGs are constructed of input nodes, function nodes and output nodes. Input nodes are labelled

```
"INPUT":x,
```
(4.13)

where `x` is an integer corresponding to a specific input of the problem. So, if a problem requires solutions to accept 3 inputs, there will be input nodes labelled `"INPUT":0`, `"INPUT":1` and `"INPUT":2` in corresponding FGs. Input nodes have no outgoing edges. In general, if some node v is an input node, we assume the use of a function $i(v)$ to recover the explicit input index implied by the integer `x` given in Equation 4.13.

Similarly, output nodes are labelled

```
"OUTPUT":x,
```
(4.14)

where `x` is an integer corresponding to a specific output of the problem. If a problem requires solutions to produce 2 outputs, there will be exactly output nodes labelled `"OUTPUT":0` and `"OUTPUT":1` in corresponding FGs. Output nodes have exactly 1 outgoing edge. If an FG's edges are explicitly ordered by labels, the single outgoing edge is given an index of 0. If an FG's edges are explicitly recurrent/non-recurrent then the single outgoing edge is non-recurrent. If an FG's edges are explicitly weighted, the single outgoing edge is given a weight that would imply an identity relation. For the simple neural network seen in Section 4.2.4, this weight was 1.0. In general, if some node v is an output node, we assume the use of a function $o(v)$ to recover the explicit output index implied by the integer `x` given in Equation 4.14.

Function nodes are explicitly associated with functions from a function set $F = \{f_1, f_2, \dots f_k\}$. In this thesis we assume that all functions $f \in F$ are multivariate functions over some domain $\mathbb{D}$, although we do discuss an extension to FGs to support multiple types in Section 4.4. If FGs are being learned over a function set F then we assume the use of an arity function

$$a : F \to \mathbb{N}_0, \tag{4.15}$$

which associates each function $f_i \in F$ with a non-negative integer describing the number of inputs that function expects. In the 1-bit adder seen in Section 4.2.1, all functions used 2 inputs, so for every function f_i used in that example, $a(f_i) = 2$. For the implementation of Fibonacci sequence seen in Section 4.2.3, the 2-input multiplication function $\times$ had $a(\times) = 2$ whereas the 1-input identity function id had $a(id) = 1$. We also assume the use of a naming function

$$p : F \to \Sigma^*, \tag{4.16}$$

where Σ^* corresponds to the set of strings available in GP 2. The naming function p associates each function with a unique name by which it can be referenced within an FG. Additionally, we assume that for any $f \in F$, $p(f) \neq$ `"INPUT"` and $p(f) \neq$ `"OUTPUT"` to avoid any confusion with input and output nodes. We also assume that naming is unique e.g. for any $f_1, f_2 \in F$ where $f_1 \neq f_2$, it holds that $p(f_1) \neq p(f_2)$. In the 1-bit adder seen in Section 4.2.1 we had that for the NAND function $\uparrow$, $p(\uparrow) =$ `"NAND"`. Each function node v is labelled

$$\mathtt{s}_v\mathtt{:x}_v, \tag{4.17}$$

where $\mathtt{s}_v$ is a string equal to some $p(f)$ with $f \in F$. The string $\mathtt{s}_v$ is then uniquely associating a function node with a function from the function set. The integer $\mathtt{x}_v$ is the bias associated with the node. If no such integer is present, the bias is assumed to be 1 and all functions are assumed to ignore the bias in their semantics. In general we recover the function associated with a function node v with the function

$$f(v) = p^{-1}(\mathtt{s}_v), \tag{4.18}$$

where $\mathtt{s}_v$ is the string component of the node v's label as given in Equation 4.17. Biases are given as integers in FGs but can be converted to any countable domain $\mathbb{D}$ with a function $w\_rel : \mathbb{Z} \to \mathbb{D}$. In the simple neural network shown in Section 4.2.4, integers were converted to (a rational subset of) reals $\mathbb{R}$ with the function

$$w\_rel(x) = \frac{x}{1000}. \tag{4.19}$$

In general we recover the bias associated with a function node v with the function

$$b(v) = w\_rel(\mathtt{x}_v), \tag{4.20}$$

where $\mathtt{x}_v$ is the integer component of the node v's label as given in Equation 4.17. A function node must have exactly as many outgoing edges as their associated functions expect e.g. a function node v should have $a(f(v))$ edges. Each edge e is labelled

$$\mathtt{w}_e\mathtt{:r}_e\mathtt{:o}_e, \tag{4.21}$$

where $\mathtt{w}_e \in \mathbb{Z}$, $\mathtt{r}_e \in \{\mathtt{0},\mathtt{1}\}$ and $\mathtt{o}_e \in \mathbb{N}_0$. The integer $\mathtt{w}_e$ is the integer weight associated with an edge. Using the same w_rel function as used with biases, $\mathtt{w}_e$ may be converted to any countable domain. In general we recover the weight associated with an edge e with the function

$$w(e) = w\_rel(\mathtt{w}_e), \tag{4.22}$$

where $\mathtt{w}_e$ is the first integer component of the edge e's label as given in Equation 4.21. If the length of an edge e's label is $2, 1$ or 0, we assume that $\mathtt{w}_e = 1$. The integer $\mathtt{r}_e$ determines whether an edge is *recurrent* or *non-recurrent*. In general we recover whether a weight is recurrent with the function

$$r(e) = \begin{cases} True, & \mathtt{r}_e = 1; \\ False, & \mathtt{r}_e = 0, \end{cases} \tag{4.23}$$

where $\mathtt{r}_e$ is the second integer component of the edge e's label as given in Equation 4.21. If the length of an edge e's label is 2 then $\mathtt{r}_e$ is assumed to be the first integer component of the label. If the length of an edge e's label is 1 or 0 then $\mathtt{r}_e$ is assumed to be 0, e.g. e is non-recurrent. The integer $\mathtt{o}_e$ assigns to each edge a position in a total ordering. If a function node v has $k = a(f(v))$ edges, $e_0, e_1, \ldots, e_{k-1}$, with values, $o_{e_0} = 0, o_{e_1} = 1, \ldots, o_{e_{k-1}} = k-1$. If the length of an edge e's label is 2 or 1 then $\mathtt{o}_e$ is assumed to be the last integer component of the label. If the length of an edge e's label is 0 then we assume that the corresponding function's node associated function is commutative and that $\mathtt{o}_e$ is assigned arbitrarily. In general, the ordering index of an edge is recovered with the function

$$ord(e) = \mathtt{o}_e. \tag{4.24}$$

We give a table summarising the functions we have introduced in Table 4.1, in the context of an FG, $G = (V, E, s, t, l_V, l_E)$, function set F and general function domain $\mathbb{D}$.

Symbol	Trace	Description
i	$i : V \to \mathbb{N}_0$	Returns the index associated with an input node.
o	$o : V \to \mathbb{N}_0$	Returns the index associated with an output node.
a	$a : F \to \mathbb{N}_0$	Returns the arity of a given function.
p	$p : F \to \Sigma^*$	Returns the unique name of a given function.
o	$o : V \to \mathbb{N}_0$	Returns the index associated with an output node.
f	$f : V \to F$	Returns the function associated with a function node.
w_rel	$w\_rel : \mathbb{Z} \to \mathbb{D}$	Translates integer weights and biases to domain $\mathbb{D}$.
b	$b : V \to \mathbb{D}$	Returns the bias associated with a function node, translated to domain $\mathbb{D}$.
w	$w : E \to \mathbb{D}$	Returns the weight associated with an edge.
r	$r : E \to \{True, False\}$	Returns whether an edge is recurrent.
ord	$ord : E \to \mathbb{N}_0$	Returns the ordering index of an edge.

Table 4.1: Functions introduced in the description of FGs. These functions are given in the context of an FG $G = (V, E, s, t, l_V, l_E)$, function set F and general function domain $\mathbb{D}$.

4.3.2 Behaviour of Function Graphs

FGs effectively operate as data flow diagrams with memory. We assume that for a given FG, $G = (V, E, s, t, l_V, l_E)$, at the previous time-step, $n-1$, there exists a previous state, $s_v(n-1)$, for all nodes $v \in V$. In this section we describe the behaviour of the m-input, n-output FG at time n to update to state $s_v(n)$ for all such nodes, when driving the FG with inputs, $I_0(n), \ldots, I_m(n)$ and computing outputs, $O_0(n), \ldots, O_n(n)$. To do this we firstly define the value associated with each edge at time n, given by

$$val_e(n) = \begin{cases} w(e).s_{t(e)}(n-1) & r(e) \text{ is } True; \\ w(e).s_{t(e)}(n) & r(e) \text{ is } False, \end{cases} \tag{4.25}$$

i.e., multiply the weight $w(e)$ by the previous state of e's target if e is recurrent, or multiply $w(e)$ by the current state of e's target if e is non-recurrent. We assume that inputs are 'loaded in', that is, for each input node $v \in V$, $s_v(n) = I_{i(v)}(n)$.

To compute the updated state $s_v(n)$ of function node v, we assume that edges, $e_0, \ldots, e_{k-1}$, are v's $k = a(f(v))$ outgoing edges, ordered in ascending order according to ord, i.e., $\forall e_i, e_j, i < j \Rightarrow ord(e_i) < ord(e_j)$. Then we can compute the update as

$$s_v(n) = f(v)(val_{e_0}(n), \ldots val_{e_{k-1}}(n), b(v)), \tag{4.26}$$

that is, apply the function node's function to the (ordered) values associated with each edge at time y and the bias associated with the function node.

For any output node v, the update is simply $s_v(n) = val_{e_0}(n)$ for the output's single outgoing edge e_0. Overall, this gives way to the update equation

$$s_v(n) = \begin{cases} I_{i(v)} & \text{v is an input node;} \\ f(v)(val_{e_0}(n), \ldots, val_{e_{k-1}}(n), b(v)) & \text{v is a function node;} \\ val_{e_0}(n) & \text{v is an output node.} \end{cases} \tag{4.27}$$

Once the state of the entire FG has been updated, we can safely return outputs with

$$O_i(n) = s_v(n) \text{ where } v \text{ is an output node and } o(v) = i. \tag{4.28}$$

Note that the formulae in Equations 4.25 and 4.27 are effectively giving a recursive definition of the update to an FG, with input nodes and values associated with recurrent edges serving as base cases. It is worth noting, then, that this definition is cyclic whenever there

Node Label	Argument Types	Returns	Description
`"INPUT":0`	-	*matrix*	The first input, a 2D matrix
`"INPUT":1`	-	*float*	The second input, a float
`"ADD_M"`	x_0:*matrix*, x_1:*matrix*	*matrix*	Matrix Addition returning $x_0 + x_1$.
`"ROT_M"`	x_0:*matrix*, x_1:*float*	*matrix*	Matrix Rotation, returning the rotation of x_0 around the x-axis by x_1 radians.
`"ADD_F"`	x_0:*float*, x_1:*float*	*float*	Float Addition returning $x_0 + x_1$.
`"OUTPUT":0`	x_0:*matrix*	*matrix*	The first (and only) output, returning 2D matrix x_0

Table 4.2: A simple set of typed inputs, functions and outputs.

exists a cycle of non-recurrent edges. For this reason, FGs *must* be constrained so that the subgraph induced by their non-recurrent edges is acyclic. Additionally, we assume that the initial state $s_v(0) = 0$ for all nodes $v \in V$, although in principle different initial states could be used.

As a final note, a reader interested in practical implementation may benefit from the knowledge that FGs can be evaluated in linear time (with respect to the size of the graph) by performing a topological sort on the non-recurrent subgraph of the FG and then evaluating each node in the sequence they appear in the topological sort, storing results for later reuse. This assumes that the cost of evaluating individual functions is constant, which may not be the case in practice. Evaluation can be further optimised by only evaluating active nodes, that is, those for which there is a path to from an output node.

4.4 Conclusions and Future Work

In this chapter we have introduced FGs as a generic model of graph-like programs capable of expressing digital programs, functional programs, stateful programs and ANNs, forming the domain for our evolutionary experiments in the coming chapters. Further, they generalise the phenotypic representations used in a number of existing evolutionary paradigms, e.g.:

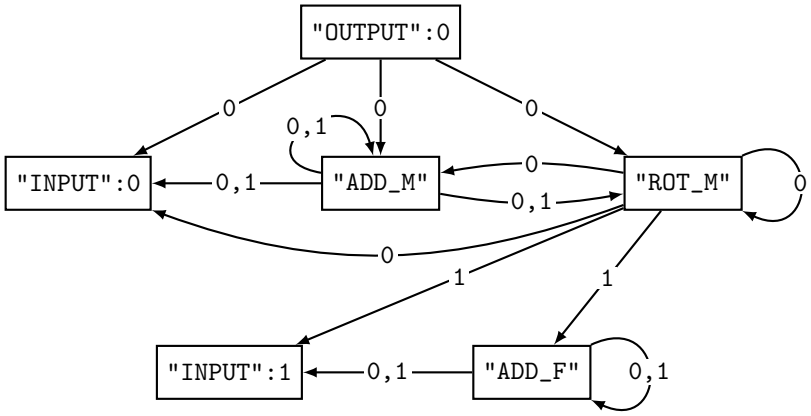


Figure 4.6: A Type Graph representing the inputs, functions, outputs and valid interconnections described in Table 4.2. Where an edge is shown with a pair of labels, e.g. `0,1`, this is shorthand for 2 parallel edges, 1 for each label.

1. Tree-based GP [129], where individuals are FGs with the restriction that each function node has exactly one incoming edge.
2. Cartesian GP [155], where individuals can be directly translated to FGs once genotypic material (such as the ordering on nodes) has been stripped out.
3. Neuroevolution techniques such as NEAT [222], once genotypic material (such as historical markers) has been stripped out.

Hence investigating the evolution of FGs directly has some additional comparative value, in clarifying the costs and benefits of genotypic design decisions made in different paradigms.

There are a number of areas in which FGs could be expanded, and we discuss 2 particular directions here. Firstly, the extension of FGs to Typed Function Graphs (TFGs) would allow the evolution of typed programs. Strongly Typed GP [160] and other GP approaches [175,212] are capable of handing typed functions and typed data. Further, there are a number of general program synthesis problems [99] where an evolutionary system must be able to handle multiple types to effectively produce a solution. It is therefore clear that the extension of FGs to TFGs would enable more general applicability of the ideas we explore later in this thesis. While the exact mechanism of extension to TFGs remains to be explored, we suggest that the approach of Strongly Typed GP, where genetic operators are designed under consideration of the underlying type system, offers a promising direction of thought. To this end, it is then necessary for the genotype (e.g. FG) being evolved to contain some information about the

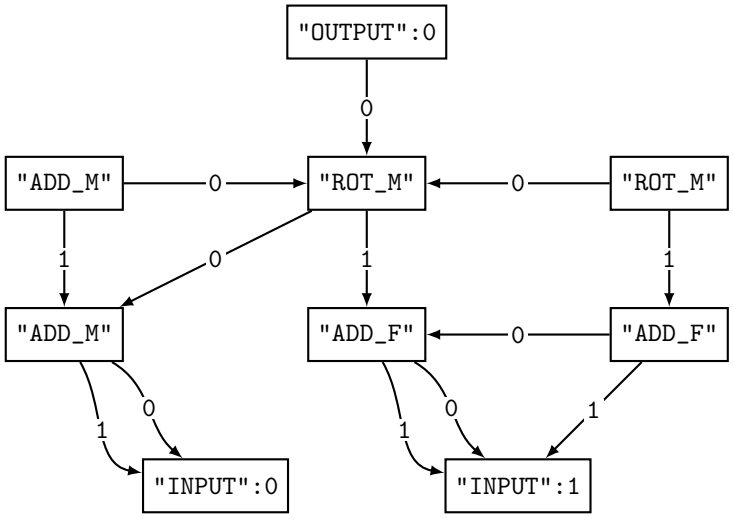


Figure 4.7: The FG shown here is a typed FG and an instance of the Type Graph shown in Figure 4.6. This is true as there exists a non-injective morphism from the TFG shown and the Type Graph.

type system.

Type Graphs (TGs) (and by extension, Typed Graph Transformation, see [96]) may provide a groundwork by which this might be achieved. Here, a TG, TG, represents the underlying concepts of a class of graphs, and a graph, G, is an instance of TG if there exists a (potentially non-injective) graph morphism $f : G \to TG$. As an example, consider the constraints on an FG which takes as inputs a 2D-matrix (`"INPUT":0`) and a float (`"INPUT":1`) and returns a 2D-matrix (`"OUTPUT":0`). The FG may be constructed from matrix addition (`"ADD_M"`), taking 2 matrices as inputs and returning a matrix, matrix rotation around the x-axis (`"ROT_M"`), taking as input a matrix and a float, and float addition (`"ADD_F"`), taking 2 floats as inputs. These various functions are listed in Table 4.2. It is possible to convert this to a TG as shown in Figure 4.6. The Typed FG in Figure 4.7 is an instance of the TG in Figure 4.6 as there exists a (non-injective) morphism from the TFG to the TG.

Secondly, we suggest that investigation into Hierarchical FGs (HFGs) would open up a wealth of new research directions. The evolvability of structural modularity and code reuse has been thoroughly explored through Automatically Defined Functions (ADFs) [129], and built upon in techniques such as Embedded Cartesian Genetic Programming (ECGP) [253] and tag-based modules in PushGP [215]. However, we believe that existing results from graph transformation may offer additional insight; we suggest the extension of FGs to HFGs based

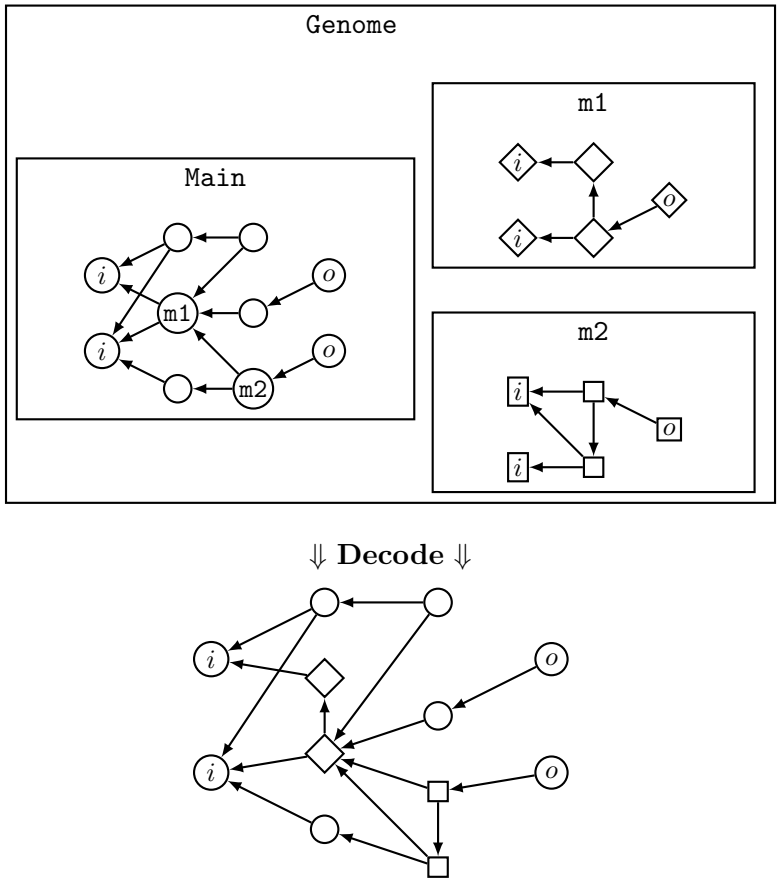


Figure 4.8: A 'flat' HFG where the graph contains a `Main` graph representing the structure of the individual, and modules `m1` and `m2` operating as learnt sub-structures. The HFG can then be translated to a conventional FG by a decoding process as shown.

on the pre-existing notion of Hierarchical Graphs set out in various forms in [31,56,176]. The common concept among these works is that graph components may contain other graphs in an arbitrarily-deep nested structure. Perhaps the most intuitive notion is set out in [56], where (hyper) edges in a hierarchical (hyper) graph may be associated with other hierarchical graphs via a containment function. The view set out in ADFs could then be represented by a simple hierarchical structure where a graph is decomposed into a 'main' subgraph and 'module' subgraphs as shown in Figure 4.8. A decoding process embeds the module subgraphs into the main subgraph to produce a conventional FG. Perhaps more interesting is the notion of abstraction embedded within a single structure, as in ECGP, with function nodes containing modular subgraphs which can be created, deleted and copied. We could then represent the

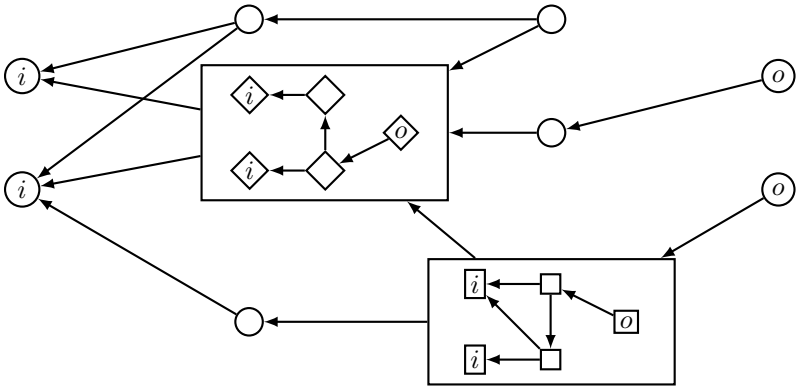


Figure 4.9: An embedded HFG. Modules are contained within nodes. These could then be modified, copied and deleted with hierarchical graph transformations [56].

HFG in Figure 4.8 with the embedded structure in Figure 4.9. It is clear that more research is required to establish the more practical representation of these options.

5 Evolving Graphs by Graph Programming

Abstract

In this chapter, we describe the first Evolutionary Algorithm (EA) based on genetic operators implemented as graph programs. The algorithm, termed Evolving Graphs by Graph Programming (EGGP), evolves Acyclic Function Graphs (AFGs). We give an initialisation procedure capable of generating such AFGs. We also give edge and node mutation operators which respect the constraints of such AFGs. We describe a typical experimental configuration for EGGP and compare the theoretical landscape available to our approach with that of Cartesian Genetic Programming (CGP).

Relevant Publications

Content from the following publications is used in this chapter:

[8] T. Atkinson, D. Plump, and S. Stepney, "Evolving graphs by graph programming," in *Proc. European Conference on Genetic Programming, EuroGP 2018,* ser. LNCS, vol. 10781. Springer, 2018, pp.35–51.

[9] T. Atkinson, D. Plump, and S. Stepney, "Probabilistic graph programs for randomised and evolutionary algorithms," in *Proc. International Conference on Graph Transformation, ICGT 2018*, ser. LNCS, vol. 10887. Springer, 2018, pp. 63–78.

5.1 Introduction

Now equipped with a well-defined notion of Function Graphs (FGs) we can look towards the evolution of programs at the level of graphs, rather than through some encoding. Free of the constraints of a specific genetic representation and armed with the powerful graph programming language P-GP 2, we examine new ideas in later chapters such as the exploitation of domain knowledge to induce Semantic Neutral Drift (SND), or the invention of new crossover operator that is entirely non-disruptive. The first step taken in this body of work is the setting out of a simple, minimal system that is capable of competing with modern state of the art techniques. We use this system as a first building block with which we can move towards more complex and advanced concepts. In this chapter we explore the new paradigm Evolving Graphs by Graph Programming (EGGP) which learns Acyclic Function Graphs (AFGs) with genetic operators defined as P-GP 2 programs.

Our algorithm consists of 4 core components:

1. A P-GP 2 program that generates AFGs to be used as an initialisation procedure.
2. The $1 + \lambda$ Evolutionary Algorithm (EA), using neither large populations nor crossover.
3. An atomic edge mutation that modifies a single edge of an AFG.
4. An atomic node mutation that modifies a single node of an AFG.

The evolutionary process induced by these components is remarkably effective at solving benchmark problems drawn from the literature, as shown experimentally in Chapter 6. In contrast to existing work on Cartesian Genetic Programming (CGP) [157] or Parallel Distributed Genetic Programming (PDGP) [188], we do not require a notion of a Cartesian grid to achieve the preservation of acyclicity. Instead, our sequentially applied rule-sets will induce landscapes which correctly identify viable mutations which preserve acyclicity. As we will see later in this chapter, this concept gives way to a generalisation of the landscapes induced in CGP.

This chapter is arranged as follows. In Section 5.2 we discuss the initialisation program used to instantiate our populations. Section 5.3 covers atomic edge mutation and node mutation programs used to modify our populations. In Section 5.4 we give a concise description of the $1 + \lambda$ EA alongside justification for its use. Section 5.5 gives an example of a very simple evolution run for learning an XOR gate. We relate our approach to other approaches in Section 5.6 giving particular comparison with CGP. Finally, in Section 5.7 we conclude the findings of this chapter and set out areas for future work.

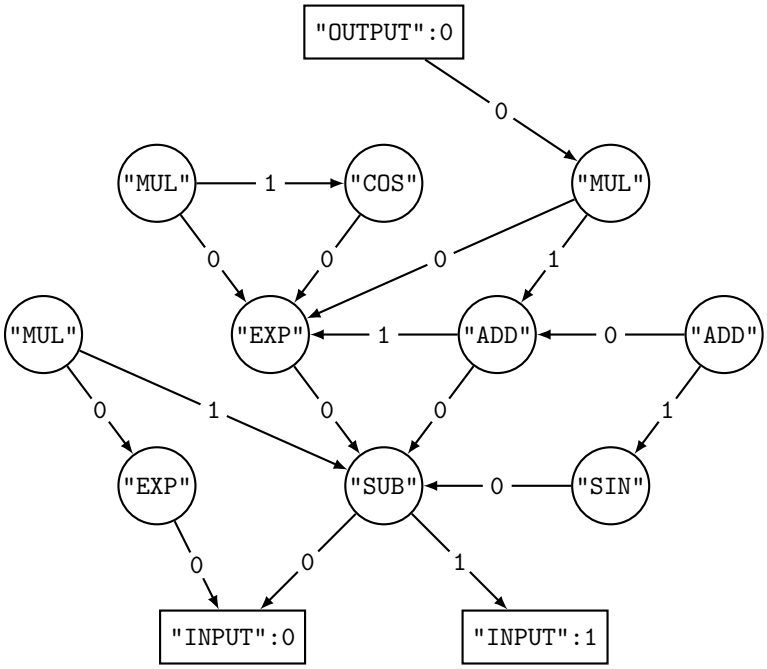


Figure 5.1: A simple AFG that has weights and biases that are effectively ignored. The information with respect to edge recurrence, edge weight and node bias are therefore not shown.

5.2 Initialisation

The first step of any EA is to generate individuals to form the initial population. In this section we set out a simple initialisation procedure given as a pair `Init`, S where `Init` is a P-GP 2 program which, when applied to initial graph S, generates AFGs suitable to the target problem.

In this chapter we deal with a simplified set of Acyclic FGs, which we refer to as AFGs. The AFGs handled here have the following properties:

1. They are acyclic, and do not feature recurrent edges.
2. All edge weights and function node biases are assumed to be equal to 1 and are effectively ignored during execution.
3. All functions are assumed to ignore the values coming in from the bias of each function node.

An example of such an AFG is shown in Figure 5.1. As we are not using the recurrence

```
Main := ([add_node_f1, ..., add_node_fk]; [connect_node]!; unmark_node!)!;
                        [connect_output]!; remove_counter
```

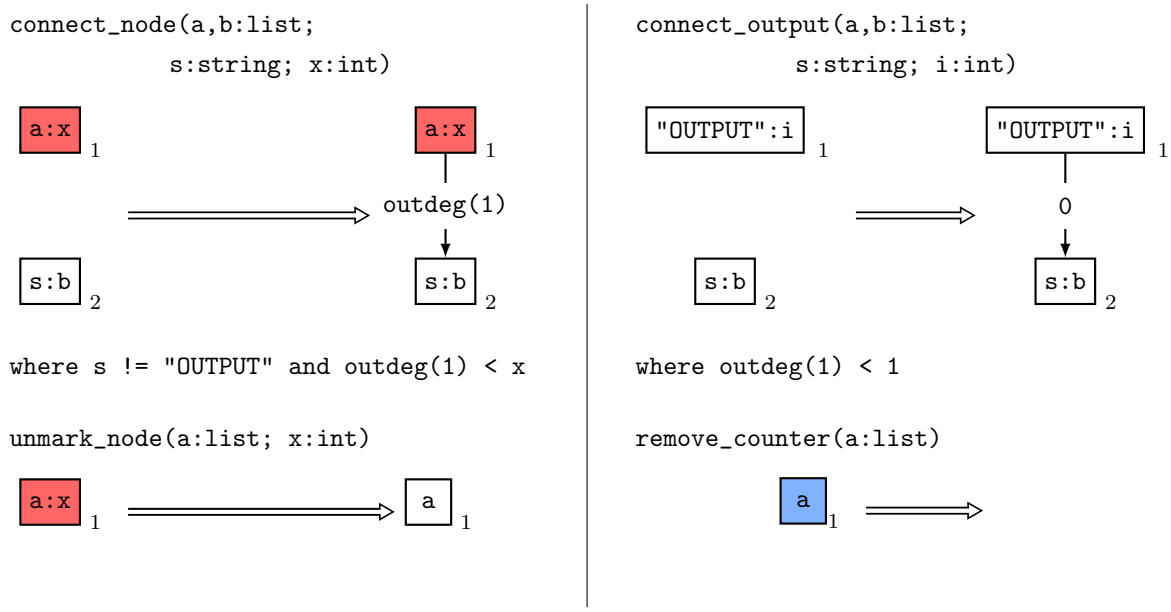


Figure 5.2: A program for generating our simplified AFGs. For each function f_x in our function set $F = \{f_1, \ldots, f_k\}$, we have a rule `add_node_f`x, as visualised in Figure 5.3.

property, edge weights or node biases, this information is not shown. The integers shown on each edge give the index of that edge, describing the order in which edges' incoming values should be presented to their associated function node.

Figure 5.2 shows our initialisation program for AFGs. This program consists of 3 sequential commands, one of which consists of 3 further sequential commands. An overview of the program is:

1. The main loop. This is called as long as possible. It consists of 3 sequential commands:
   a) `[add_node_f1, ..., add_node_fk]`. For each function f_x in function set $F = \{f_1, \ldots, f_k\}$, we have a unique rule `add_node_f`x in our probabilistically called rule-set which adds a node corresponding to that function. The newly added node is marked `red` and is labelled with both the string representation of a given function f_x, and the arity of that function. The result of this command is that

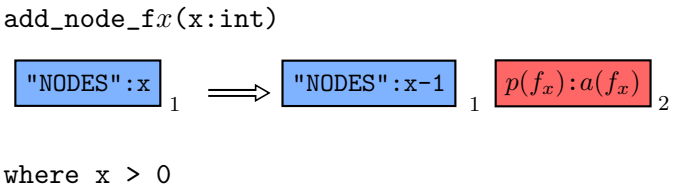


Figure 5.3: When we construct an initialisation program such as the one shown in Figure 5.2, we generate a rule `add_node_f`x for each function $f_x \in F$ where F is the function set, a is the arity function and p is the naming function. The added function node, marked `red`, is labelled with a list consisting of the function f_x's associated unique name $p(f_x)$ followed by its arity $a(x)$. At the same time, the `blue` marked node counter is decremented. The rule cannot be called once the counter is 0.

we add a node of some randomly chosen function from F until we have added as many function nodes as specified by the input graph. At the same time, our node counter, marked `blue` is decremented, ensuring that this rule is only called as many times as indicated by the input graph. The general model of the rules used here is shown in Figure 5.3.

b) `[connect_node]!`. This node connects our newly added `red` marked node to randomly chosen (non-output) nodes. This is done until the outdegree of the added node matches the arity of its function. Every time such an edge is added, its label is set to the current outdegree of the node, ensuring that we have a proper ordering on the node's outgoing edges.

c) `unmark_node`. This rule is called once after edges have been added to ensure the newly added node's outdegree matches its function's arity. It simply unmarks the added node and removes its arity indicator, leaving it as a regular node ready to be connected to by other added nodes and outputs.

2. `[connect_output]!`. This rule connects each output to some randomly chosen (non-output) node in the graph. It is called as long as possible, and requires that the matched output node's outdegree is 0, ensuring that each output node gains 1 new outgoing edge.

3. `remove_counter`. This rule removes our unique `blue` marked node counter indicating the number of nodes to be added. This ensures that this node does not persist beyond the initialisation program.

Overall, the program's semantics can be seen to add randomly chosen function nodes until

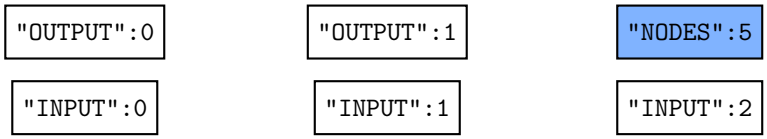


Figure 5.4: An input graph for the initialisation program in Figure 5.2. This input graph can be used to generate AFGs with 3 inputs, 2 outputs and 5 function nodes.

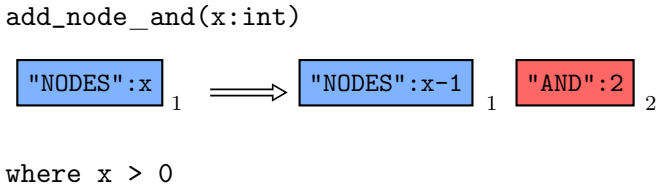


Figure 5.5: The instantiation of the generic rule shown in Figure 5.3 with $f_x = \wedge =$ `and`, $a(f_x) = 2$ and $p(f_x) =$ `"AND"`.

the number of function nodes equals the amount specified by the input graph. Each function node is randomly connected to previously added function nodes and input nodes. Then, output nodes are connected at random to the rest of the graph. Finally, the node specifying the number of function nodes to add is removed. We therefore expect the input graph to this program to consist of the following:

1. For each input associated with the problem, there exists an input node.
2. For each output associated with the problem, there exists an output node.
3. A node, marked `blue` and labelled `"NODES":x` where `x` $\in \mathbb{N}_0$ specifies the number of function nodes to add.

An example of such an input graph is given in Figure 5.4. When applying an initialisation program to this graph, we expect to produce graphs with 3 inputs, 2 outputs and 5 function nodes. As a visual example we will consider this input graph alongside the function set F with arity function a and naming function p considering the function set of AND, OR, NAND and NOR logic gates, each with arity 2. More formally, we have

$$F = \{\wedge, \vee, \uparrow, \downarrow\}, \tag{5.1}$$

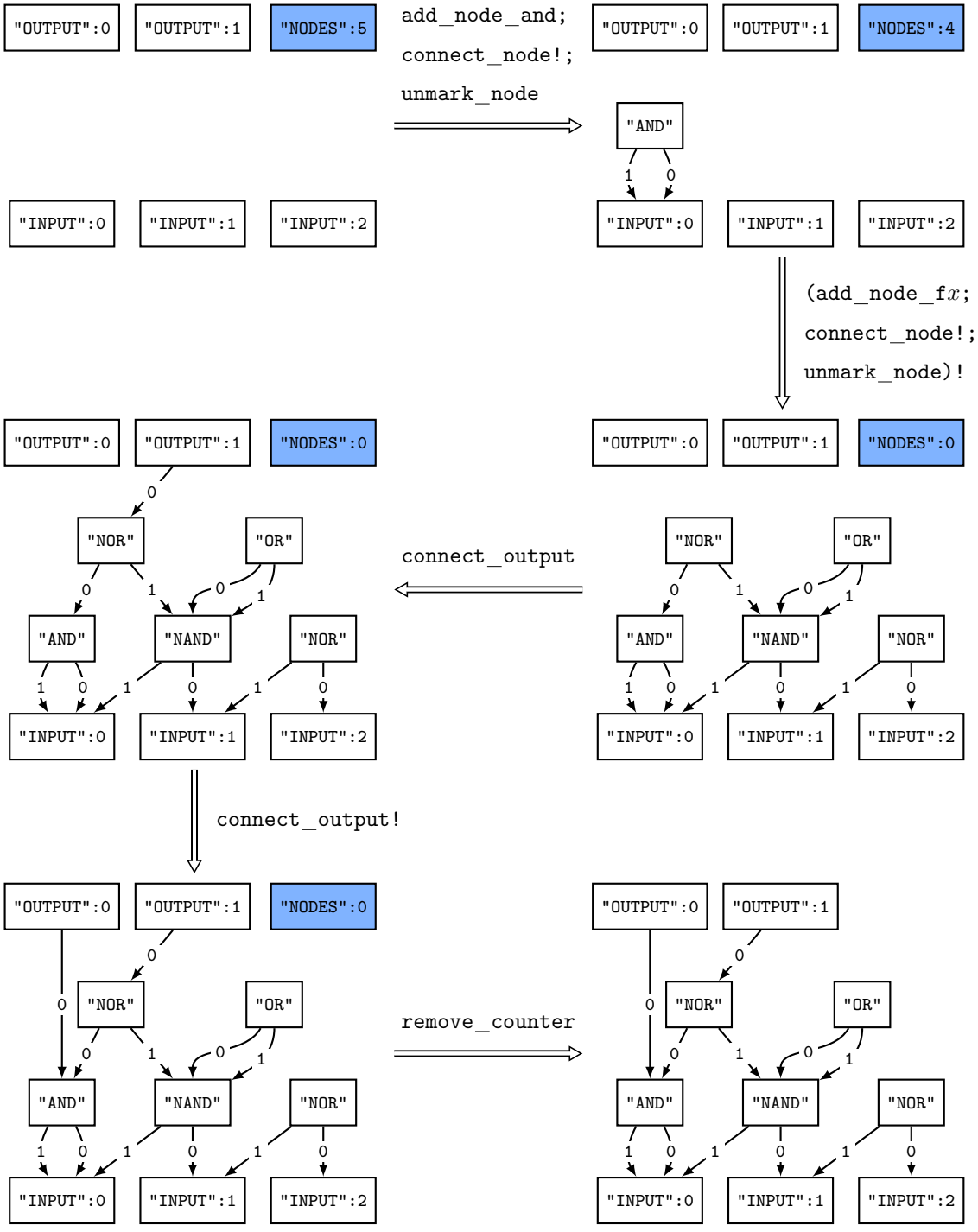


Figure 5.6: A trace of the application of the initialisation program in Figure 5.2 with function set {AND, OR, NAND, NOR } applied to the input graph in Figure 5.4.

with arity function a and naming function p given by

$$a = \begin{cases} \wedge \to 2; \\ \vee \to 2; \\ \uparrow \to 2; \\ \downarrow \to 2, \end{cases} \quad \text{and} \quad p = \begin{cases} \wedge \to \texttt{"AND"}; \\ \vee \to \texttt{"OR"}; \\ \uparrow \to \texttt{"NAND"}; \\ \downarrow \to \texttt{"NOR"}, \end{cases} \tag{5.2}$$

respectively.

As a clarifying visual, we give the instantiation of the generic rule shown in Figure 5.3 with $f_x = \wedge = \texttt{and}$, shown in Figure 5.5.

We provide a trace of the initialisation program alongside the function set F applied to the graph from Figure 5.4 in Figure 5.6. In the first transformation, an AND gate is added. In the next step, we show the result of running the main loop of the initialisation program until completion; 4 more function nodes are added. The third step shows the application of a single call to `connect_output`. The next step shows the result of applying `connect_output` as long as possible. This results in an AFG with a now 'junk' `blue` node counter. The final step shows the application of `remove_counter` deleting this node, resulting in the final output AFG.

While not presented here, a proof can be given that our initialisation program is complete. That is, if the initialisation program is given for function set F and presented with an appropriate input graph with i inputs, o outputs and a node counter for n function nodes, then any AFG describable over i, o, n and F can be generated. However, it is not clear whether all individual AFGs are generated with equal probability.

5.3 Mutation

In this section we detail mutation in EGGP. EGGP provides two forms of atomic mutation:

1. Edge mutation. Here a single edge is chosen at random and redirected while preserving acyclicity. This mutation is explained and detailed in Section 5.3.1.
2. Node mutation. Here a single node is chosen at random and relabelled to be associated with some different function from the function set. Then edges are added or removed to respect the new function's arity, and finally shuffled to avoid biasing towards certain

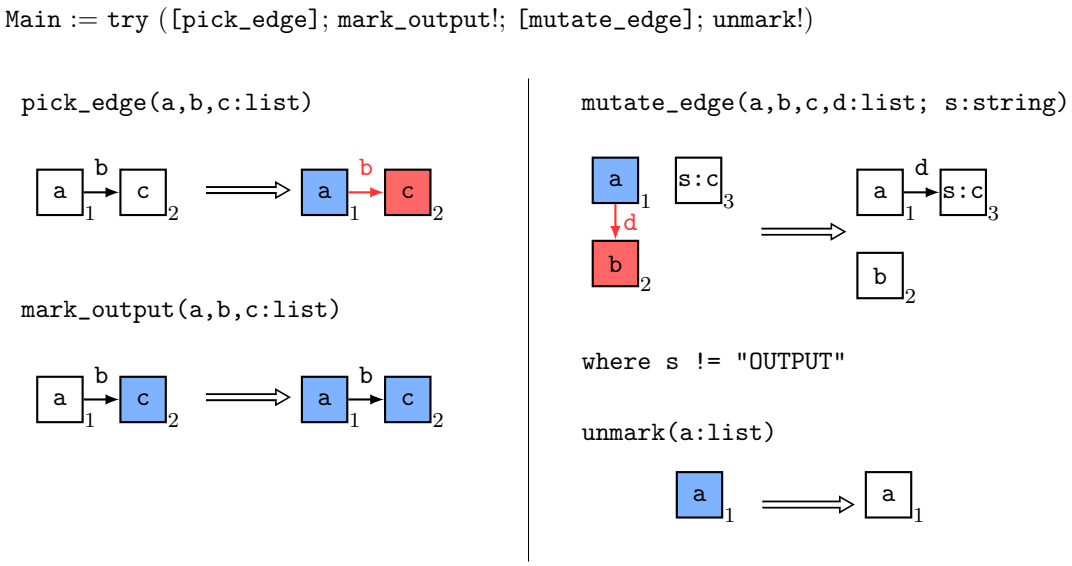


Figure 5.7: A program for mutating AFGs' edges while preserving acyclicity.

landscapes when using non-commutative functions. This mutation is explained and detailed in Section 5.3.2.

Additionally, to provide larger jumps in the landscape via multiple applications of our atomic mutations, we use the Binomial mutation. This is described in Section 5.3.3.

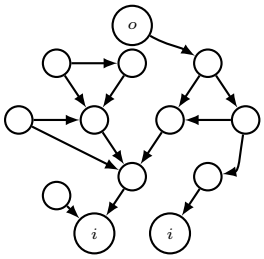
5.3.1 Edge Mutation

Edge mutation in EGGP consists of 4 steps:

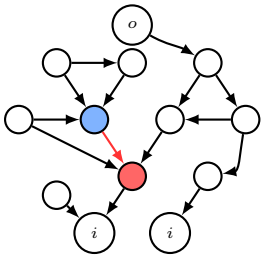
1. Pick an edge to redirect uniformly at random.
2. Identify all nodes for which there is a path from that node to the source of the chosen edge. If the edge were redirected to target these nodes, then a cycle would be created.
3. Redirect the chosen edge to target some node for which there is no such path.
4. Remove any annotations made in the graph by step 2.

We present a P-GP 2 program implementing this mutation in Figure 5.7. It should be stressed that in general, such a program works on the assumption that the host graph is unmarked. Each of the commands called sequentially corresponds to a step of the process outlined above:

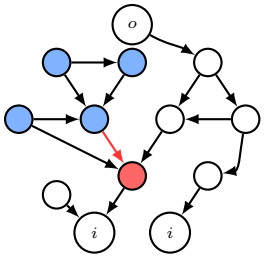
This individual is to undergo an edge mutation preserving acyclicity.



(1) `pick_edge`:
An edge to mutate is chosen at random and marked (red) alongside its source node s (blue) and target node t (red).



(2) `mark_output!`:
Invalid candidate nodes for redirection are identified. If a node v has a directed path to s it is marked blue, as targeting it would introduce a cycle.



(3) `mutate_edge; unmark!`:
The edge e is mutated to target some randomly chosen unmarked (non-output) node, preserving acyclicity. The new target has been marked with a star '$\star$' for visual clarity. Finally, all marks are removed.

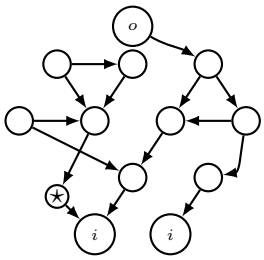


Figure 5.8: A trace of the application of the edge mutation program in Figure 5.7. For visual simplicity, node and edge labels have been omitted.

1. The rule `pick_edge` chooses an edge uniformly at random. Its source is marked `blue` and its target is marked `red` to uniquely identify them. Additionally, the edge itself is marked `red` to avoid any confusion with parallel edges.
2. The rule `mark_output` called as long as possible ensures all nodes with paths to the source of our chosen edge are marked `blue`. As a consequence, redirecting the edge to target and `blue` node would introduce a cycle, whereas redirecting the edge to target any unmarked node would not.
3. The rule `mutate_edge` redirects the edge to target some unmarked node. Not only does this avoid introducing a cycle, but the fact that the target of the chosen edge is `red` ensures that the mutation always produces a change.
4. The rule `unmark` called as long as possible removes the `blue` marks created by step 2.

We give an example execution of our mutation in Figure 5.8. In this diagram, all node and edge labels are not shown to aid visual clarity and to stress that this mutation depends only on the topology of the graph and effectively ignores labels.

Correctness

Here we present an outline of a proof that the edge mutation is correct in the sense that, when presented with an unmarked AFG as an input graph, the edge mutation can only produce AFGs as output graphs.

The overall correctness is a simple argument; if there exists a path $v_1 \to v_2$, then creating an edge, $v_2 \to v_1$, clearly creates a cycle, $v_1 \to v_2 \to v_1$. In contrast, if there is no such path, $v_1 \to v_2$, then it is clear that creating an edge, $v_2 \to v_1$, cannot create such a cycle. Hence the correctness of our program depends on the correctness of the claim in step 2, that the application of `mark_output` as long as possible causes all nodes for which there is such a path to become marked `blue`. To see that the claim in step 2 is true, we can use a simple proof by induction on the length of paths to the source of the chosen edge:

Lemma 1 (Correctness of Edge Mutation.)**.** *Let G be an unmarked AFG, and $G \Rightarrow_{pick\_edge} H$ be a valid derivation and e be the single `red` marked edge in G. Then for any $D = (V, E, l_V, l_E, s, t)$ with $H \Rightarrow_{mark\_output!} D$, it holds that*

$$\text{For all } v \in V, \text{ if a path } v \to s(e) \text{ exists then } v \text{ is } \texttt{blue}. \tag{5.3}$$

Proof of Lemma 1. **Base case $n = 1$:**

$$\text{For all } v \in V, \text{ if an edge } v \to s(e) \text{ exists then } v \text{ is } \texttt{blue}. \tag{5.4}$$

where an edge $v \to s(e)$ is equivalent to a path $v \to s(e)$ of length 1.

Now suppose this base case did not hold; then there would be some node, $v_x \in V$, for which there existed an edge, $v_x \to s(e)$, and v_x is not marked `blue`. As G is acyclic and no edges have been added in $G \Rightarrow H \Rightarrow^* D$, it is clear that $v_x \neq t(e)$ as this would imply a cycle. As G is unmarked, and only 1 edge has been marked by the single call to `pick_edge`, it follows that the edge, $v_x \to s(e)$, must be unmarked. Further, as v_x is not `blue` and $v_x \neq t(e)$, v_x must be unmarked as G is unmarked and no other marks are introduced by `pick_edge` or `mark_output`.

As no `blue` marked nodes are unmarked or remarked by `mark_output` it must hold that:

1. $s(e)$ is marked `blue`.
2. The edge $v_x \to s(e)$ must be unmarked.
3. v_x must be unmarked.

Then it is clear that there exists a match for `mark_output` with node 2 matched to $s(e)$ and node 1 matched to v_x. Hence we have a contradiction with $H \Rightarrow_{\texttt{mark\_output!}} D$ and it follows that v_x cannot exist.

Inductive Hypothesis n = k: Assume that for $n = k$:

$$\text{For all } v \in V, \text{ if a path } v \to s(e) \text{ of length } k \text{ exists then } v \text{ is } \texttt{blue}. \tag{5.5}$$

Inductive step n = k + 1: Consider a node v_x such that

$$\text{There exists a path of length } k+1, \ v_x \to s(e), \text{ and } v_x \text{ is not marked } \texttt{blue}. \tag{5.6}$$

As G is acyclic and no edges have been added in $G \Rightarrow H \Rightarrow^* D$, it is clear that $v_x \neq t(e)$ as this would imply a cycle. As G is unmarked, and only 1 edge has been marked by the single call to `pick_edge`, it follows that all edges in the path edge, $v_x \to s(e)$, must be unmarked. Further, as v_x is not `blue` and $v_x \neq t(e)$, v_x must be unmarked as G is unmarked and no other marks are introduced by `pick_edge` or `mark_output`.

For such a path of length $k + 1$ to exist it is clear that there must exist some v_y where there is an edge, $v_x \to v_y$, and a path of length $k : v_y \to s(e)$. By our inductive hypothesis, v_y must be marked `blue`. Therefore we have that:

1. v_y is marked `blue`.
2. The edge $v_x \to v_y$ must be unmarked.
3. v_x must be unmarked.

Once again it is clear that there exists a match for `mark_output` with node 2 matched to v_y and node 1 matched to v_x. Hence we have a contradiction with $H \Rightarrow_{\texttt{mark\_output!}} D$ and it follows that v_x cannot exist.

Hence for all $n \geq 1$, it holds that

$$\text{For all } v \in V, \text{ If a path } v \to s(e) \text{ of length } n \text{ exists then } v \text{ is } \texttt{blue}. \tag{5.7}$$

□

Hence it is clear that our edge mutation preserves acyclicity. Further, the only relabelling of nodes to take place is in the marks, and all marks are removed, it is clear that all node labels are unchanged by the edge mutation. Additionally, the only modified edge is that chosen by the rule `pick_edge`, and this is simply marked, unmarked and redirected. In combination, these facts guarantee that, when presented with an unmarked AFG, the edge mutation always produces an AFG e.g. is correct with respect to the domain we are interested in.

5.3.2 Node Mutation

Node mutation in EGGP consists of 4 steps:

1. Pick a function node to mutate uniformly at random.
2. Mutate the function node, associating it with some new function from the function set.
3. Correct the arity of the function node. Either:
   a) If the outdegree of the function node is less than the new function's arity, identify all nodes for which there is a path from that node to the mutated node. Add new edges to nodes for which there is no such path until the outdegree matches the new function's arity.
   b) If the outdegree of the function node is greater than the new function's arity, delete edges until the outdegree matches the new function's arity.
4. Shuffle the function node's outgoing edges with respect to their ordering indices.

```
Main := [pick_node]; [mutate_node_f1, ..., mutate_node_fk]; mark_output!; [add_edge]!;
        [delete_edge]!; store_edge!; [order_edge]!; clean_node; unmark_node!
```

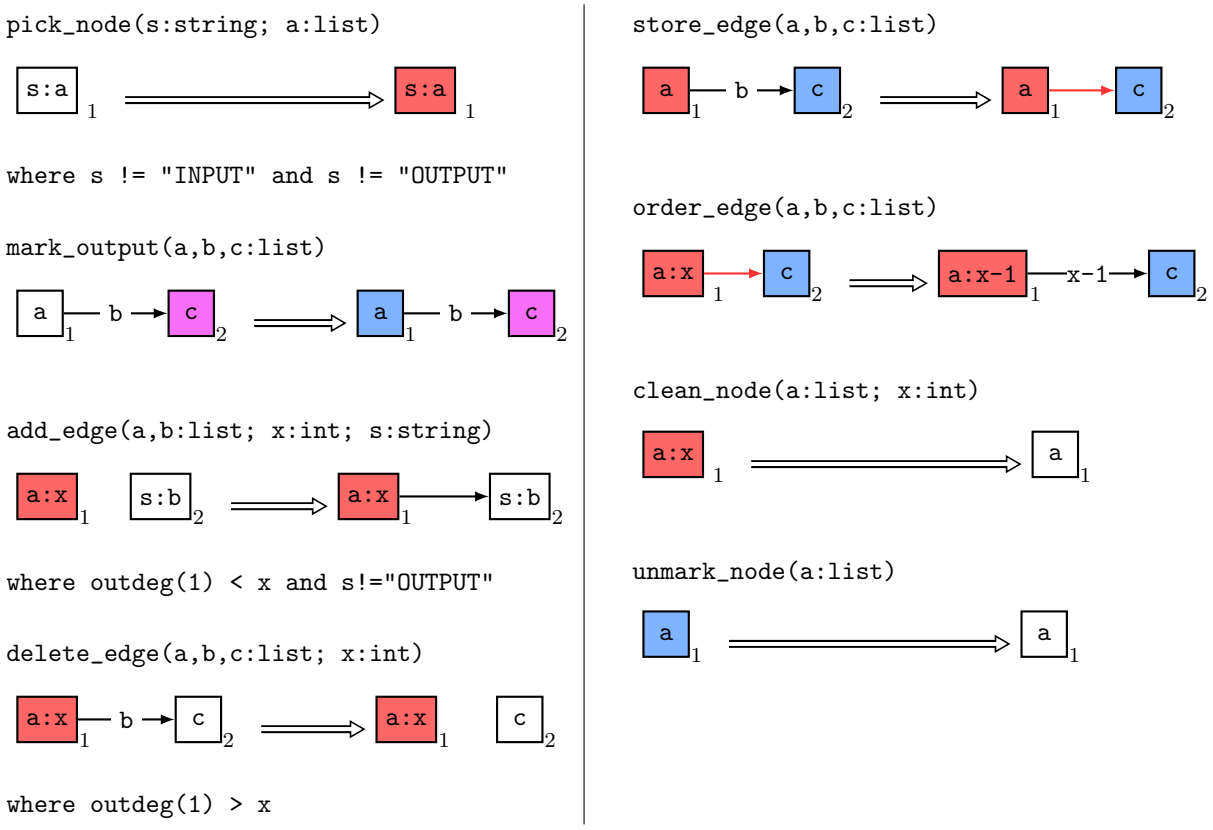


Figure 5.9: A program for mutating a function node in an AFG. For each function in our function set, we have a rule `mutate_node_f`x, which is visualised in Figure 5.10. This mutation respects arity and acyclicity, and shuffles the mutated node's outgoing edges.

This process is implemented by the P-GP 2 program shown in Figure 5.9. An overview of this program is:

1. `[pick_node]`. This rule call selects a function node to mutate uniformly at random and marks that node `red`. By the rule's condition, the selected node cannot be an input or output node.
2. `[mutate_node_f1,...,mutate_node_fk]`. For each function f_x in our set of functions $F = \{f_1, \ldots, f_k\}$, we have a unique rule `mutate_node_f`x in our probabilistically called rule-set which mutates the selected `red` node by relabelling it with the name and arity

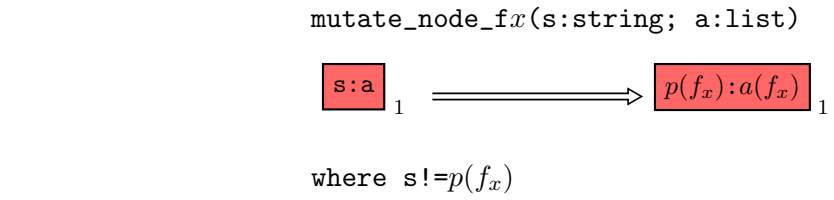


Figure 5.10: When we construct a mutation program such as the one shown in Figure 5.9, we generate a rule `mutate_node_f`x for each function $f_x \in F$ where F is the function set, a is the arity function and p is the naming function. A `red` marked node is relabelled with the name and arity of the function f_x.

of the corresponding function. The general model of these rules is shown in Figure 5.10. The mutated node remains marked `red`. The result of this command is that the chosen function node is mutated to be associated with some other function. The condition of each rule guarantees that the new function is different from the previous function.

3. `mark_output!`. It may be necessary to add new outgoing edges to the mutated node to ensure that its outdegree matches its arity. In the same manner as in edge mutation, we call a rule `mark_output` which iteratively marks every node with a path to the mutated node `blue`. We can then safely insert edges from the mutated node to unmarked nodes in the knowledge that this will not introduce a cycle.

4. `[add_edge]!`. This rule adds edges at random from the `red` marked mutated node to unmarked nodes. The condition guarantees that the rule is called as long as possible while new edges need to be added to make the outdegree match the arity. The newly created edges are unlabelled, as their labels will be assigned when edges are shuffled.

5. `[delete_edge]!`. This rule deletes at random from the `red` marked mutated node. The condition guarantees that the rule is called as long as possible while edges need to be deleted to make the outdegree match the arity.

6. `store_edge!`. This rule marks edges from the `red` marked mutated node to unmarked nodes `red` and removes their labels. By calling this rule as long as possible, we are constructing an ordered set of the mutated nodes outgoing edges as the first stage in shuffling them with respect to ordering indices.

7. `[order_edge]!`. This rule unmarks `red` marked edges from the `red` marked mutated node to unmarked nodes and labels them with the function node's integer label minus 1. At the same time the function node's integer label is decremented by 1. By prob-

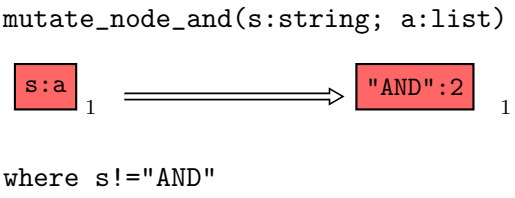


Figure 5.11: An instantiation of the node mutation rule shown in Figure 5.10 with $f_x = \wedge =$ `and`, $a(f_x) = 2$ and $p(f_x) =$ `"AND"`.

abilistically calling this rule as long as possible, the result is that the mutated node's outgoing edges are assigned an order at random.

8. `clean_node`. This rule unmarked the `red` mutated node and removes the integer component of its label. This returns the mutated node to a normal function node state.
9. `unmark_node!`. This rule removes `blue` marks from the graph, reversing the marking effect of the `mark_output!` call.

As a clarifying visual, we give an instantiation of the node mutation rule shown in Figure 5.10 with $f_x = \wedge =$ `and`, $a(f_x) = 2$ and $p(f_x) =$ `"AND"`, shown in Figure 5.11.

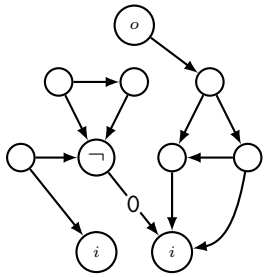
We give an example execution of our mutation in Figure 5.12. In this diagram, most node and edge labels are not shown to aid visual clarity. A function node associated with a negation function $\neg$ is relabelled to be associated with a AND function $\wedge$. An edge is inserted to respect the new node's arity while preserving acylicity, and the node's outgoing edges are shuffled to a random order. The correctness of node mutation with respect to acylicity follows as an extension of the correctness of edge mutation described in Section 5.3.1. The correctness with respect to arity is straightforwardly seen in the semantics of our program with the rules `add_edge` and `delete_edge` and does not therefore require an extended proof.

5.3.3 Binomial Mutation

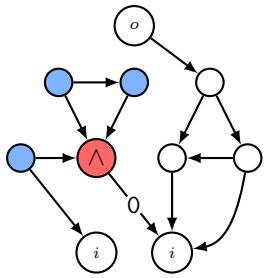
To control the mutation process and induce larger steps in the landscape composed of multiple atomic mutations, we introduce the notion of binomial mutation controlled by a mutation rate parameter.

The mutation rate of an individual is m_r. Certain mutations may prevent other mutations. For example, mutating one edge to target some node may then prevent other mutations of that node's outgoing edges with respect to preserving acyclicity. Therefore, iterating through

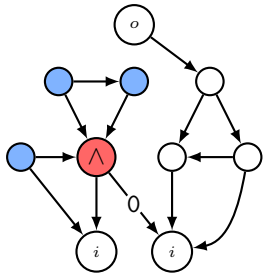
This individual is to undergo a node mutation preserving acyclicity and respecting arity.



(1) `[pick_node]; [mutate_node_and, ...]; mark_output!`:
A function node (now marked `red`) is chosen uniformly at random and mutated by a randomly chosen `mutate_node_`f_x rule - in this case the rule corresponds to an $\wedge$ function. The call to `mark_output` then identifies all nodes with a path to the mutated node, marking them `blue`



(2) `[add_edge]!; [delete_edge]!`:
As the arity of $\wedge$ is 2 and the node's outdegree is 1, the call to `add_edge` as long as possible creates 1 new edge targeting a randomly chosen non-`blue` node thereby preserving acylicity. Correspondingly, there are no successful applications of `delete_edge`.



(3) `store_edge!; [order_edge]!; clean_node; unmark_node!`:
The mutated node's outgoing edges are assigned a random order by `store_edge!; [order_edge]!;`. The call to `clean_node; unmark_node!` returns the successfully mutated graph to an unmarked state.

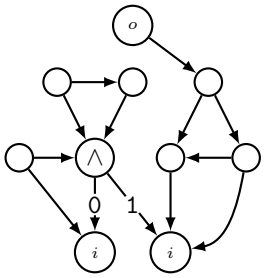


Figure 5.12: A trace of the application of the node mutation program in Figure 5.9. For visual simplicity, (most) node and edge labels have been omitted.

Algorithm 6 The $1 + \lambda$ EA with neutral drift enabled.

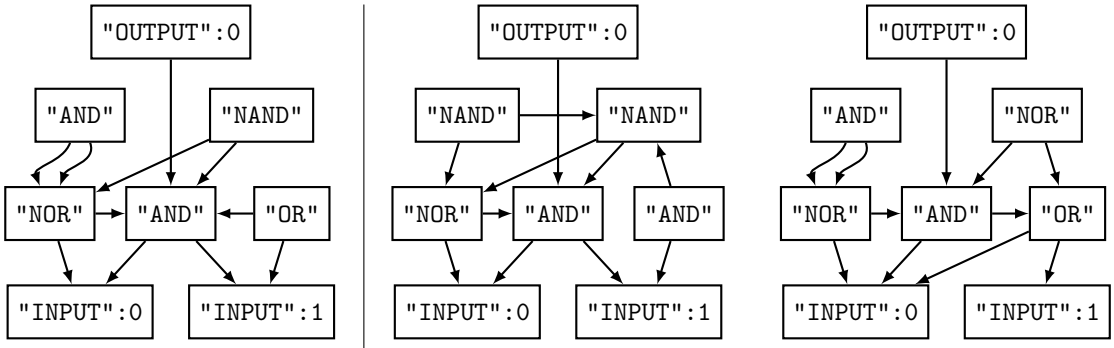
```
1: procedure 1 + λ(max_generations, λ)
2:     parent ← generate_individual
3:     parent_score ← evaluate(parent)
4:     generation ← 0
5:     while solution not found and generation ≤ max_generations do
6:         new_parent ← parent
7:         for i = 0 to λ do
8:             child ← mutate(parent)
9:             child_score ← evaluate(child)
10:            if child_score ≤ parent_score then
11:                new_parent ← child
12:                parent_score ← child_score
13:            end if
14:        end for
15:        parent ← new_parent
16:        generation ← generation + 1
17:    end while
18: end procedure
```

the individual and considering each node or edge in turn for mutation may introduce bias. So our point mutations first choose a random point to mutate, and then mutate it.

We calculate the number of node or edge mutations to apply based on binomial distributions. For an individual with v_f function nodes and e edges, with mutation rate m_r, we sample a number of node mutations, $m_v \in \mathcal{B}(v_f, m_r)$, and edge mutations, $m_e \in \mathcal{B}(e, m_r)$, where $\mathcal{B}(n, p)$ indicates a binomial distribution with n trials and probability of success, p. We then place all $m_v + m_e$ mutations in a list, and shuffle the list, applying mutations in a random order. While this approach is likely to have some biases, it guarantees reproducible probabilistic behaviour. The overall expected number of atomic mutations is $m_r(v_f + e)$.

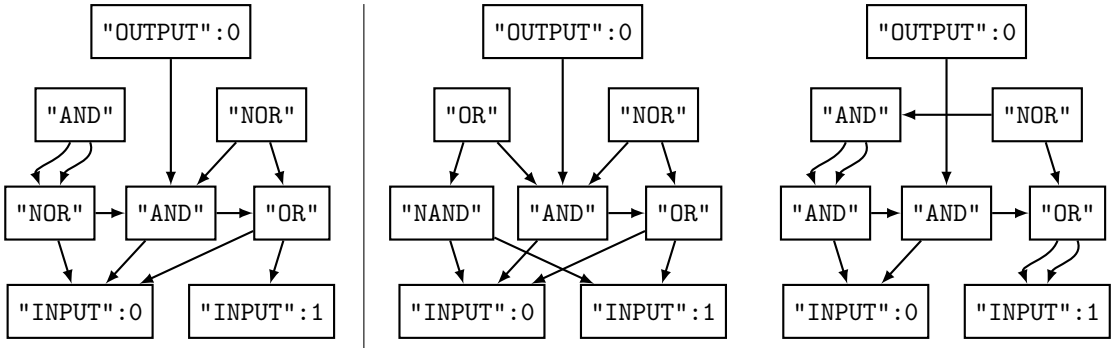
5.4 $1 + \lambda$ Evolutionary Algorithm

Recombination of graphs is in itself a challenging research area that will be considered in Chapter 9. In this chapter we use only mutation operators, and it is therefore natural to



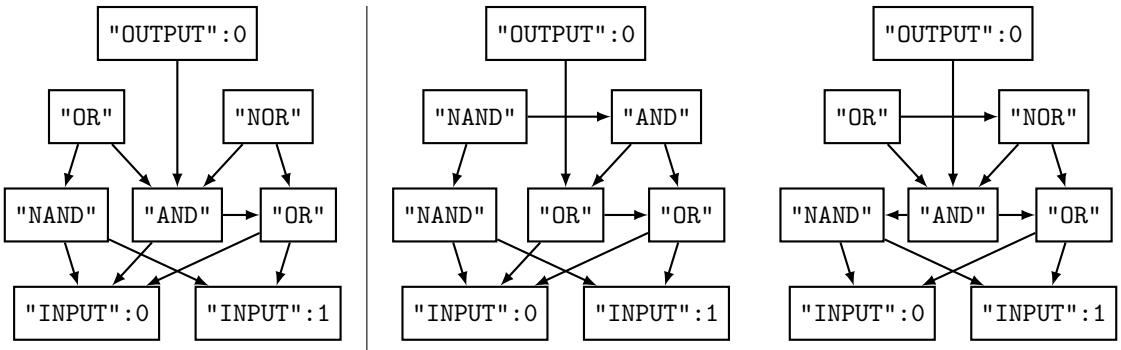
Gen 1, Fitness = 3 | **Child 1**, Fitness = 3 | **Child 2**, Fitness = 2

⇓ **Child 2** replaces the parent ⇓



Gen 2, Fitness = 2 | **Child 1**, Fitness = 2 | **Child 2**, Fitness = 3

⇓ **Child 1** replaces the parent (Neutral Drift) ⇓



Gen 3, Fitness = 2 | **Child 1**, Fitness = 1 | **Child 2**, Fitness = 0

Child 2 is a correct implementation of an XOR gate.

Figure 5.13: A visualisation of an EGGP evolutionary run learning an XOR gate from AND, OR, NAND and NOR gates.

consider single-survivor EAs. As we intend to benchmark against CGP, we propose the use of the EA most commonly used with it, the $1 + \lambda$ EA shown in Algorithm 6. This algorithm is an extended form of Random Hill Climbing, where in each generation λ new individuals are generated by mutating the sole surviving parent from the previous generation. Additionally, we allow a new individual with equal fitness to its parent to replace its parent in the next generation, facilitating the phenomena of "neutral drift". Propagating changes in the genotype which result in neutral changes in the phenotype is known to positively influence the performance of CGP [156] and we see no obvious reason why this would not also be the case in EGGP.

5.5 Example: Learning an XOR Gate

In this Section we give a visual example of how EGGP can learn an AFG to fit a given fitness function. We focus on the evolution of an implementation of an XOR gate with truth table

i_0	i_1	$o_0 = i_0 \oplus i_1$
0	0	0
0	1	1
1	0	1
1	1	0

(5.8)

from AND, OR, NAND and NOR gates. An idealised evolutionary run for this problem is given in Figure 5.13. Here each individual consists of 1 output node, 5 function nodes and 2 input nodes. The initial graph implements an AND gate on the 2 inputs, yielding a fitness of 3. This evolutionary run is using the $1 + \lambda$ EA with $\lambda = 2$, so in each generation, 2 children are produced. In the first generation, one of the children (Child 2) effectively implements an identity on input 0, giving a fitness value of 2. As this is less than the fitness of the parent, Child 2 replaces the initial graph moving into the next generation.

In the second generation, a child is produced (Child 1) where all active components of the parent are unmodified. Instead, only inactive material is modified, giving this child exactly the same fitness as its parent. As a result, Child 1 replaces the parent moving into the next generation. In the final generation, the second child (Child 2) implements an XOR gate. As this is a correct solution with a fitness of 0, the evolutionary run terminates.

5.6 Related Work

There are a number of approaches to evolving graphs and graph-like programs, indeed many of these are discussed in far greater detail in Section 2.4. Of particular note are:

- Tree-Based Genetic Programming (TGP) [129]. This GP approach can be viewed as learning tree-structured AFGs without sharing.
- CGP [157]. This approach places function nodes on a grid and expresses mutation operators on that grid. Acyclicity is preserved by an ordering imposed on the grid.
- PDGP [188]. Similarly, PDGP also places function nodes on an ordered grid. Unlike CGP, PDGP typically heavily utilises recombination.

However, we will draw a particular comparison with CGP. The reason for this is two-fold:

1. Focusing on the graph structure used in CGP, ignoring the underlying grid, the representations used in EGGP and CGP are highly similar.
2. In both EGGP and CGP, the typical genetic operators used are:
   a) Atomic node mutation, where a single node is associated with some new function.
   b) Atomic edge mutation, where a single edge is redirected while preserving acylicity.

Hence we perceive particular value in clarifying the *differences* between EGGP and CGP. We note that some of this discussion may well be relevant to PDGP also. In Section 5.6.1 we briefly revisit the relevant aspects of CGP necessary for this comparison. In Section 5.6.2 we demonstrate that EGGP generalises the landscapes associated with CGP both in terms of representation and neighbourhoods induced by the available genetic operators.

5.6.1 Cartesian Genetic Programming

CGP is a type of EA in which individuals are represented as linear sequences of genes corresponding to a Directed Acyclic Graph (DAG). Each gene is an integer representing either (1) where a node gets its inputs from or (2) the function of a node. These nodes are ordered so that all input connections must respect that ordering, preventing cycles. When evolving over a function set where each function takes 2 inputs, there are 3 genes for each node in the individual; 2 representing each of the node's inputs, and 1 representing the node's function. Outputs are represented as single genes describing the node in the individual which corresponds to that output. These connection genes (nodes' input genes and the singular output

	Node 1			Node 2			Node 3			Output
Genotype:	0	0	1	1	0	2	0	1	0	2

Phenotype:

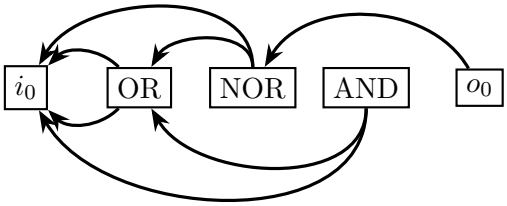


Figure 5.14: The genotype-phenotype mapping of a simple CGP individual consisting of 1 input, 3 nodes and 1 output and arity 2. Each node is represented by 3 genes; the first 2 describe the indices of the node's inputs (starting at index 0 for the individual's input i) and the third describing the node's function. Function indices 0, 1 and 2 correspond to AND, OR and NOR respectively. The final gene describes the index of the node used by the individual's output o.

genes) point to other nodes based on their index in the ordering.

An example genotype-phenotype mapping is given in Figure 5.14. Here an individual consisting of 3 nodes over a function set of arity 2, 1 input and 1 output is represented by 10 genes. These genes decode into the shown DAG. In CGP individuals may be seen as a grid of n_r rows and n_c columns; a node in a certain column may use any node from any row in an earlier column as an input. Hence the total $n = n_r \times n_c$ nodes are ordered under a $\leq$ operator. The example shown in Figure 5.14 is a single row instance of CGP.

5.6.2 Comparison with Cartesian Genetic Programming

Here we demonstrate that EGGP provides a richer representation than CGP:

- For a fixed number of nodes n and function set F, any CGP individual can be represented as an EGGP individual, whereas the converse may not always hold when the number of rows in a CGP individual is greater than one.
- Any order-preserving CGP mutation can be represented as a feed-forward preserving mutation in EGGP, whereas some feed-forward preserving mutations may not be order-preserving nor valid in the CGP framework.

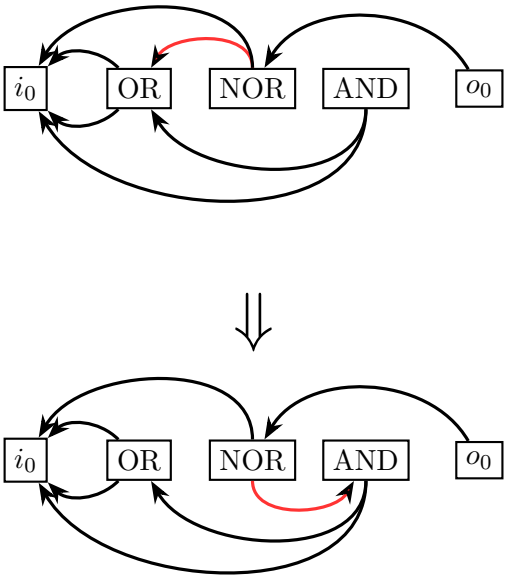


Figure 5.15: An acyclicity preserving edge mutation. An edge (red) is redirected from the OR node to the AND node. This mutation produces a valid circuit but, assuming that nodes are ordered from left to right, is impossible in CGP as it does not preserve order.

Firstly, consider the genotype-phenotype decoding of a CGP individual. Here we have clearly defined sets of input, output and function nodes. Additionally each function node is associated with some function from the function set, and there are ordered input connections (edges) from each function node to its inputs. Clearly this decoded individual can be treated as an AFG. Conversely, consider the case where $n_r > 1$. Then there is the trivial counter example of an EGGP individual with a solution depth greater than n_c (as $n > n_c$) which clearly cannot be expressed as a CGP individual limited to depth n_c.

We now consider mutations available over a CGP individual in comparison to those for an EGGP individual where feed-forward preserving mutations are used. Clearly, as each order preserving mutation is feed-forward preserving, any valid mutation for the CGP individual is available for its EGGP equivalent. However, consider the example shown in Figure 5.15. Here a feed-forward mutation, redirecting the red edge from the OR gate to the AND gate, is available in the EGGP setting but is not order preserving so is impossible in the CGP setting. Additionally, the semantic change that has occurred here, where an active node has been inserted between two adjacent (with respect to node ordering) active nodes is a type

of phenotype growth that is impossible in CGP. Hence every mutation available in CGP is available in EGGP for an equivalent individual but the converse may not be true.

Therefore the landscape described by EGGP over the same function set and number of nodes is a generalisation of that described by CGP, with all individuals and viable mutations available, alongside further individuals and mutations that were previously unavailable.

5.7 Conclusions and Future Work

In this chapter we have described the approach EGGP. This technique synthesises acylic graph-like programs such as AFGs using evolutionary computation and landscapes described by graph programs. The system consists of 4 core components which we have described:

1. A simple initialisation procedure that generates AFGs.
2. The $1 + \lambda$ EA with a minimal surviving population and no crossover.
3. An atomic edge mutation that preserves acylicity.
4. An atomic node mutation that preserves acylicity and function arity while shuffling a function node's outgoing edges.

For both atomic mutations, we have set out arguments for their correctness with respect to the domain we are considering. A simple example of learning an XOR gate has been given to show how these simple components interact to solve problems. Despite its simplicity, Chapter 6 demonstrates the empirical effectiveness of the described approach on various standard benchmark problems from the literature. Our comparison with related work highlights that the landscapes we are inducing with graph programs are strict generalisations of those accessed with CGP. This further strengthens the case for using graph programming as a language for describing EAs over graphs, as we are able to describe apparently novel, non-trivial genetic operators with concise and intuitive P-GP 2 programs.

Due to its simplicity, it is straightforward to use EGGP as a building block to access new ideas and previously unvisited techniques for the evolution of graphs. In Chapter 8, EGGP is extended to incorporate known equivalence laws which accelerate the evolutionary process. In Chapter 9, EGGP is combined with a passive genetic recombination operator which aids the system in finding higher quality solutions faster. However it is clear that this is not the limit of the possible extensions to EGGP, and that there are many areas for potential future work. We outline some of these possibilities below:

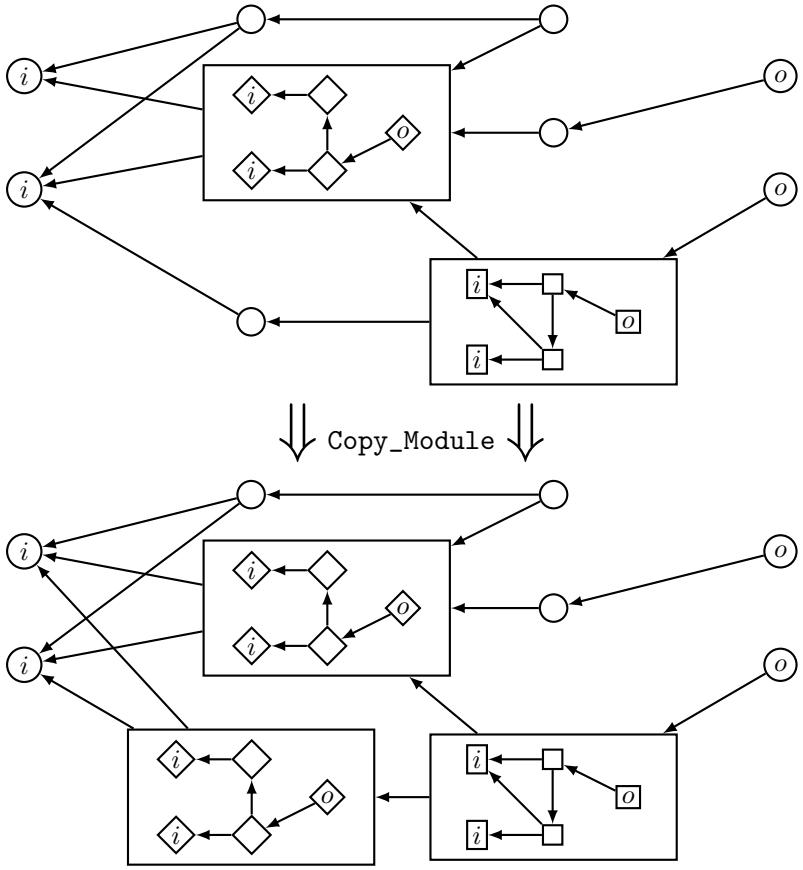


Figure 5.16: A notion of how subgraph copying (see [56]) might be utilised in a modular extension of EGGP to copy entire modules between function nodes.

- **Typed EGGP**. As described in Section 4.4, it may be possible to extend FGs to Typed FGs (TFGs) via Type Graphs. An obvious direction of research, then, is to investigate what genetic operators are required to effectively evolve TFGs while respecting the underlying type system. As the type system is effectively specifying a restriction on allowed connections in a TFG, it is likely that the behaviours of atomic genetic operators for TFGs are strict restrictions of the behaviours of the atomic node and edge mutations we have described in this chapter.
- **Hierarchical EGGP**. Another idea described in Section 4.4 is that FGs can be extended to Hierarchical FGs (HFGs) using pre-existing notions of hierarchical graphs. While the genetic operators we have described in this chapter readily extend to ADF hierarchical graphs where modules are evolved as independent components of a larger

genome, a possible area of research is in the use of Hierarchical Graph Transformation concepts such as subgraph copying and subgraph deletion [56] as a model for modifying HFGs where modules are embedded within function nodes. We give a notion of how subgraph copying might be used in Figure 5.16.

- **Other Genetic Operators**. There are a number of other ideas for genetic operators on graphs in the literature which might be recreated and utilized in EGGP. For example, it is possible to describe a P-GP 2 program which targets only active nodes/edges, thereby achieving 'active-only' mutations as in [244]. Similarly, it would be possible to include operators which may 'activate' or 'deactivate' nodes (or entirely add/delete nodes!) as in [111]. We refer the reader to the discussion of mutation in [158] for more ideas from CGP literature on possible genetic operators for EGGP.

6 Benchmarking EGGP

Abstract

In this chapter we present benchmark results comparing Evolving Graphs by Graph Programming (EGGP) to popular Genetic Programming (GP) approaches from the literature. We draw statistical comparisons with results produced using Tree-Based GP (TGP) and Cartesian Genetic Programming (CGP) on standard digital circuit and symbolic regression benchmark problems. We find that EGGP outperforms other approaches under standard parameters on many of the studied problems.

Relevant Publications

Content from the following publications is used in this chapter:

[8] T. Atkinson, D. Plump, and S. Stepney, "Evolving graphs by graph programming," in *Proc. European Conference on Genetic Programming, EuroGP 2018,* ser. LNCS, vol. 10781. Springer, 2018, pp.35–51.

[9] T. Atkinson, D. Plump, and S. Stepney, "Probabilistic graph programs for randomised and evolutionary algorithms," in *Proc. International Conference on Graph Transformation, ICGT 2018,* ser. LNCS, vol. 10887. Springer, 2018, pp. 63–78.

[10] T. Atkinson, D. Plump, and S. Stepney, "Evolving graphs with horizontal gene transfer," in *Proc. Genetic and Evolutionary Computation Conference, GECCO 2019,* ACM, 2019, pp. 968–976.

6.1 Introduction

To verify our approach, we require empirical comparisons with other approaches from the literature. In this chapter, we study the performance of Evolving Graphs by Graph Programming (EGGP) on standard problems in relation to Tree-based Genetic Programming (TGP) and Cartesian Genetic Programming (CGP).

The problems studied in this chapter are broken into 2 categories:

1. **Digital circuit**. In the digital circuit benchmark problems, the task is to find a combination of logic gates that induce a truth table equal to the target truth table. Many real world circuits, such as digital adders, multipliers and parity checks, are studied. The measurement of quality on these problems is the number of evaluations required to completely solve a given problem.
2. **Symbolic regression**. In the symbolic regression benchmark problems, the task is to find a combination of real-valued functions which minimise the error on a synthetic dataset. On these problems the number of evaluations is presented as a fixed budget, and the measurement of success is the error of final solutions found for a given problem.

For digital circuit benchmark problems, we compare to CGP [155] as the most relevant approach. The 16 studied problems and experimental parameters are in general drawn from CGP literature [155, 253]. For symbolic regression benchmark problems, we compare to CGP as the most relevant approach, and TGP [129] as the typical approach used for solving such problems. The 14 studied problems and experimental parameters for TGP are drawn from [169]. The experimental parameters for CGP are taken from symbolic regression benchmarks in [155]. We find statistically significant differences in many of the digital circuit problems, and we find that the differences in performance between EGGP and CGP increase with problem difficulty. We observe less statistical differences in our symbolic regression comparisons.

The rest of this chapter is arranged as follows. In Section 6.2 we describe our use of statistics through the remainder of this thesis. In Section 6.3 we describe our digital circuit benchmark experiments; the results of these experiments are given in Section 6.4. We provide further discussion and clarifying experiments on the distinction between EGGP and CGP in Section 6.5. In Section 6.6 we describe our symbolic regression benchmark experiments; the results of these experiments are given in Section 6.7. We discuss the differences between the results we observe in our digital circuit benchmarks and symbolic regression benchmarks in Section

6.8, and provide some plausible explanations for these differences. Finally in Section 6.9 we summarise our findings and set out further problems on which EGGP could be benchmarked.

6.2 Statistical Comparison throughout this Thesis

Throughout the remainder of this thesis we often compare approaches on a set of problems. While the metric of success may vary, we do not assume that our data is normally distributed - indeed often the opposite is the case, with a few 'bad' evolutionary runs introducing heavy tails to the observed distributions. For this reason, we will in general report the median of our metrics, rather than the mean, and the interquartile range in our metrics, rather than the variance or standard deviation.

To test for statistical significance we in general use the two-tailed Mann–Whitney U test [147], which (essentially) tests the null hypothesis that two distributions have the same medians (the non-parametric analogue of the t-test applicable only to normally distributed data). Throughout our experiments we use a significance threshold of $p < 0.05$ and perform a Bonferroni procedure for family of hypotheses tests. We view a family of hypothesis tests to be the set of statistical tests performed comparing the performance of two approaches across a family of related problems. For example, in Section 6.3 we compare EGGP to CGP on 16 different digital circuit synthesis tasks. We therefore use a corrected significance threshold of $p < \frac{0.05}{16}$. In general, when we are testing a family of m hypotheses, we use a corrected significance threshold of $p < \frac{0.05}{m}$.

In the case where we get a statistically significant result ($p < \frac{0.05}{m}$), we also calculate the effect size, using the non-parametric Vargha–Delaney A Test [248]. This gives a quantitative metric of the magnitude of the differences we observe. When we find that $A > 0.71$ a given result is said to have 'large effect'.

6.3 Digital Circuit Experiments

Our digital circuit benchmark problems are drawn from CGP literature [155, 253]. On our digital circuit benchmark problems, the task is to synthesise a circuit with an exact truth table from a function set of logic gates. Therefore our fitness is a measure of absolute distance from an individual graph's truth table and a target truth table. For example, if an individual

had truth table

i_0	i_1	o_0
0	0	0
0	1	0
1	0	1
1	1	1

(6.1)

and the target truth table was that of an XOR gate, i.e.,

i_0	i_1	$o_0 = i_0 \oplus i_1$
0	0	0
0	1	1
1	0	1
1	1	0

(6.2)

then the fitness of the individual would be 2. That is, there are 2 incorrect output bits in the truth table (for inputs $0, 1$ and $1, 1$). A solution is considered optimal when their fitness is 0.

We study a number of problems with various degrees of complexity. The simplest problems, such as the 3-bit even parity problem, provide 3 inputs and expect 1 output and can be specified quite easily by hand. The hardest problems, such as the 3-bit multiplier, provide 6 inputs and expect 1 output and take substantial thought to design manually from scratch. A full listing of the problems we study is given in Table 6.1.

Our problems can be broken up into several classes of circuit:

- **Adders**. These are circuits which add together a pair of binary representations of integers. For example, a 2-bit adder presented with 2 (10), 1 (01) and a carry bit of 1 is expect to produce as output $2 + 1 + 1 = 4$ (100).
- **Multipliers**. These are circuits which multiply together a pair of binary representations of integers. For example, a 2-bit multiplier, presented with 2 (10) and 3 (11) is expected to produce as output $2 \times 3 = 6$ (0110).
- **3:8-bit De-Multiplexer**. This is a specialist circuit which converts a 3-bit representation of an integer to a one-hot encoding. For example, the value 5 (101) is converted to 00010000.
- **4x1-bit Comparator**. This circuit compares 4 input bits pair-wise, producing for each pair a $<$ bit, $=$ bit and $>$ bit where the appropriate output is 1 depending on the inputs. For example, if the input to the circuit were 1010 then the comparison of the

Digital Circuit	Number of Inputs	Number of Outputs
1-bit Adder (1-Add)	3	2
2-bit Adder (2-Add)	5	3
3-bit Adder (3-Add)	7	4
2-bit Multiplier (2-Mul)	4	4
3-bit Multiplier (3-Mul)	6	6
3:8-bit De-Multiplexer (DeMux)	3	8
4×1-bit Comparator (COMP)	4	18
3-bit Even Parity Check (3-EP)	3	1
4-bit Even Parity Check (4-EP)	4	1
5-bit Even Parity Check (5-EP)	5	1
6-bit Even Parity Check (6-EP)	6	1
7-bit Even Parity Check (7-EP)	7	1
5-bit Odd Parity Check (5-OP)	5	1
6-bit Odd Parity Check (6-OP)	6	1
7-bit Odd Parity Check (7-OP)	7	1
8-bit Odd Parity Check (8-OP)	8	1

Table 6.1: Digital circuit benchmark problems.

first 2 bits would yield 001 ($1 > 0$) and the comparison of the first and the third bits would yield 010 ($1 = 1$). With 6 pairs, each producing 3 outputs, the circuit produces 18 outputs in total.

- **Even Parity Checks**. These are circuits which compute whether the number of 1s in the input is even. For example, a 3-bit even parity circuit presented with input 011 would return 1 as there are an even number of 1s. In comparison, if the input were 100 then the circuit would return 0.
- **Odd Parity Checks**. These are similar to even parity circuits except that they return 1 when the number of 1s in the input is odd.

As many of these circuits are typically constructed manually using XOR gates, we use the function set $\{\text{AND}, \text{OR}, \text{NAND}, \text{NOR}\}$ to artificially increase the difficulty of these problems. We use the number of incorrect bits produced by a candidate solution in comparison to the full truth table of the given problem as the fitness function.

We produce our own CGP benchmark results, which are roughly in line with those available in [254], by using the C-based CGP library [243]. Each algorithm is run 100 times, with a maximum generation cap of 20,000,000; every run in each case successfully produced a result with the exception of the 3-Mul for CGP, which produced a correct solution in 99% of cases. In all benchmarks, 100 function nodes are used for each individual. Following conventional wisdom for CGP, we use a mutation rate of 0.04 for CGP benchmarks. Additionally, a single row of nodes is used in each of these cases ($n_r = 1$). However, from our observations EGGP works better with a lower mutation rate, so for EGGP benchmarks we use 0.01. An investigation of how mutation rate influences the performance in EGGP is left for future work. The $1 + \lambda$ algorithm is used in all both cases, with $\lambda = 4$.

To provide comparisons, we use the following metrics; median number of evaluations (ME), median absolute deviation (MAD)[1] and interquartile range (IQR). The number of evaluations taken for each run is calculated as the number of generations used multiplied by the total population size ($1 + \lambda = 5$). The hypothesis that we are investigating in these experiments is that EGGP performs significantly better than CGP on the same problems under similar conditions. This hypothesis, if validated, would demonstrate the value of our approach.

6.4 Digital Circuit Results

Here we present results from our benchmarking experiments. Digital circuit results for EGGP and CGP are given in Table 6.2.

To test for statistical significance we use the two-tailed Mann–Whitney U test with a significance threshold of $\alpha = \frac{0.05}{16}$ as corrected by a Bonferonni procedure. In the case where we get a statistically significant result, $p < \frac{0.05}{16}$, we also calculate the effect size, using the non-parametric Vargha–Delaney A Test.

Comparing EGGP to CGP in Table 6.2, we find no significant improvement of EGGP over CGP for small problems (1-Add, 2-Mul, DeMux, 3-EP). As the problems get larger and harder we find significant ($p < \frac{0.05}{16}$) improvement of EGGP over CGP in all cases. The effect size is medium ($0.64 < \mathrm{A} < 0.71$) for 4-EP. We find significant ($p < \frac{0.05}{16}$) improvements along with large effect sizes ($0.71 > \mathrm{A}$) on all other problems, including the most difficult problems: 3-Add, 3-Mul, 4 × 1-Bit Comparator, 7-Bit Even Parity and 8-Bit Odd Parity. So there is a clear progression of increasing improvement with problem difficulty.

[1] Median of the absolute deviation from the evaluation median ME.

	EGGP			CGP				
Problem	ME	MAD	IQR	ME	MAD	IQR	p	A
1-Add	5,723	3,020	7,123	6,018	3,312	7,768	0.62	–
2-Add	74,633	32,863	66,018	180,760	88,558	198,595	10^{-15}	**0.82**
3-Add	275,180	114,838	298,250	2,161,378	957,035	1,837,942	10^{-31}	**0.97**
2-Mul	14,118	5,553	12,955	10,178	5,258	14,459	0.018	-
3-Mul	1,241,880	437,210	829,223	15,816,940	7,948,870	19,987,744	10^{-34}	**0.99**
DeMux	16,763	4,710	9,210	20,890	6,845	14,063	0.013	-
COMP	262,660	84,248	174,185	1,148,823	425,758	1,012,149	10^{-31}	**0.97**
3-EP	2,755	1,558	4,836	4,365	2,530	5,345	0.038	-
4-EP	13,920	5,803	11,629	22,690	11,835	24,340	10^{-6}	0.69
5-EP	34,368	15,190	30,054	106,735	55,615	126,063	10^{-18}	**0.86**
6-EP	83,053	33,273	66,611	485,920	248,150	535,793	10^{-3}	**0.97**
7-EP	197,575	61,405	131,215	1,828,495	843,655	1,860,773	10^{-33}	**0.99**
5-OP	38,790	13728	29,490	96,372	41,555	91,647	10^{-18}	**0.86**
6-OP	68,032	22,672	52,868	502,335	274,132	600,291	10^{-31}	**0.97**
7-OP	158,852	69,477	142,267	1,722,377	934,945	2,058,077	10^{-33}	**0.99**
8-OP	315,810	128,922	280,527	7,617,310	4,221,075	9,830,470	10^{-34}	**0.99**

Table 6.2: Results from digital circuit benchmarks for CGP and EGGP. The p value is from the two-tailed Mann–Whitney U test. Where $p < \frac{0.05}{16}$, the effect size from the Vargha–Delaney A test is shown; large effect sizes ($A > 0.71$) are shown in **bold**. The values for CGP on the 3-Mul problem include the single failed run.

We visualise some highly significant results as box-plots, with raw data overlayed and jittered, in Figure 6.1. For each of the named problems, it can be clearly seen that EGGP's interquartile range shares no overlap with CGP's, highlighting the significance of the improvement made. Overall, we see these results to validate our hypothesis that EGGP performs significantly better than CGP when addressing the same harder problems, although we note that no significant improvement is made for some simpler problems.

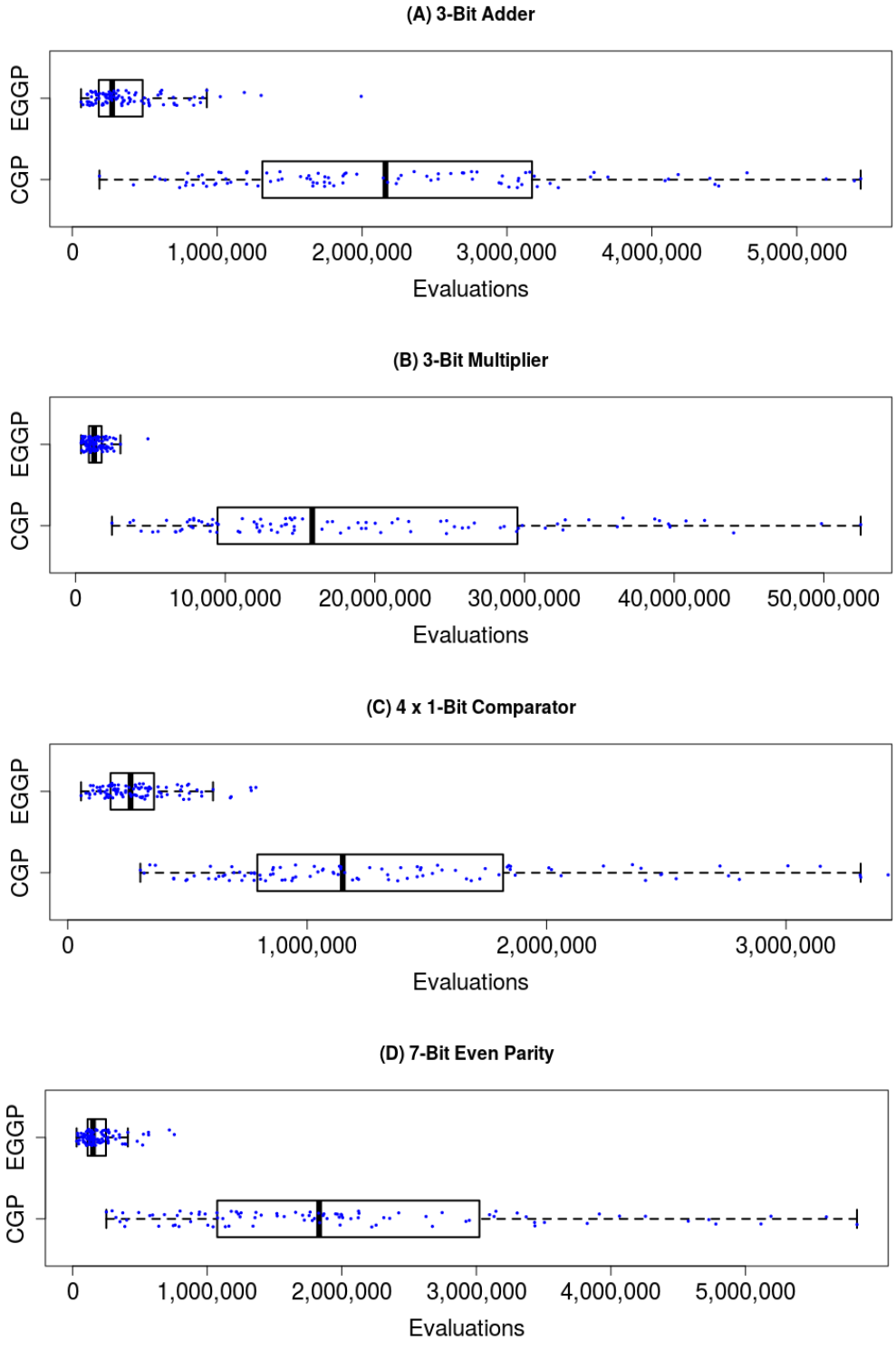


Figure 6.1: Box-plots with data overlayed for the following highly significant results; (A) 3-bit Adder, (B) 3-bit Multiplier, (C) 4 x 1-bit Comparator and (D) 7-bit Even Parity. Overlayed data is jittered for visual clarity.

6.5 Digital Circuit Discussion

The comparisons we have seen offer a unique opportunity to validate the claim made in Section 5.6.2 that EGGP generalises the landscape provided by CGP. To this end, we proposed the use of what we call Ordered-EGGP (O-EGGP).

Each node in an O-EGGP individual is associated with an order in an analogous manner to CGP. Node function mutations from EGGP are used, but input mutations are order-preserving rather than feed-forward preserving. Hence the same set of atomic mutations is available for equivalent O-EGGP and CGP individuals. This approach simulates the landscape and a very similar search process of CGP under identical conditions, so should produce highly similar results to an equivalent CGP implementation. By also benchmarking O-EGGP we demonstrate that it is EGGP's free graphical representation and the associated more general ability to mutate input connections with respect to preserving feed-forwardness that yields higher quality results.

By setting the mutation rate to 0.04, the last meaningful difference between O-EGGP and CGP is in the interpretation of mutation rate. In CGP, each function node and edge is mutated individually with probability equal to the mutation rate. In O-EGGP, we still have that the unit to mutate is chosen at random as the first step of mutation. Hence, although the binomial mutation operator provides a similar probability distribution over outcomes, there are clearly some small distances. This is shown by the fact that it is possible to mutate an edge twice in O-EGGP during the same mutation step.

We evaluate O-EGGP on a subset of the studied digital circuit benchmark problems. We argue that if the results from these benchmarks are in line with the CGP benchmark results we may extrapolate that O-EGGP is indeed approximately simulating CGP. We are effectively testing the hypothesis that O-EGGP does not perform significantly better or worse than CGP on the same problems under identical conditions. This hypothesis, if validated, would indicate that the possible factors influencing EGGP's greater performance for the first hypothesis would be reduced to the use of the feed-forward mutation operator and the mutation rate. The results of our comparisons are given in Table 6.3.

Overall we find no significant difference between either approach on any of the problems in the smaller benchmark set. The results show similar numbers of MEs in each case, and produce p values indicating no significant difference between the samples. We believe that these findings support our hypothesis that O-EGGP does not perform significantly better or worse than CGP on identical problems under identical conditions. As O-EGGP approxi-

	O-EGGP			
Problem	ME	MAD	IQR	p
1-Add	6,253	3,610	9036	0.66
2-Add	193,753	109,420	239,133	0.95
2-Mul	13,930	7,905	19,104	0.12
DeMux	21,406	5,115	10,065	0.66
3-EP	3,903	2,315	4,831	0.64
4-EP	23,360	11,893	21,865	0.84
5-EP	121,820	51,150	107,868	0.56

Table 6.3: Results from digital circuit benchmarks for O-EGGP on a smaller benchmark suite. The p value is from the two-tailed Mann–Whitney significance test comparing against CGP results in Table 6.2; no result is statistically significant ($\alpha = \frac{0.05}{7}$).

mately simulates CGP, this indicates that we can consider the differences between the runs of EGGP and O-EGGP, namely feed-forward preserving mutations and mutation rate, as the major contributors to the differences in performance shown in Table 6.2. These findings empirically validate our claim in Section 5.6.1 that EGGP generalises CGP's landscape, with O-EGGP's landscape a clearly defined subset of EGGP's.

Further, we suggest that the significant differences in results would not be resolved by tuning the mutation rate parameter. Therefore we turn our attention to the feed-forward preserving edge mutation operator. As feed-forward preserving mutations may insert nodes between nodes that would be considered adjacent in the CGP framework. This allows a subgraph of the solution to grow and change in previously unavailable manners. Performing functionally equivalent mutations with order preserving edge mutations might require the construction of an entirely new subgraph in the neutral component of the individual which is then activated. We propose that the former mutation is more likely to occur than the sequence of mutations required to achieve the latter. Therefore where those unavailable mutations are "good" mutations in the sense of the fitness function, better performance will be achieved by using them directly. A future investigation into the quality of the neighbourhood when using the feed-forward preserving mutation would clarify this hypothesis.

Additionally, this ability to insert material from anywhere in the individual that preserves feed-forwardness allows various neutral drifts to occur in the active component, even between

nodes that would be considered adjacent in the CGP framework. For example, a connection using node x as input could be replaced by the semantically equivalent AND(x, x), for the function set used here. The insertion of that AND gate would then allow new mutations in the active component; for example changing its function, or mutating one of its inputs. Similar neutral mutations exist in this domain, such as the insertion of double negations using NAND gates. Additionally, the reverses of these transformations are also possible, freeing up genetic material to be used elsewhere. This direction of thought sets the context for the work on Semantic Neutral Drift (SND) described in Chapter 8.

6.6 Symbolic Regression Experiments

We benchmark the approaches on 14 of the 21 synthetic symbolic regression problems [169]. That work justifies the exclusion of Grammatical Evolution (GE) [175], as it finds that TGP generally outperforms GE on these problems. Experiments for benchmarks F13-–17, 19, 20 (omitted here) showed very little variety in performance; the results of [169] suggests these are poor benchmark problems in that the functions are almost invariant on their inputs. While F1–3 also exhibit relatively invariant responses, approaches here and in [17] produce a variety of performances that compel their inclusion. Similarly, while F4 and F21 do not show a variety of performances, the functions themselves produce a variety of responses on different inputs, again compelling their inclusion. For all 14 problems, see Table 6.4.

These benchmarks were introduced in response to various criticisms of the GP community for 'arbitrarily' chosen benchmark problems, and the reasoning for their design is set out in detail in [169]. We view these problems as good measures of performance of a TGP system. Of the 14 problems, 9 take 2 inputs, 1 takes 3 inputs, 3 take 5 inputs and 1 takes 10 inputs. Each function's input variables are randomly sampled from the interval $[-5, 5]$.

We use 1000 training samples, $10,000$ validation samples and $40,000$ test samples [2]. The training data is used to guide the different approaches, while every solution explored is evaluated on the validation data. The globally best performing individual (with respect to the validation data) is returned at the end of a run, and then evaluated on the test data to produce a test performance measure.

The function set for these problems is that of [169] and is given by

$$\{+, -, \times, \div, e^x, \ln(x), \sin(x), \tanh(x), \sqrt{x}\}, \tag{6.3}$$

[2] The author is very grateful to Miguel Nicolau for providing the datasets used in [169]

Name	Number of Inputs	Function
F1	2	$f(x_1, x_2) = \frac{e^{-(x_1-1)^2}}{1.2+(x_2-2.5)^2}$
F2	2	$f(x_1, x_2) =$ $e^{-x_1}x_1^3\cos(x_1)\sin(x_1)(\cos(x_1)sin^2(x_1)-1)(x_2-5)$
F3	5	$f(x_1, x_2, x_3, x_4, x_5) = \frac{10}{5+\Sigma_{i=1}^{5}(x_i-3)^2}$
F4	3	$f(x_1, x_2, x_3) = 30\frac{x_1-1)(x_3-1)}{x_2^2(x_1-10)}$
F5	2	$f(x_1, x_2) = 6\sin(x_1)\cos(x_2)$
F6	2	$f(x_1, x_2) = (x_1-3)(x_2-3) + 2\sin((x_1-4)(x_2-4))$
F7	2	$f(x_1, x_2) = \frac{(x_1-3)^4+(x_2-3)^3+(x_2-3)}{(x_2-2)^4+10}$
F8	2	$f(x_1, x_2) = \frac{1}{1+x_1^{-4}} + \frac{1}{1+x_2^{-4}}$
F9	2	$f(x_1, x_2) = x_1^4 - x_1^3 + \frac{x_2^2}{2} - x_2$
F10	2	$f(x_1, x_2) = \frac{8}{2+x_1^2+x_2^2}$
F11	2	$f(x_1, x_2) = \frac{x_1^3}{5} + \frac{x_2^3}{3} - x_2 - x_1$
F12	10	$f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}) =$ $x_1x_2 + x_3x_4 + x_5x_6 + x_1x_7x_9 + x_3x_6x_{10}$
F18	5	$f(x_1, x_2, x_3, x_4, x_5) = x_1x_2x_3x_4x_5$
F21	5	$f(x_1, x_2, x_3, x_4, x_5) = 2 - 2.1\cos(9.8x_1)\sin(1.3x_5)$

Table 6.4: Symbolic regression problems used for benchmarking EGGP. These problems are taken from [169].

and each approach has access to the 18 constants: $-0.9, -0.8, \ldots, -0.1, 0.1, 0.2, \ldots, 0.9$. In TGP these are constants, whereas in the EGGP variants and CGP, they are further input nodes.

We evaluate all individuals using the Mean Square Error (MSE) fitness function. For a given set of inputs X, candidate f and target values Y, the MSE is computed as

$$MSE(f, X, Y) = \frac{\Sigma_{i=0}^{|X|}(Y_i - f(X_i))^2}{|X|}. \tag{6.4}$$

We measure statistics taken over 100 independent runs of each approach on each dataset. For EGGP, we use a fixed 100 nodes and a mutation rate $m_r = 0.03$.

For CGP, we use the experimental parameters in [253], [155, Ch.3], at which values CGP outperforms TGP on symbolic regression problems. We use 100 fixed nodes, and a mutation rate of 0.03. We use the $1 + \lambda$ EA with $\lambda = 4$. We do not use any of the published CGP crossover operators, as their usefulness, particularly on symbolic regression problems, remains disputed [106], and [155, 243] recommend the $1 + \lambda$ approach.

For TGP, we use the experimental parameters in [169] with a minor adjustment. The population size is 500, with 1 elite individual surviving in each generation. Subtree crossover is used with a probability of 0.9, and when it is not used, the 'depth steady' subtree replacement mutation operator is used, which, when replacing a subtree of depth d generates a new subtree of depth between 0 and d [169]. Tournament selection is used to select reproducing individuals, with a tournament size of 4, and the maximum depth allowed of any individual is 10. Unusually for TGP, we add each new individual to the population one-by-one, discarding one of the children produced by each crossover operator. This allows us to immediately replace invalid (with respect to maximum depth) individuals, guaranteeing that every individual in a new population is valid and should be evaluated. To initialise the population, we use the ramped half-and-half technique [129], with a minimum depth of 1 and a maximum depth of 5.

For all experiments, the maximum number of evaluations allowed is 24 950, a value taken from [169] (50 generations with a population size of 500 and 1 elite individual that does not require re-evaluating). In TGP this is achieved by allowing the search to run for 50 generations. In EGGP and CGP, we use the optimisation from [155, Ch.2], where individuals are evaluated only when their active components are mutated; there is no fixed number of mutations, and the search continues until the total number of evaluations is performed. There is no analogous optimisation for TGP, as TGP individuals contain no neutral material. This optimisation makes a large difference to the depth of search; for example, in CGP running on F_1, the median number of generations is 12 385, but if all individuals are evaluated (rather than only those with active region mutations), the number of generations would be capped at 6237 (assuming elite individuals are never re-evaluated).

Our CGP experiments are based on the publicly available CGP library [243] with modifications made to accommodate the 'active evaluations only' optimisation and the use of validation and training sets. Our TGP experiments are based on the Distributed Evolutionary Algorithms in Python (DEAP) evolutionary computation framework [70] with modifications made to accommodate our crossover strategy, mutation operator, and use of validation and training sets.

	EGGP		TGP		CGP	
F	MF	IQR	MF	IQR	MF	IQR
F1	**4.45E-3**	7.35E-3	5.77E-3	3.40E-3	6.74E-3	4.30E-3
F2	**8.17E6**	6.05E6	1.28E7	7.86E6	1.73E7	2.54E6
F3	1.18E-2	7.34E-3	**1.04E-2**	3.56E-3	1.48E-2	4.39E-3
F4	2.58E13	1.05E9	3.55E13	8.35E13	**2.58E13**	2.35E9
F5	**3.96E0**	3.56E0	5.13E0	3.81E0	7.17E0	1.47E0
F6	1.69E1	2.24E1	**2.61E0**	6.86E0	9.28E0	2.03E1
F7	**3.06E2**	7.40E2	4.20E2	3.50E2	5.76E2	4.39E2
F8	**3.91E-2**	7.43E-2	1.09E-1	4.99E-2	4.49E-2	9.59E-2
F9	7.09E2	5.40E3	**1.46E2**	3.04E1	1.71E2	1.11E3
F10	**1.52E-1**	2.05E-1	3.22E-1	5.62E-2	1.66E-1	1.42E-1
F11	3.93E1	7.26E1	**3.88E1**	3.37E1	4.96E1	4.73E1
F12	1.21E3	5.25E2	1.25E3	5.02E1	**7.08E2**	5.19E2
F18	4.07E4	9.27E3	4.13E4	3.54E2	**1.20E2**	4.10E4
F21	1.07E0	6.16E-4	1.07E0	4.90E-4	**1.07E0**	1.53E-5

Table 6.5: Results from symbolic regression benchmarks as described in Section 6.6. MF indicates the **M**edian **F**itness over observed runs; the lowest (best) MF result across all algorithms is highlighted in **bold**. IQR indicates the Inter-quartile range in fitness.

6.7 Symbolic Regression Results

The results of our symbolic regression experiments are given in Table 6.5. For each of the three approaches studied, we give the Median Fitness (MF) recorded across 100 runs, and the IQR in observed test fitness. We select 3 cases (F2, F6, F18) where we observe statistical differences and present our results as box-plots in Figure 6.2.

To test for statistical significance we use the two-tailed Mann–Whitney U test and perform a Bonferroni procedure for each hypothesis giving a corrected significance threshold of $\alpha = \frac{0.05}{14}$. In the case where we get a statistically significant result ($p < \frac{0.05}{14}$), we also calculate the effect size, using the Vargha–Delaney A Test. This results in pair-wise p and A values for each problem and pair of algorithms, given in Table 6.6.

The first point of interest is that there was no universal 'winner' across the studied prob-

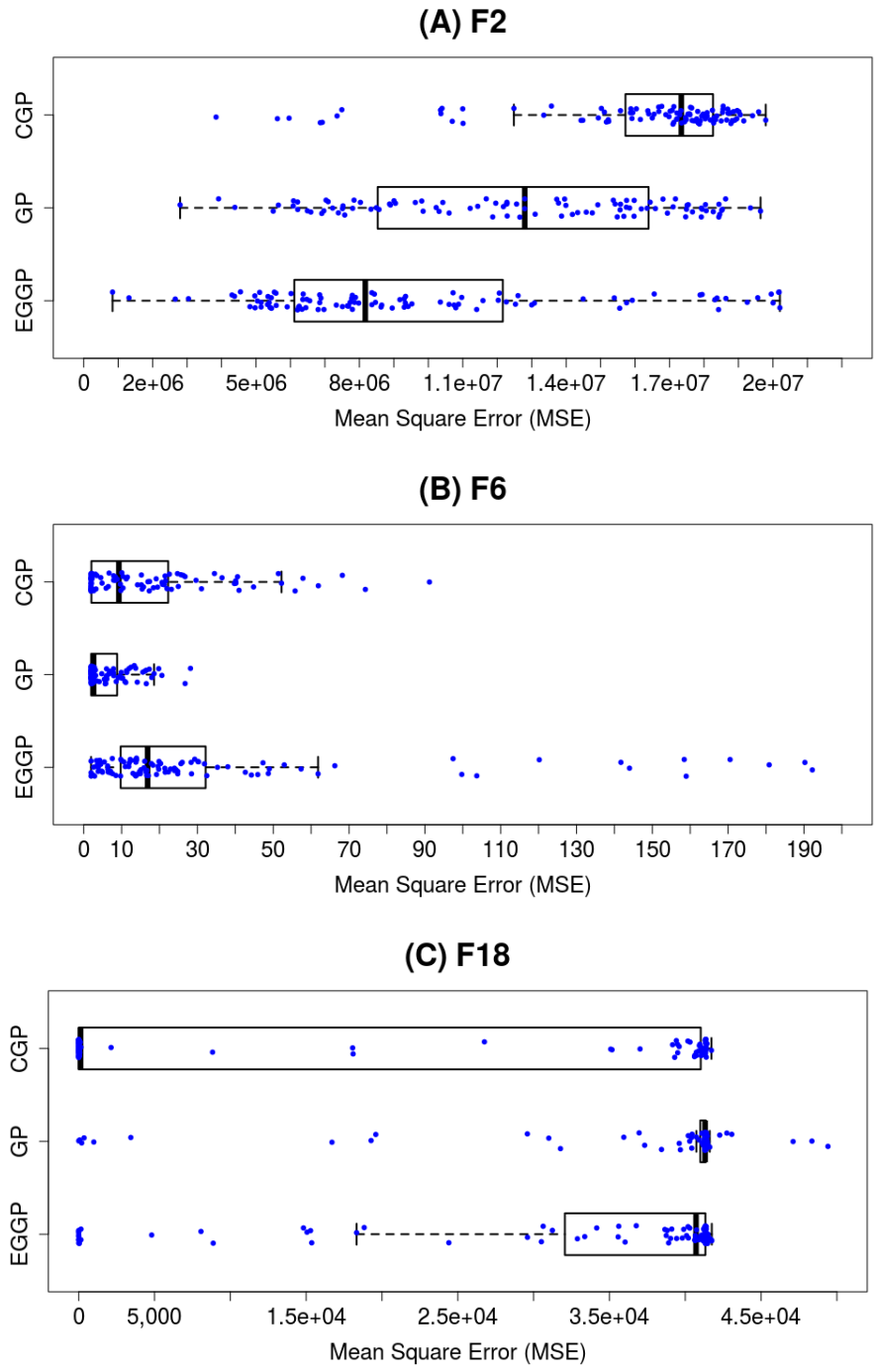


Figure 6.2: Box-plots with data overlayed for the following symbolic regression problems; (A) F2, (B) F6, (C) F18. Overlayed data is jittered for visual clarity.

	EGGP vs. TGP		**EGGP vs. CGP**		**TGP vs. CGP**	
F	p	A	p	A	p	A
F1	0.08	-	0.02	-	0.25	-
F2	$< \alpha$	0.68	$< \alpha$	**0.82**	$< \alpha$	**0.75**
F3	0.22	-	$< \alpha$	0.70	$< \alpha$	**0.79**
F4	$< \alpha$	0.67	0.29	-	$< \alpha$	0.66
F5	0.09	-	$< \alpha$	**0.86**	$< \alpha$	**0.84**
F6	$< \alpha$	**0.85**	$< \alpha$	0.66	$< \alpha$	0.65
F7	0.02	-	$< \alpha$	0.64	0.01	-
F8	$< \alpha$	**0.77**	0.54	-	$< \alpha$	**0.73**
F9	$< \alpha$	**0.75**	$< \alpha$	0.64	$\geq \alpha$	-
F10	$< \alpha$	**0.86**	0.99	-	$< \alpha$	**0.93**
F11	0.63	-	0.49	-	0.05	-
F12	0.09	-	$< \alpha$	**0.75**	$< \alpha$	**0.85**
F18	$< \alpha$	0.64	$< \alpha$	0.70	$< \alpha$	**0.81**
F21	0.75	-	$< \alpha$	0.66	$< \alpha$	0.70

Table 6.6: Statistical tests comparing the observed distributions associated with Table 6.5 for EGGP, TGP and CGP. The p value is from the two-tailed Mann–Whitney U test. The corrected threshold for statistical significance is $\alpha = \frac{0.05}{14}$. Where $p < \alpha$, the effect size from the Vargha–Delaney A test is shown; large effect sizes (A ¿ 0.71) are shown in bold. Where $\alpha \leq p < 0.005$, p is listed as $\geq \alpha$.

lems, although in many (12/14) cases, EGGP performed either best or second best with respect to the reported MF. On 6 problems, EGGP reported the lowest (best) MF, whereas TGP and CGP reported the lowest in 4 problems each. In 6 of the 8 problems where EGGP did not report the lowest MF, it instead reported the second lowest MF. It is also clear from Table 6.6 that in many cases where EGGP *appears* to outperform another approach with respect to MF, we do not observe any statistical significance. This 'middle-of-the-pack' behaviour of EGGP is reflected in the statistical values given in Table 6.6; in 12 of the 28 statistical tests involving EGGP, we find no statistical differences.

Comparing EGGP directly to TGP we observe 2 problems (F8, F10) where EGGP achieved a statistically significant lower MF with a large effect size. On 3 further problems (F2, F4, F18), EGGP did better with statistical significance, but without large effect. Conversely,

TGP achieved a statistically significant lower MF with large effect on 2 problems (F6, F9). On all 7 other problems (F1, F3, F5, F7, F11, F12, F21) we observed no statistical differences. So as an emergent trend we find that on most (10) problems, there are either insignificant differences between EGGP and TGP, or statistically significant differences without large effect.

Comparing EGGP directly to CGP, we observe 2 problems (F2, F5) where EGGP achieved a statistically significant lower MF with a large effect size. On 2 further problems (F3, F7), EGGP did better with statistical significance, but without large effect. Conversely, CGP achieved a statistically significant lower MF with large effect on 1 problem (F12). On 4 further problems (F6, F9, F18, F21) CGP did better with statistical significance, but without large effect. On all 5 other problems (F1, F4, F8, F10, F11) we observed no statistical differences. Again, we find that on most (11) problems there are either insignificant differences between EGGP and CGP, or statistically significant differences without large effect.

The pattern we see in both cases of few results with statistical significance and large effect stands in contrast the comparison between CGP and GP. Here we see 3 cases where TGP achieved a statistically significant lower median fitness than CGP, with large effect, and 4 cases where the converse holds. So on exactly half of the problems studied, there is a statistically significant difference with large effect.

6.8 General Discussion

The results from our symbolic regression benchmarking are remarkably different from our experiments comparing EGGP and CGP on digital circuit benchmark problems. In the symbolic regression experiments, we saw few statistical differences with large effect between EGGP and either TGP or CGP. In the digital circuit experiments, we saw EGGP outperform CGP on many hard problems with statistical difference and large effect. In this section we set out some plausible explanations for this.

Firstly, the two benchmark domains had different objectives. In the digital circuit problems, the objective was to find a globally optimal solution and the performance was measured by the (median) effort required to achieve this. In the symbolic regression problems, the objective was to find a high quality solution within a fixed budget of fitness evaluations. One plausible explanation, then, is that EGGP is better suited to rapidly converging on a global solution than it is to finding approximate solutions in a (relatively) short computational budget.

Secondly, the two domains utilised different function sets, with different internal interactions. As mentioned in Section 6.5, it is possible for EGGP solutions to undergo phenotypic growth and shrinkage[3] for example when exploiting functions with easily expressed identities (such as AND(x, x)) in ways that are impossible under the constraints of the CGP genotype. It is plausible that this is beneficial to the evolutionary process as a form of phenotypic neutral drift. If easily created/destroyed intronic code is less available in the symbolic regression function set, then it could follow that the theoretical benefits of the generalised landscape of EGGP have less impact on the performance of the evolutionary search.

One further possibility arises from the question of bloat. Bloat is a commonly observed [2] and theoretically studied process in TGP where evolved programs grow through time, often hampering the evolutionary process. The other graph-based Evolutionary Algorithm (EA) we have studied, CGP, is known to have inherent anti-bloat properties [240] in that solutions remain small even when the number of nodes in the representation is increased. One possible explanation as to why EGGP was outperformed by CGP on some symbolic regression problems is that the generalised landscape offered by EGGP does not exhibit this property. If this explanation holds, then this has interesting consequences for the study of anti-bloat in CGP, the cause of which remains an open question. Further, this explanation interconnects with our previous comments with respect to the function set; it may be the case that the digital circuit function set is inherently robust against bloating.

6.9 Conclusions and Future Work

In this chapter we have carried out extensive experiments demonstrating that EGGP can be used to effectively synthesise digital circuits and symbolic expressions. We have studied the performance of EGGP on 16 digital circuit benchmark problems, and compared our approach to CGP under very similar experimental conditions. In 12 of the studied problems we observed EGGP requiring less evaluations (measured by median) than CGP with statistical significance. In 11 of these problems, we found that the difference had large effect. Further, the 4 problems where we saw no statistical differences are clearly the easiest of the studied problems, both conceptually and with respect to the effort required to solve them. As the problems increased in difficulty, we saw an increasing difference between the two studied algorithms. This is perhaps most strongly highlighted by the box-plots in Figure 6.1, where, in the case of the hardest problem (3-Mul) we see the box-plot of EGGP results outside the

[3] Anecdotally, this appears to happen very often throughout a digital circuit evolutionary run!

outliers for CGP results.

We have also performed further clarifying experiments through the use of an ordered variant of EGGP, O-EGGP, which enforces an ordering on nodes which must be respected when modifying edges. When comparing O-EGGP to CGP under standard conditions, we observed no statistical differences. This provides robust evidence to our claim in Section 5.6.2 that the landscape used in EGGP is distinct from that used in CGP. Further, this supports the hypothesis that EGGP outperforms CGP on the digital circuit benchmark problems as a result of the generalised landscape that we have described.

We have studied the performance of EGGP on 14 symbolic regression problems, and compared our approach to CGP and TGP. In these problems, we saw less statistical differences between EGGP and the compared approaches. On 6 problems, EGGP found the highest quality solution (measured by median), and on 6 further problems, EGGP found the second highest quality solution. However, through statistical tests we observe that many of our comparisons are not statistically significant, and of those that are, few have large effect. Statistical comparisons between TGP and CGP are much more discriminating, with half of the problems seeing statistical significance and large effect. We have therefore set out a number of plausible explanations for our observations, and the distinctions between the digital circuit problems and the symbolic regression problems.

Clearly, it would be beneficial to perform more experiments with EGGP to study the effect of its various parameters. For example, it would be possible to vary the size of the representation, the mutation rate, or the choice of the λ parameter. Such experiments would be particularly interesting in comparison to previous work modifying the parameters of CGP. For example, it is known that CGP can perform better when the representation size is massively increased [156], so it would be interesting to examine whether the same effect is observed with EGGP, particularly in light of our discussion of bloat in Section 6.8.

Additional experiments investigating the hypotheses we set out in Section 6.8 would shed light on the discrepancies in our observations on digital circuit and symbolic regression problems. For example, the explanation that the differences arise from the different objectives could be tested by investigating the performance of EGGP on both sets of problems in both settings. The notion of possible bloat in EGGP could be investigated through the study of anti-bloat techniques such as destructive mutation operators applied larger solutions [122].

There are a number of other domains where we could benchmark the effectiveness of EGGP in comparison to other GP approaches with minimal modification to the algorithm. Studying

these problems would shed further light onto the similarities and differences between the ideas we set out in this thesis and other approaches from the literature. Examples of appropriate problems that have been studied in the literature include:

1. Approximate circuits, as in [165,249,250], by introducing a multi-objective evolutionary algorithm such as SPEA2 [268] as a replacement of the $1 + \lambda$ algorithm.
2. Cryptographic circuits, as in [179, 180].
3. Image processing, as in [88, 89, 201].
4. Multi-step forecasting, as in [58].
5. A number of other plausible problems are discussed in [253], including the lawnmower problem and the hierarchical if-and-only-if problem.

Further, if EGGP were extended to accommodate for Typed Function Graphs (TFGs), as discussed in Section 5.7, then clearly it would be desirable to evaluate this typed extension in the context of problems over typed function sets. A number of examples of such problems may be found in [99]. Similarly, if EGGP were extended to accommodate for Hierarchical Function Graphs (HFGs), as discussed in Section 5.7, then there would be a clear desire to evaluate such an approach. The research on Embedded CGP (ECGP) found in [253] sets some precedents for empirical evaluation of such an approach.

7 Evolving Recurrent Graphs by Graph Programming

Abstract

In this chapter, we extend Evolving Graphs by Graph Programming (EGGP) to handle Recurrent Function Graphs (RFGs). This extension, termed Evolving Recurrent Graphs by Graph Programming (R-EGGP), maintains an acyclic subgraph induced by non-recurrent edges while allowing recurrent edges to connect freely throughout the graph. In this chapter we propose suitable genetic operators for evolving RFGs. We give an initialisation procedure that generates RFGs which can be parameterised with a probability p_{rec} of a recurrent edge occurring. We give two mutation operators, one for mutating non-recurrent edges while maintaining the acyclicity of the subgraph induced by those edges, and the other for mutating recurrent edges. We evaluate R-EGGP on a variety of benchmarking problems; digital counters, mathematical sequences and generalising n-bit parity digital circuits. On many problems studied, we find statistically significant improvements over Recurrent Cartesian Genetic Programming (RCGP).

7.1 Introduction

So far we have seen how genetic operators, implemented in P-GP 2, can be used to effectively evolve Acyclic Function Graphs (AFGs) through the Evolutionary Algorithm (EA), Evolving Graphs by Graph Programming (EGGP). In this chapter, we discuss an extension of EGGP, named 'Evolving Recurrent Graphs by Graph Programming' (R-EGGP), to handle potentially Recurrent Function Graphs (RFGs). This allows us to describe evolution over *stateful* programs which make use of recurrent edges. The results of this chapter will be used in Chapter 9's study of neuroevolution.

This is not the first work to consider evolution of stateful programs. Indeed, one might consider the various evolved automata we have discussed in Chapter 2 to be examples of evolution of stateful programs. Examples of this include Graph Structured Program Evolution (GRAPE) [206], Genetic Network Programming (GNP) [118] or Parallel Algorithm Discovery and Orchestration (PADO) [231, 232]. Similarly, one might consider the many works that effectively evolve recurrent Artificial Neural Networks (ANNs) [117, 195, 222] to be examples of such a domain. However, these evolutionary techniques are specific to domains.

Various works have introduced recursive functions to conventional Genetic Programming (GP) constructs. Some works allow evolved programs to call themselves, thereby inducing recursive programs [28,262]. Others integrate recursive functions into the function set to gain access to recursive logic [1,40,170]. In [105] machine code is executed with access to a program counter, thereby allowing recursive behaviour to emerge. Self modifying machine code was used in [172], which may then induce recursive behaviour. In some literature [228, 233], the problems of inducing explicit recursion and guaranteeing terminating behaviour are resolved by factoring out the recursive patterns of the given problem. For a more complete survey of these approaches, see [1]. However, we distinguish between the stateful *recurrent* programs that we study here and these GP techniques that have explicit extensions to support *recursive* behaviour. This is motivated by a desire to learn stateful programs without needing to design recursive functions for the function set. Designing such functions may be problematic when working in domains such as synthesising direct implementations of digital circuits.

Recurrent Cartesian Genetic Programming (RCGP) is a particularly relevant technique [241,242,245]. In RCGP, Cartesian Genetic Programming (CGP) is extended to accommodate for recurrent edges which may target any node within the genotype. When a RCGP graph is evaluated, its nodes are evaluated moving 'left to right' from inputs to outputs. In doing this, edges which connect to nodes later in the network gain access to those nodes' previous

computed state, therefore providing these individuals with a form of memory similar to that of Function Graphs (FGs) as set out in Chapter 4. When an edge is mutated in RCGP, with probability p_{rec} it may be directed to point anywhere, whereas with probability $1 - p_{rec}$ it is directed to point to a node earlier in the ordering. As the reader will see, these basic ideas will form the inspiration of this chapter where the distinctions between our proposed work and RCGP will be discussed later in it.

In this chapter, we discriminate between non-recurrent and recurrent edges, and thereby gain access to a rich set of behaviours. For example, it becomes possible for such a program to compose many functions together over its previous state, by first accessing the previous state with recurrent edges, and then computing over this space with non-recurrent edges. However, the evolution of these RFGs is less straightforward than with the AFGs we have seen in Chapters 5 and 6. For an RFG to have well-defined semantics, it remains the case that the subgraph induced by non-recurrent edges must be acyclic. It is the recurrent edges in the RFG that may form cycles, either through non-recurrent edges or recurrent edges.

We therefore design genetic operators capable of handling both the preservation of acylicity over non-recurrent edges, and the possibility of cycles over recurrent edges. It is this resultant system that we refer to as R-EGGP. To evaluate our proposed approach, we draw empirical comparisons with RCGP. We find that R-EGGP generally outperforms RCGP when synthesising various digital counters, mathematical sequences and n-bit parity checks that generalise.

This chapter is arranged as follows. In Section 7.2 we describe a P-GP 2 program for initialising RFGs which can be parameterised with recurrent edge probability p_{rec}. In Section 7.3 we give two mutation operators which allow for appropriate manipulation of RFGs. We compare our approach to RCGP in Section 7.4. We describe experiments for synthesising digital counters in Section 7.5 and present results from those experiments in Section 7.6. We describe experiments for synthesising famous mathematical sequences in Section 7.7 and present results from those experiments in Section 7.8. Section 7.9 describes our final experiments, where we synthesise n-bit parity checks that generalise, with results presented in Section 7.10. Finally, we conclude our findings in Section 7.11.

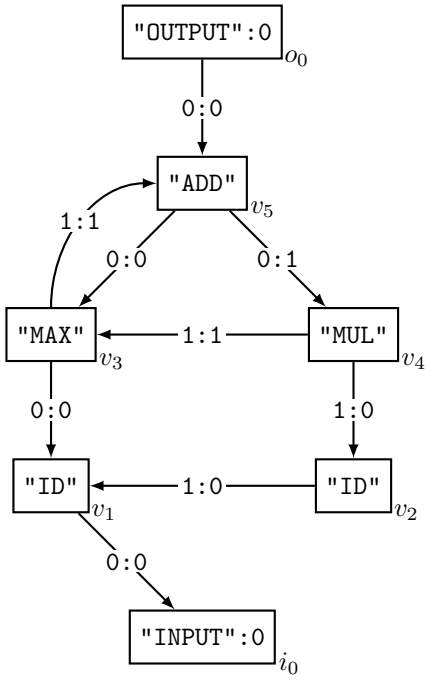


Figure 7.1: A simple RFG generating the Fibonacci sequence that has weights and biases that are effectively ignored. The information with respect to edge weight and node bias are therefore not shown.

7.2 Initialisation

As we did with AFGs in Section 5.2, in this section we set out a simple initialisation procedure given as a pair `Init`, S where `Init` is a P-GP 2 program which, when applied to initial graph S, generates RFGs suitable to the target problem.

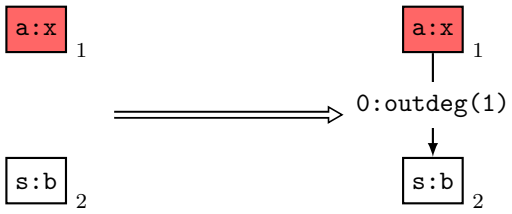
In this chapter we deal with a simplified set of RFGs. The RFGs handled here have the following properties:

1. All edge weights and function node biases are assumed to be equal to 1 and are effectively ignored during execution.
2. All functions are assumed to ignore the values coming in from the bias of each function node.
3. Edges from output nodes are non-recurrent.

We return to the Fibonacci sequence generating RFG shown in Figure 7.1, which we dis-

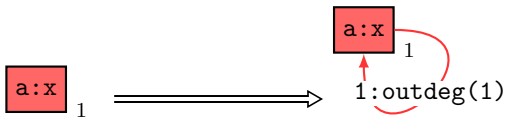
Main := ([add_node_f_x | $f_x \in F$]; [connect_node, connect_node_rec]!; unmark_node!)!;
(pick_loop; [pick_target]; {expand_loop, expand_edge})!;
[connect_output]!; remove_counter

connect_node(a,b:list;
s:string; x:int)[1-p_{rec}]



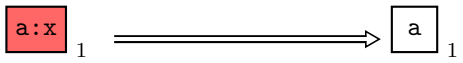
where s != "OUTPUT" and outdeg(1) < x

connect_node_rec(a:list; x:int)[p_{rec}]

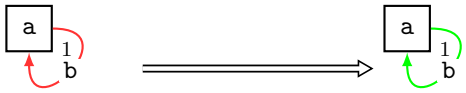


where outdeg(1) < x

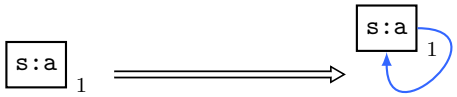
unmark_node(a:list; x:int)



pick_loop(a,b:list)

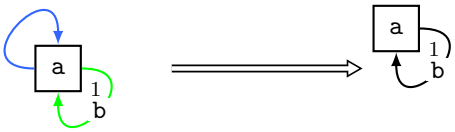


pick_target(a:list; s:string)

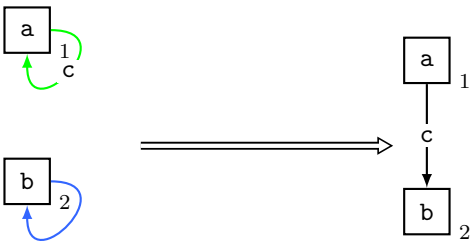


where s != "OUTPUT"

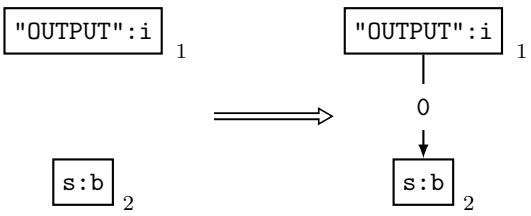
expand_loop(a,b:list)



expand_edge(a,b,c:list)



connect_output(a,b:list;
s:string; i:int)



where outdeg(1) < 1

remove_counter(a:list)

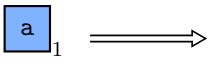


Figure 7.2: A program for generating our RFGs. This program is also parameterised by probability p_{rec}, which determines the likelihood of creating recurrent edges.

cussed in Section 4.2.3. Each edge label is a pair of integers `a:b` where `a` determines whether the edge is recurrent (`a` = 1) or non-recurrent (`a` = 0) and `b` is the ordering of the edge. As a reminder of how such RFGs are executed, we give the trace execution of this RFG when driven with the input 1:

Time	i_0	v_1	v_2	v_3	v_4	v_5	o_0	*Description*
-1	–	0	0	0	0	0	–	*Initial state.*
0	1	1	0	1	0	1	1	$Fib(0)$
1	1	1	1	1	0	1	1	$Fib(1)$
2	1	1	1	1	1	2	2	$Fib(2)$
3	1	1	1	2	1	3	3	$Fib(3)$
4	1	1	1	3	2	5	5	$Fib(4)$
5	1	1	1	5	3	8	8	$Fib(5)$
6	1	1	1	8	5	13	13	$Fib(6)$
					...			

(7.1)

Figure 7.2 shows our initialisation program, which is parameterised with probability p_{rec} which determines the rate at which recurrent edges are added. This program generally functions in the same manner as the initialisation procedure of EGGP. However, we have replaced the probabilistic rule call

```
[connect_node],
```

with the probabilistic rule-set call

```
[connect_node, connect_node_rec].
```

The weight associated with `connect_node` is $1 - p_{rec}$; this rule functions as normal and simply gradually grows an acyclic subgraph of non-recurrent edges. The weight associated with `connect_node_rec` is p_{rec}; this rule stores recurrent edges as `red` marked loops which are expanded after all function nodes have been created. Once all nodes have been added, these `red` marked loops are expanded. In the next loop, the rule `pick_loop` is called, identifying one of these loops and marking it `green`. This rule ensures that this loop terminates; once all `red` marked loops have been expanded, it will have no matches. We pick a target uniformly at random across all non-input nodes with the rule `pick_target`, and uniquely identify that target node by inserting a `blue` loop. Then either the rule `expand_loop` removes the `green` marking from the stored edge and deletes the `blue` loop (if both are attached to the same

node), or the rule `expand_edge` deletes both loops and creates an edge from the source of the stored loop to the chosen target, which is labelled with the same label as the stored loop. These rules effectively expand the stored edge to point to the chosen target, regardless of whether that target is the same node as the source of the stored recurrent edge.

Overall, the program's semantics can be seen to add randomly chosen function nodes until the number of function nodes equals the amount specified by the input graph. Each function node is randomly connected to previously added function nodes and input nodes by non-recurrent edges with probability $1 - p_{rec}$, or acquires stored recurrent edges with probability p_{rec}. Once all nodes have been added, the stored recurrent edges are expanded to point to random, non-output, nodes in the graph. Then, output nodes are connected at random to the rest of the graph via non-recurrent edges. Finally, the node specifying the number of function nodes to add is removed. As with EGGP, we expect the input graph to this program to consist of the following:

1. For each input associated with the problem, there exists an input node.
2. For each output associated with the problem, there exists an output node.
3. A node, marked `blue` and labelled `"NODES":x` where `x` $\in \mathbb{N}_0$ specifies the number of function nodes to add.

7.3 Mutation

To mutate our RFGs, we modify the existing EGGP mutation operators. Conveniently, the function sets we use in our R-EGGP experiments consist only of nodes with arity 2. For the purposes of this chapter, we reuse the node mutation operator described in Section 5.3.2 as the fixed arity ensures that node mutation will not modify the structure. However, it would be straightforward to extend this operator to support creating both recurrent and non-recurrent edges by following the general ideas we set out here.

In this section, we present 2 edge mutation operators; one mutates an edge to a non-recurrent edge preserving acyclicity (Section 7.3.1), the other mutates an edge to a recurrent edge that is indifferent to acyclicity (Section 7.3.2). In principle, these can be combined into a single operator with a probabilistic rule-set acting as a control construct. However, it may be more convenient for the reader to see them as separate mutation operators, one called with probability $1 - p_{rec}$ and the other called with probability p_{rec}.

```
Main := try ([[pick_edge, pick_loop]]; {mark_output_1, mark_output_2}!;
            [mutate_edge, mutate_loop]; unmark!)
```

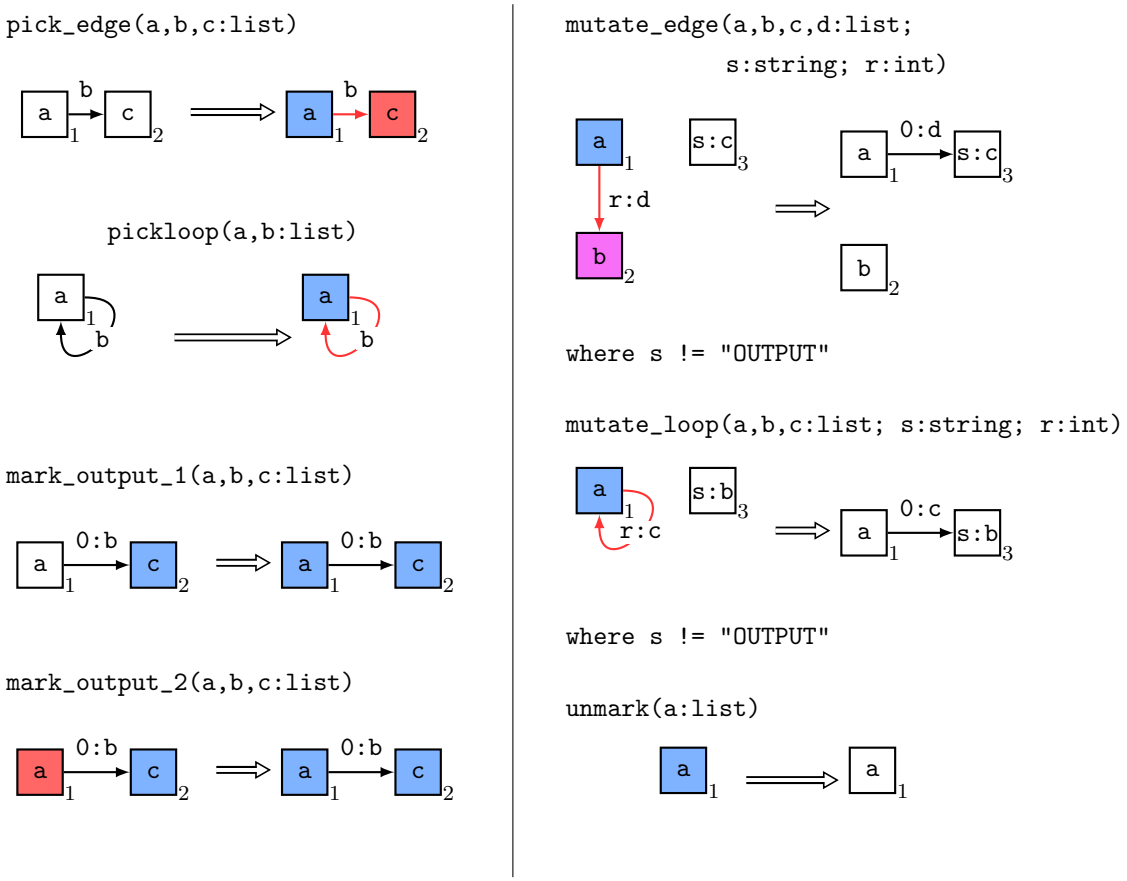
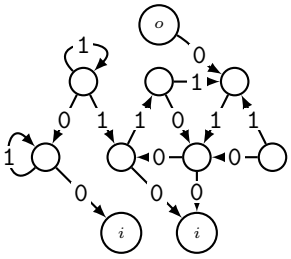
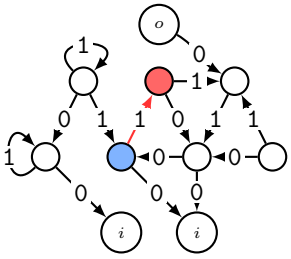


Figure 7.3: A program for mutating arbitrary edges in an RFG while preserving acyclicity of the underyling subgraph induced by non-recurrent edges. The mutated edge becomes non-recurrent and may target any (non-output) node.

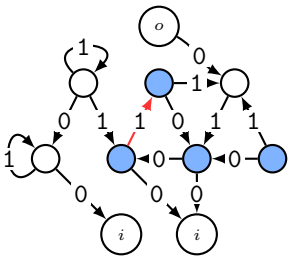
This individual is to undergo an edge mutation that mutates an edge to be a non-recurrent edge and preserves acyclicity of the subgraph induced by non-recurrent edge.



(1) `[[pick_edge, pick_loop]]`:
An edge e to mutate is chosen at random and marked (red) alongside its source node s (blue) and target node t (red).



(2) `{mark_output_1, mark_output_2}!`:
Invalid candidate nodes for redirection are identified. If a node v has a directed path of non-recurrent edges to s it is marked blue, as targeting it would introduce a cycle of non-recurrent edges. In this case, this includes target node t.



(3) `[mutate_edge, mutate_loop]; unmark!`:
The edge e is mutated to be a non-recurrent edge targeting some randomly chosen unmarked (non-output) node, preserving acyclicity of the subgraph induced by non-recurrent edges. The new target has been marked with a star '$\star$' for visual clarity. Finally, all marks are removed.

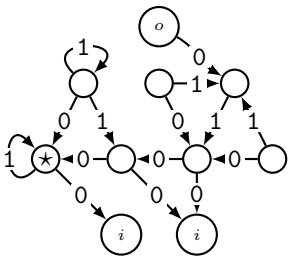


Figure 7.4: A trace of the application of the non-recurrent edge mutation program in Figure 7.3. For visual simplicity, node labels have been omitted. Additionally, we are omitting the ordering index component of edge labels; the integers shown correspond to whether or not an edge is recurrent.

7.3.1 Non-Recurrent Edge Mutation

We present a mutation operator which modifies our RFGs by picking uniformly at random *any* edge and then mutating it so that it becomes a non-recurrent edge, while also preserving the acyclicity of the subgraph induced by non-recurrent edges. This mutation operator is given in Figure 7.3. This mutation operator is almost identical to that described in Section 5.3.1, except it may also match loops and that when inducing the set of all nodes with paths to the source of the mutating edge, it only considers paths consisting of non-recurrent edges.

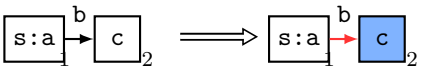
When an edge mutation is called, this program is applied with probability $1 - p_{rec}$. Recall from Chapter 3 that the double square bracket syntax "`[[r_1, ..., r_k]]`" means 'apply this rule-set with a uniform distribution across all matches for all rules'. Then it is clear that the initial rule-set call of the program uniformly chooses an edge to mutate at random, marking that edge `red`. The source of the edge is marked `blue` and, unless the chosen edge is a loop, the target of the edge is marked `red`. The call to rules `mark_output_1` and `mark_output_2` then marks every node for which there exists a path of *non-recurrent* edges from that node to the source of the chosen edge. The `mark_output_2` rule accounts for the fact that the chosen edge, in its current state, may be recurrent. Following from the logic of Section 5.3.1, we know that we can mutate this edge to point to any unmarked node and this will not introduce a cycle of non-recurrent edges. Finally, the probabilistic call to rules `mutate_edge` and `mutate_loop` redirect the mutating edge to some new non-output target node chosen uniformly at random, and the rule `unmark` returns the graph to an unmarked state.

Hence we have a mutation operator that chooses any edge to mutate uniformly at random, and redirects it so that it is a non-recurrent edge and that the graph does not contain any cycles of non-recurrent edges.

We give an example execution of our mutation operator in Figure 7.4. For visual simplicity, we do not show ordering relations, and instead only show recurrence relations on edges, hence each edge is labelled with only 1 integer. Here, a recurrent edge is chosen by the rule `pick_edge`. The application of the `mark_output` rules as long as possible marks all nodes for which there is a path to the source of the chosen edge, including the target of the chosen edge. Then the `mutate_edge` rule is applied, deleting the mutating edge and creating a new, non-recurrent edge while preserving acyclicity of the subgraph induced by non-recurrent edges.

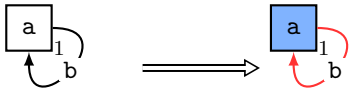
```
Main := try ([[pick_edge, pick_loop]]; [pick_target];
            {mutate_edge_edge, mutate_edge_loop, mutate_loop_edge})
```

```
pick_edge(a,b,c:list; s:string)
```

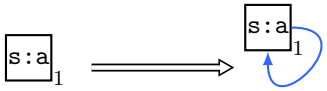



```
where s != "OUTPUT"
```

```
pickloop(a,b:list)
```

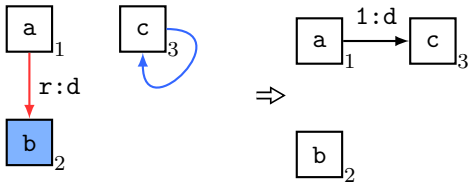



```
pick_target(a:list; s:string)
```

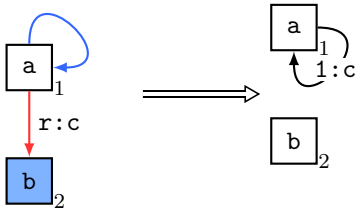



```
where s != "OUTPUT"
```

```
mutate_edge_edge(a,b,c,d:list; r:int)
```




```
mutate_edge_loop(a,b,c:list; r:int)
```




```
mutate_loop_loop(a,b,c:list; r:int)
```

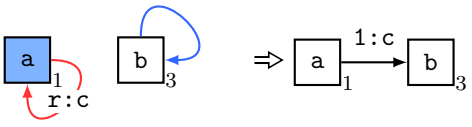
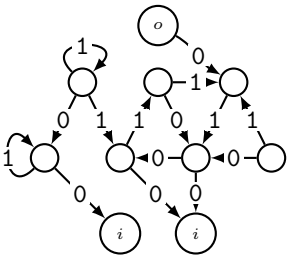
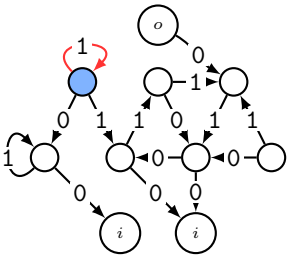


Figure 7.5: A program for mutating arbitrary edges in an RFG. The mutated edge becomes recurrent and may target any (non-output) node.

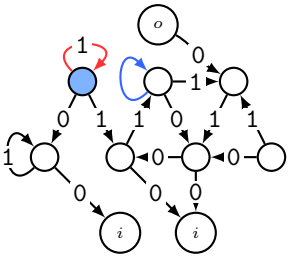
This individual is to undergo an edge mutation that mutates an edge to be a recurrent edge.



(1) `[[pick_edge, pick_loop]]`:
An edge e to mutate is chosen at random and marked `red` and its target t is marked `blue`. In this case, the chosen edge is a loop.



(2) `[pick_target]`:
A new target node t' is chosen uniformly at random from the nodes which are not t and not outputs. A `blue` marked loop is added to t'.



(3) {`mutate_edge_edge, mutate_edge_loop mutate_loop_edge`}:
The edge e is mutated to be a recurrent edge targeting t'. The new target has been marked with a star '$\star$' for visual clarity. Finally, all marks are removed.

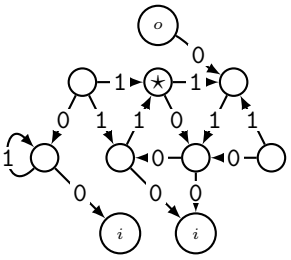


Figure 7.6: A trace of the application of the recurrent edge mutation program in Figure 7.5. For visual simplicity, node labels have been omitted. Additionally, we are omitting the ordering index component of edge labels; the integers shown correspond to whether or not an edge is recurrent.

7.3.2 Recurrent Edge Mutation

We present a mutation operator which modifies our RFGs by uniformly picking *any* edge and then mutating it so that it becomes a recurrent edge. This transformation is indifferent to acyclicity of the graph. Our mutation operator is given in Figure 7.5. While it may require more rules, this mutation operator is conceptually simpler than the one discussed in Section 7.3.1; the additional rules handle permutations of loops and edges.

When an edge mutation is called, this program is applied with probability p_{rec}. Again, recall from Chapter 3 that the double square bracket syntax "`[[r_1, ..., r_k]]`" means 'apply this rule-set with a uniform distribution across all matches for all rules'. Then the initial probabilistic rule-set call of the program picks any edge or loop uniformly at random, marking the edge `red` and its target `blue`. There is one exception to allowed matches; the source must not be an output, as seen in the condition of `pick_edge`, as we have previously stated that we do not want out outputs to have recurrent edges. However, this can be relaxed if desired. After an edge to mutate has been chosen, the rule `pick_edge` is probabilistically applied, picking some non-output non-`blue` node, and adding a `blue` unlabelled loop to that node. This rule chooses the new target of our mutating edge, making exceptions for output nodes and the current target of the mutating edge. Note that it can in principle choose the source of the mutating edge, as long as the mutating edge is not a loop. The three rules, `mutate_edge_edge`, `mutate_edge_loop` and `mutate_loop_edge`, describe the three permutations of mutations now possible. By applying one of these rules, we redirect the mutating edge to target the chosen target, and the mutating edge becomes a recurrent edge.

Hence we have a mutation operator that picks an edge uniformly at random, as long as the source of that edge is not an output node. This edge is redirected so that it is a recurrent edge that does not target its previous target node or an output node.

We give an example execution of our mutation operator in Figure 7.6. For visual simplicity, we do not show ordering relations, and instead only show recurrence relations on edges, hence each edge is labelled with only 1 integer. Here, a recurrent loop is chosen by the rule `pick_loop`. The application of `pick_target` chooses a new target for the loop. Then the `mutate_loop_edge` rule is applied, deleting the mutating edge and creating a new, recurrent edge targeting the chosen target.

7.4 Comparison with Recurrent Cartesian Genetic Programming

In this section we compare our approach to RCGP. We claim that R-EGGP strictly generalises the landscape of RCGP, just as we claimed that EGGP strictly generalises the landscape of standard CGP in Section 5.6.1. From Section 5.6.1 we know that the non-recurrent edge mutations we use here are strict generalisations of the non-recurrent edge mutations used in RCGP. However, as recurrent edge mutations allow edges to target any node in the graph, it may be the case that the neighbourhoods lost by order-preserving mutations are still available via recurrent edge mutations that happen to preserve acyclicity. It is also clear that any recurrent mutation in RCGP may be replicated with a recurrent mutation in R-EGGP.

We recall the execution of a RCGP individual from [241]:

1. Set all active nodes to output zero.
2. Apply the next set of inputs.
3. Update all active nodes once from inputs to outputs and read the outputs.
4. Repeat from 2 until all input sets have been applied.

It is clear from this that there is no way to describe a pair of nodes *both* of which access each others' previous state. The reason for this is that 'recurrent' mutations in RCGP do not explicitly induce recurrent behaviour; instead they may create cycles which, in combination with the ordering imposed on nodes, leads to recurrent behaviour. However we can express such a solution as an RFG as shown in Figure 7.1, where there exists a cycle of recurrent edges between the MAX, ADD and MUL nodes.

Not only do we have access to solutions which may not be directly expressed in RCGP, we also clearly have access to additional mutations. Consider a solution which is expressible in RCGP where there exists a pair of nodes, v_1, v_2, and an edge, $v_1 \to v_2$. If we assume that v_2 appears later in the ordering than v_1, then this edge exhibits recurrent behaviour. It is impossible in RCGP for one of v_2's edges to mutate to target v_1 and for both edges to exhibit recurrent behaviour. However, this is possible in R-EGGP by a recurrent edge mutation.

Hence we have that R-EGGP can express any mutations available in RCGP. We also have that there are solutions which can be expressed in R-EGGP which can not be expressed in RCGP, and that there are mutations available in R-EGGP that transform solutions which can be expressed in RCGP into solutions which can not. Therefore R-EGGP generalises the landscape of RCGP with respect to both available solutions and available mutations.

7.5 Digital Counter Experiments

We investigate R-EGGP's ability to learn digital counters. These are a class of stateful digital circuits which have fixed behaviour and need not be evaluated on different input sequences, which greatly simplifies their evaluation and makes them very practical for benchmarking. We investigate 2 classes of counter, both of which we will assume are driven with a signal 1. The first class, ring counters, count according to a one-hot encoding. For example, the 3-bit ring counter has trace given by

$Time$	i_0	o_0	o_1	o_2
1	1	1	0	0
2	1	0	1	0
3	1	0	0	1
4	1	1	0	0
5	1	0	1	0
6	1	0	0	1
7	1	1	0	0
8	1	0	1	0
9	1	0	0	1

(7.2)

The second class of digital counters, Johnson counters, are similar to ring counters, except that they circulates strings of 1s, rather than a single 1. The 3-bit Johnson counter has trace given by

$Time$	i_0	o_0	o_1	o_2
0	1	0	0	0
1	1	1	0	0
2	1	1	1	0
3	1	1	1	1
4	1	0	1	1
5	1	0	0	1
6	1	0	0	0
7	1	1	0	0
8	1	1	1	0
9	1	1	1	1

(7.3)

.

Digital Circuit	Number of Inputs	Number of Outputs
3-bit ring counter (3-RC)	1	3
4-bit ring counter (4-RC)	1	4
5-bit ring counter (5-RC)	1	5
6-bit ring counter (6-RC)	1	6
3-bit Johnson counter (3-JC)	1	3
4-bit Johnson counter (4-JC)	1	4
5-bit Johnson counter (5-JC)	1	5
6-bit Johnson counter (6-JC)	1	6

Table 7.1: Digital counter benchmark problems.

We study 3, 4, 5 and 6 bit instantiations of each class of counter. A full listing of our benchmark problems is given in Table 7.1.

To evaluate a candidate solution we consider the sequence of outputs in comparison to the target sequence, following [242]. We run our circuits for 100 time steps. The fitness is initially 100, and is decremented by 1 for every correct output the candidate makes until it makes a mistake. By correct output, we mean the entire n-bit output, rather than comparing individual bits of the output. Once the candidate has made a mistake, any further correct predictions decrement the fitness by 0.01. This encourages circuits to generate correct sequences and provides a gradient which rewards early correct predictions over later correct predictions. This is particularly useful in the problems we study here, as simply measuring the fitness by the overall number of correct predictions induces some unhelpful local optima. For example, a candidate for a 3-bit ring counter problem can output '100' at every time step and achieve a fitness of 66.

We compare to RCGP where, across all problems, we use the function set

$$\{\text{AND}, \text{OR}, \text{NAND}, \text{NOR}\}. \tag{7.4}$$

As digital counters are a new class of benchmark problems, we therefore have to propose appropriate parameters for both algorithms. For both R-EGGP and RCGP, we use a fixed 50 node representation, the $1 + \lambda$ EA with $\lambda = 4$ and a recurrent edge probability $p_{rec} = 0.1$. With R-EGGP, we use a mutation rate of 0.02, which is a simple re-scaling of the mutation rate used in Section 6.3 to match the reduced size. We attempted to apply the same logic with

	R-EGGP		RCGP			
Problem	ME	IQR	ME	IQR	p	A
3-RC	2,683	2,105	3,495	3,263	0.01	-
4-RC	7,875	7,433	13,930	12,383	10^{-9}	**0.74**
5-RC	20,850	17,683	42,268	28,110	10^{-14}	**0.81**
6-RC	56,510	45,023	155,878	135,965	10^{-21}	**0.89**
3-JC	12,995	11,898	18,135	17,280	10^{-3}	0.63
4-JC	44,163	37,765	75,378	97,943	10^{-6}	0.69
5-JC	104,513	94,438	211,318	207,130	10^{-10}	**0.75**
6-JC	213,958	177,330	446,180	399,185	10^{-14}	**0.82**

Table 7.2: Results from Digital Counter benchmarks for RCGP and R-EGGP. The p value is from the two-tailed Mann–Whitney U test. Where $p < \frac{0.05}{8}$, the effect size from the Vargha–Delaney A test is shown; large effect sizes ($A > 0.71$) are shown in **bold**.

RCGP (i.e. a mutation rate of 0.08) but, through some trial-and-error, found a mutation rate of 0.05 to be preferable. We run each algorithm on each problem 100 times to sample data points. We set a maximum generation cap of $20,000,000$ but this is never reached as every run is successful. For RCGP experiments, we use the publicly available implementation [243].

7.6 Digital Counter Results

The results of our Digital Counter experiments are given in Table 7.2. For each approach and on each problem, we list the MEs required to solve the problem and IQR in evaluations. We test for statistical significance with the two-tailed Mann–Whitney U test producing the p values shown. Where $p < \frac{0.05}{8}$, the effect size from the Vargha–Delaney A test is shown; large effect sizes ($A > 0.71$) are shown in **bold**.

Overall, we see that R-EGGP requires fewer evaluations (measured by median) than RCGP on all problems. For every problem except the easiest problem, the 3-bit ring counter (3-RC), we find that the differences are statistically significant ($p < \frac{0.05}{8}$). For 5 of these 7 significant results, we observe a large effect size ($A > 0.71$). Only on the easiest ring counter and the two easiest Johnson counter problems (3-RC, 3-JC, 4-JC) do we not see a large effect

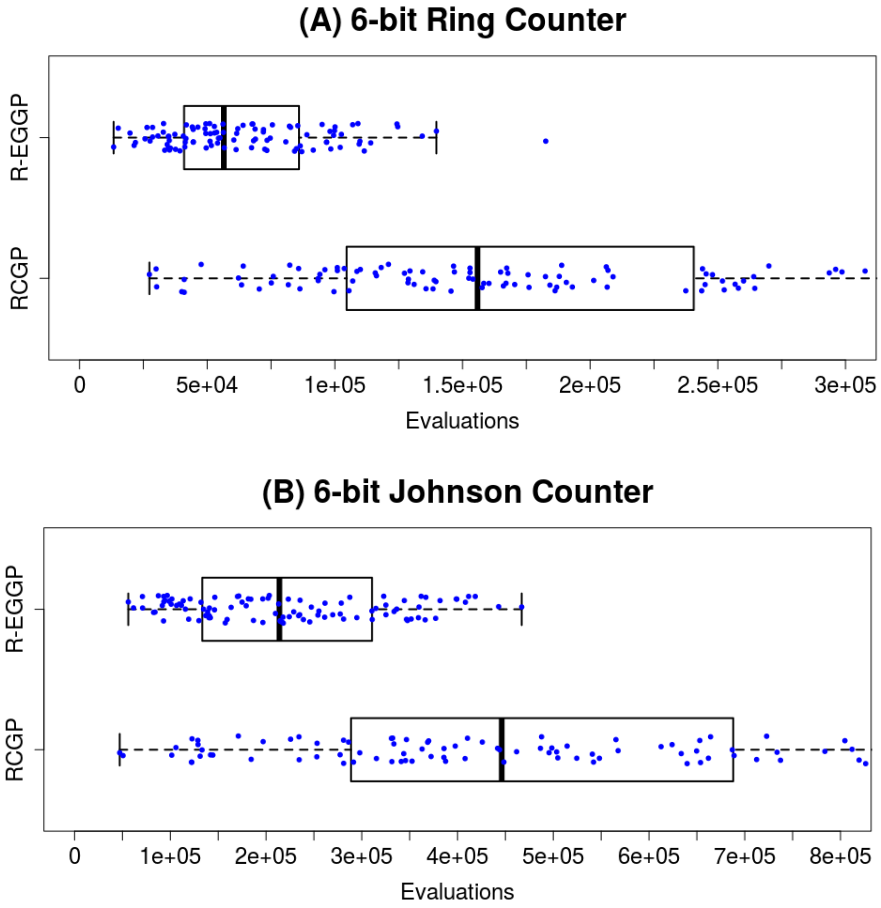


Figure 7.7: Box-plots with data overlayed for the following symbolic regression problems; (A) 6-bit Ring Counter (6-RC), (B) 6-bit Johnson Counter (6-JC). Overlayed data is jittered for visual clarity.

size. From these results we can conclude that R-EGGP can significantly outperform RCGP under comparable conditions on stateful digital circuit synthesis tasks. As we observed with the EGGP digital circuit benchmark problems, we see the p values decreasing and A values increasing as the difficulty of the task increases, suggesting that in particular R-EGGP scales better to harder problems than RCGP. We give box-plots of results for the 6-bit ring counter (6-RC) and 6-bit Johnson counter (6-JC) in Figure 7.7.

Mathematical Sequence	No. Inputs	No. Outputs	First 5 Elements
Fibonacci Sequence (Fib)	1	1	1, 1, 2, 3, 5
Hexagonal Numbers (Hex)	1	1	1, 6, 15, 28, 45
Lazy Caterers (Laz)	1	1	1, 2, 4, 7, 11

Table 7.3: Mathematical sequence benchmark problems.

7.7 Mathematical Sequence Experiments

We also study R-EGGP's ability to synthesise some famous mathematical sequences. Again, these problems have fixed behaviour which greatly simplifies their evaluation thereby making them practical for benchmarking. We study 3 famous Mathematical Sequences, taken from [242]. These are the Fibonacci sequence, hexagonal number sequence and the lazy caterers sequence. We assume that programs implementing each of these are driven with an input signal of 1. Table 7.3 details these problems.

We use the same fitness function as in our digital counter experiments, initialising the fitness equal to the sequence length and decrementing it by 1 for every successful prediction followed by 0.01 for every successful prediction after a mistake was made. The Fibonacci sequence is evaluated for a sequence length of 50[1], whereas the other 2 problems are evaluated for a sequence length of 100.

We replicate the experimental conditions used in [242] using the function set

$$\{+, -, \times, \div\}, \tag{7.5}$$

where our division operator $\div$ need not be protected as a NaN output is simply an incorrect prediction.

For both algorithms, we use a fixed 20 node representation and the $1 + \lambda$ EA with $\lambda = 4$. The probability of creating recurrent edges is set $p_{rec} = 0.1$, and the mutation rate is set to 0.05. We run each algorithm on each problem 100 times to sample data points. We set a maximum generation cap of $20,000,000$ but, as with the Digital Counter Experiments, this is never reached as every run is successful. For RCGP experiments, we use the publicly available implementation [243].

[1] We ran into some problems with integer overflows at the 100th element of the Fibonacci sequence.

	R-EGGP		RCGP			
Problem	ME	IQR	ME	IQR	p	A
Fib	6,513	13,033	6,275	14,935	0.99	-
Hex	8,158	14,903	16,988	37,715	10^{-3}	0.62
Laz	11,063	24,943	10,605	34,315	0.78	-

Table 7.4: Results from mathematical sequence benchmarks for RCGP and R-EGGP. The p value is from the two-tailed Mann–Whitney U test. Where $p < \frac{0.05}{3}$, the effect size A from the Vargha–Delaney A test is shown.

7.8 Mathematical Sequence Results

The results of our mathematical sequence experiments are given in Table 7.4. For each approach and on each problem, we list the MEs required to solve the problem and the IQR in evaluations. We test for statistical significance with the two-tailed Mann–Whitney U test producing the p values shown. Where $p < \frac{0.05}{3}$, the effect size from the Vargha–Delaney A test is shown.

Overall, we see relatively little difference in performance between R-EGGP and RCGP. On 2 of the problems, we see no statistical differences ($p \geq \frac{0.05}{3}$). On only 1 problem do we see a statistical difference, the hexagonal numbers sequence problem (Hex), and on that problem we do not see large effect ($A \leq 0.71$).

These results have an interesting interaction with our symbolic regression results from EGGP benchmarking in Section 6.7; we again see very few differences when comparing an EGGP based approach to a CGP based approach on 'symbolic' problems. This again leads us back to the discussion in Section 6.8 as to why this may be the case; these results appear to reduce the credibility of the 'bloat' hypothesis as the overall representation size used in these experiments is very small (20 nodes). However, the problems we are studying here are ones of finding globally optimal solutions, rather than approximately optimal solutions within a given budget, and that the problems here study a different class of graphs featuring recurrent edges, we cannot discount that hypothesis entirely on the basis of these experiments.

7.9 Generalising n-bit Parity Check Experiments

In our final experiments for R-EGGP, we study the algorithm's ability to learn n-bit parity checking circuits that generalise. These are circuits which take in a sequence of n bits and verify whether or not the input satisfies even or odd parity. For example, if the problem is n-bit even parity, the circuit should return that the input is valid (1) if the input sequence has an even number of 1s.

To train the circuits, we use 5 training bits. We consider each of the 2^5 unique 5-bit input sequences and use a generalisation of the fitness function used before; we initialise the fitness equal to the sum of sequence lengths and decrement it by 1 for every successful prediction followed by 0.01 for every successful prediction after a mistake was made per sequence. This is simply an extension to our earlier fitness function that supports multiple independent sequences. Note that because we are checking the circuit's outputs after 1, 2, 3, 4 and 5 bits we are implicitly testing whether the learnt circuit correctly implements 1, 2, 3 and 4-bit parity checks as well as 5-bit parity checks.

To test whether the circuit found by an evolutionary run generalises, we then test it on 14 test bits, considering each of the 2^{14} unique 14-bit sequences. Again, because we are checking the circuit's outputs after $1, 2, \ldots, 13$ and 14 bits, we are implicitly checking if the circuit correctly implements all parity checks with inputs of length 1 to 14. If the fitness at this point is 0, we consider the solution to be generalised; it has been trained on 5-bit problems, but generalises to at least 14-bit problems.

We refer to our even parity checking problem as n-EP and our odd parity checking problem as n-OP. We use the function set

$$\{\text{AND}, \text{OR}, \text{NAND}, \text{NOR}\}, \tag{7.6}$$

and use the same experimental conditions as in Section 7.5. For both R-EGGP and RCGP, we use a fixed 50 node representation, the $1 + \lambda$ EA with $\lambda = 4$ and a recurrent edge probability $p_{rec} = 0.1$. With R-EGGP, we use a mutation rate of 0.02 and for RCGP we use a mutation rate of 0.05. We run each algorithm on each problem 100 times to sample data points. We set a maximum generation cap of $20,000,000$ but, as in experiments previously described, this is never reached as every run is successful. For RCGP experiments, we use the publicly available implementation [243].

As we will shortly see, we find both systems to be remarkably effective at solving these problems. To artificially increase the difficulty of the task and thereby strengthen comparison,

	R-EGGP			RCGP				
Problem	ME	IQR	SR	ME	IQR	SR	p	A
n-EP	1,968	2,578	99%	2,735	4315	100%	10^{-3}	0.61
n-OP	1,980	2,495	98%	2,700	3513	99%	0.06	-
n-EP$_h$	3,080	3,633	99%	4,228	4,930	100%	10^{-3}	0.62
n-OP$_h$	2,098	1,925	99%	3,678	4,493	97%	10^{-6}	0.70

Table 7.5: Results from generalising n-bit parity check benchmarks for RCGP and R-EGGP. The p value is from the two-tailed Mann–Whitney U test. Where $p < \frac{0.05}{4}$, the effect size A from the Vargha–Delaney A test is shown.

we also use the (h)arder function set

$$\{\mathrm{OR}, \mathrm{NOR}\}, \tag{7.7}$$

and refer to the problems in which they are used as n-EP$_h$ and n-OP$_h$ respectively.

7.10 Generalising n-bit Parity Check Results

The results from our generalising n-bit parity checking experiments are given in Table 7.5. We list the MEs required to solve the problem and the IQR in evaluations. We also give the generalisation success rate (SR) of each algorithm, which is the proportion of runs in which the found solution perfectly generalised to 14-bit sequences. We test for statistical significance with the two-tailed Mann–Whitney U test producing the p values shown. Where $p < \frac{0.05}{4}$, the effect size from the Vargha–Delaney A test is shown.

We find that on 3 of the problems (n-EP, n-EP$_h$ and n-OP$_h$) R-EGGP found solutions more quickly than RCGP with respect to evaluations used (measured by median) with statistical significance ($p < \frac{0.05}{4}$). On 1 problem, n-OP, we see lower MEs used but without statistical significance. In no problems do we see a large effect size. From these results we can infer that R-EGGP is in general more effective than RCGP at synthesising generalising n-bit parity checks, although on the problems studied the difference in performance is not large.

It is interesting to examine the rate of successful generalisation of solutions found by both algorithms. On all problems, the success rate (SR) was close to 100%. There are no significant differences between SRs, but it is interesting to see that both algorithms are capable of finding

n-bit parity checks which generalise to bit sequences longer than those they were trained on.

7.11 Conclusions and Future Work

In this chapter we have presented R-EGGP. We have presented an initialisation procedure which generates RFGs and parameterised by probability p_{rec} which controls the rate of recurrent edges in the initial solution. We have also given two edge mutations, one which mutates an edge to be non-recurrent while maintaining the acyclicity of the subgraph induced by non-recurrent edges, and the other which mutates an edge to be recurrent.

We have extensively compared R-EGGP to RCGP on various benchmark problems. On digital counter synthesis problems, we found that R-EGGP significantly outperforms RCGP on many problems, particularly the most difficult problems. On mathematical sequence synthesis problems, we found few statistical differences, but did observe that R-EGGP significantly outperforms RCGP on one problem. On three of the n-bit parity problems, we found that R-EGGP significantly outperforms RCGP.

Overall, we have described and rigorously evaluated an approach for evolving RFGs which has genetic operators described as P-GP 2 programs. We have found that this technique can effectively learn solutions to a variety of recurrent program synthesis tasks and often outperforms RCGP. In particular we have seen that R-EGGP can synthesise recurrent digital circuits which generalise to solve problems they were not trained on.

There are a number of areas for future work on R-EGGP. Firstly, in our experiments we have fixed the rate of recurrent edges $p_{rec} = 0.1$. It would be interesting to carry out experiments varying this parameter. We expect that increasing p_{rec} would lead to a larger solution size, as recurrent edges may add entire new subgraphs to the active component. Whether this helps or hinders the evolutionary process is a matter for empirical analysis.

Interesting behaviour occurs at the two extremes of parameterisation of p_{rec}. When $p_{rec} = 0$, R-EGGP is equivalent to EGGP. However, when $p_{rec} = 1$, then all constraints of acyclicity of the individual are removed. This leads to some interesting insights, for example that the initialisation procedure might be viewed as a variant of the directed random graph model from Section 3.3.4 with a fixed degree sequence (see [159]). Observations such as this may then yield new understanding of the biases of initialisation. R-EGGP with $p_{rec} = 1$ may be used as a model of of search in many interesting domains, for example, in the search for a topology of an echo state network [108] or a random Boolean network [211].

8 Evolving Graphs with Semantic Neutral Drift

Abstract

We introduce the concept of Semantic Neutral Drift (SND) for Evolving Graphs by Graph Programming (EGGP), where we exploit equivalence laws to design semantics-preserving mutations guaranteed to preserve individuals' fitness scores. A number of digital circuit benchmark problems are implemented with rule-based graph programs and empirically evaluated, demonstrating quantitative improvements in evolutionary performance. Analysis reveals that the benefits of the designed SND reside in more complex processes than simple growth of individuals, and that there are circumstances where it is beneficial to choose otherwise detrimental parameters for a Genetic Programming (GP) system if that facilitates the inclusion of SND.

Relevant Publications

Content from the following publications is used in this chapter:

[11] T. Atkinson, D. Plump, and S. Stepney, "Evolving graphs with semantic neutral drift," *Natural Computing*, 2019.

8.1 Introduction

In Genetic Programming (GP) the ability to escape local optima is key to finding globally optimal solutions. Neutral drift, a mechanism whereby individuals with fitness-equivalent phenotypes to the existing population may be generated by mutation [72] offers the search of new neighbourhoods for sampling thus increasing the chance of leaving local optima. A number of studies on neutrality in Cartesian Genetic Programming (CGP) [156,244,251] find it to be an almost always beneficial property for studied problems. In general, comparative studies [155] find that CGP using only mutation and neutral drift is able to compete with traditional Tree-Based GP (TGP) which uses more familiar crossover operators (see [129]) to introduce genetic variation.

A distinction has been made [244] between *implicit* neutral drift, where a genetic operator yields a semantically equivalent child, and *explicit* neutral drift, where a genetic operator only modifies intronic code. We note that many comparative studies largely focus on the role of both types of neutral drift as byproducts of existing genetic operators and neutrality within the representation [19,156,244,251] rather than as deliberately designed features of an evolutionary system. We propose the opposite; to employ domain knowledge of equivalence laws to specify mutation operators on the active components of individuals which always induce neutral drift. Hence our work can be viewed as an attempt to explicitly induce additional implicit neutral drift in the sense of [244].

We build on our approach, Evolving Graphs by Graph Programming (EGGP), by implementing *semantics-preserving mutations* to directly achieve neutral drift on the active components of individual solutions. Here, we implement logical equivalence laws as mutations on the active components of candidate solutions to digital circuit problems to produce semantically equivalent, equally fit, children. While our semantics-preserving mutations produce semantically equivalent children they do not guarantee preservation of size; our fitness measures evaluate semantics only, not, for example, size or complexity.

We describe and implement Semantic Neutral Drift (SND) straightforwardly by using rule-based graph programs in P-GP 2. This continues from Chapter 5 where we use P-GP 2 to design acyclicity-preserving edge mutations for digital circuits that correctly identify the set of all possible valid mutations. The use of P-GP 2 here enables concise description of complex transformations such as De Morgan's laws by identifying and rewriting potential matches for these laws in the existing formalism of graph transformation. This reinforces the notion that the direct encoding of solutions as graphs is useful as it allows immediate access to the

phenotype of individual solutions and makes it possible to design complex mutations by using powerful algorithmic concepts from graph programming.

We investigate four sets of semantics-preserving mutations for digital circuit design, three built upon logical equivalence laws and a fourth taken from term-graph rewriting. We run EGGP with each rule-set on a set of benchmark problems and establish statistically significant improvements in performance for most of our visited problems. An analysis of our results reveals evidence that it is the semantic transformations, beyond simple 'neutral growth', which are aiding performance. We then combine our two best performing sets of mutation operators and evaluate this new set under the same conditions, achieving further improvements. We also provide evidence that, although operators implementing semantics-preserving mutations may be more difficult to use, the inclusion of those semantics-preserving mutations may allow evolution to out-perform equivalent processes that use 'easier' operators.

The rest of this chapter is organised as follows. In Section 8.2 we review existing literature on neutral drift in GP. In Section 8.3 we describe our extension to EGGP where we incorporate deliberate neutral drifts into the evolutionary process. In Section 8.4 we describe our experimental setup and in Section 8.5 we give the results from these experiments. In Section 8.6 we provide in-depth analysis of these results to establish precisely what components of our approach are aiding performance. In Section 8.7 we conclude our work and propose potential future work on this topic.

8.2 Neutrality in Genetic Programming

Neutral drift remains a controversial subject in Evolutionary Computation, see [72] for a survey on this matter. Here, we focus on neutrality in the context of GP as the most relevant area to our own work; there is also literature on, for example, Genetic Algorithms (GAs) [93] and landscape analysis [20].

The process of neutral drift might be described as the mutation of individual candidate solutions to a given problem without advantageous or deleterious effect on their fitness. This exposes the Evolutionary Algorithm (EA) to a fitness 'plateau' with each fitness-equivalent individual offering a different portion of the landscape to sample. Neutral drift can be viewed as random walks on the neighbourhoods of surviving candidate solutions. In a system with neutral drift, an apparently local optimum might be escaped by 'drifting' to some other fitness-equivalent solution that has advantageous mutations available to it.

The most apparent demonstration of neutral drift in GP literature occurs in CGP [157], where individuals encode directed acyclic graphs; some portion of a genome may be 'inactive', contributing nothing to the phenotypic fitness, because it represents a subgraph that is not connected to the phenotype's main graph. These inactive genes can mutate without influencing an individual's fitness and then, at some later point, may become active. Early work on CGP has found that by allowing neutral drift to take place (by choosing a fitness-equivalent child over its parent in the $1 + \lambda$ algorithm), the success rate of experiments significantly improves [251]. A later claim that neutrality in CGP aids search in needle-in-haystack problems [266] has been contested by a counter-claim that better performance can be achieved by random search [43]. It has been found that better performance can be achieved with neutral drift enabled by increasing the amount of redundant material present in individuals [156]. Further, distinction has been established between *explicit* and *implicit* neutral drift [244]. Explicit neutral drift occurs on inactive components of the individual, whereas implicit neutral drift occurs when active components of the individual are mutated but the fitness does not change. The authors were able to isolate explicit neutral drift and demonstrate that it offers additive benefits beyond those of implicit neutral drift.

Outside of CGP, [19] describes Linear Genetic Programming (LGP), where the results of individual instructions may never be used. There are notable similarities between CGP and LGP with respect to their representation of neutral code as unused elements of a list of functions. In both approaches, unused material may undergo explicit neutral drift thereby exposing the search process to new neighbourhoods. A study of evolvability in LGP [103]

found that neutrality cooperates with 'variability' (the ability of a system to generate phenotypic changes) to generate adaptive phenotypic changes which aid the overall ability of the system to respond to the landscape. Recent work [104] studying the role of neutrality in small LGP programs found that the robustness of a genotype (the proportion of its neighbours within the landscape which are neutral changes) has a complex and non-monotonic relationship with the overall evolvability of the genotype. A detailed discussion of the role of neutrality in LGP can be found in [27].

In [54], binary decision diagrams are evolved with explicit neutral mutations. Although those neutral mutations are not isolated for their advantages/disadvantages, a later work has found that a higher rate of neutral drift on binary decision diagrams is advantageous [55]. Koza also makes some reference to the ideas we employ in Section 8.3 when he describes the editing digital circuits by applying De Morgan's laws to them [129, Ch.6]. A study of neutrality in TGP for Boolean functions [247] found a correlation between using a more effective function set and the existence of additional neutrality when using that function set.

While not directly related to neutrality, a number of investigations have been carried out exploring the notion of semantically aware genetic operators to improve the locality of mechanisms such as crossover in TGP [162,168]. We refer the reader to the extensive survey [246] on this field of research. Whereas neutrality is the process whereby phenotypically identical and genotypically distinct individuals are visited by the evolutionary process, semantically aware genetic operators attempt to produce phenotypically 'close' individuals to improve the locality of the search neighbourhood. It should be noted that employing semantically aware genetic operators may sometimes lead to a loss of diversity [178]. It could be argued that the deliberate neutral operators we propose in this work are a form of semantically aware mutation operators designed to explicitly exploit neutrality.

Neutral drift has some parallels with work on biological evolution. Kimura's *Neutral Theory of Molecular Evolution* [127] posits that most mutations in nature are neither advantageous or deleterious, instead introducing 'neutral' changes that do not affect phenotypes but account for much of the genetic variation within species. While Kimura's theory remains controversial (see [87]), it corresponds to the notions of neutral mutation described in GP literature.

Throughout the literature we have covered, neutrality is mostly considered in the sense of *explicit* neutral drift as defined in [244]. Conversely, in our work here we are focusing on neutral drift on the active components of individual solutions with some relationship, therefore, to the neutral mutations on binary decision diagrams in [54].

8.3 Semantic Neutral Drift

8.3.1 The Concept

SND is the augmentation of an evolutionary system with semantics-preserving mutations. These mutations are added to the standard mutation and crossover operators, which are intended to introduce variation to search. In this section we refer to mutation operators and individuals generally, not just our specific operation. For individual solutions, i, j, and mutation operator, m, we write $i \rightarrow_m j$ to mean that j can be generated from i by using mutation m. A semantics-preserving mutation is one that guarantees that the semantic meaning of a child generated by that mutation is identical to that of its parent, for any choice of parents and a given semantic model. This definition is adequate for our domain of GP, where there is no distinction between the genotype and phenotype.

For our digital circuits case study, this semantic equivalence is well-defined: two circuits are semantically equivalent if they describe identical truth tables. Therefore, semantics-preserving mutations in this context are ones which preserve an individual's truth table. As we will be evaluating individuals by the number of incorrect bits in their truth tables, there may be individuals with equivalent fitness but different truth tables. Therefore, semantic equivalence is distinct from, but related to, fitness equivalence.

Additionally, semantics-preserving mutations do not necessarily induce neutral drift. In the circumstance that a fitness function considers more than the semantics of an individual, there is no guarantee that the child of a parent generated by a semantics-preserving mutation has equal fitness to its parent. For example, if a fitness function penalised the size of an individual, a semantics-preserving mutation which introduces additional material (i.e. increases its size) would generate children less fit than their parents under this measure.

We identify a special class of fitness functions, where fitness depends only on semantics, and so where semantics-preserving mutations are guaranteed to preserve fitness. In this circumstance, any use of semantics-preserving mutations is a deliberate, designed-in, form of neutral drift. The fitness function in our case study is an example of this; the fitness of an individual depends only on its truth table. Formally we have the following: a set of semantics-preserving mutation operators, M, over search space, S, with respect to a fitness function, f, that considers only semantics guarantees that

$$\forall i, j \in S, m \in M : (j \rightarrow_m i) \Rightarrow (f(i) = f(j)).$$

Consider an evolutionary run that has reached a local optimum; no available mutations or

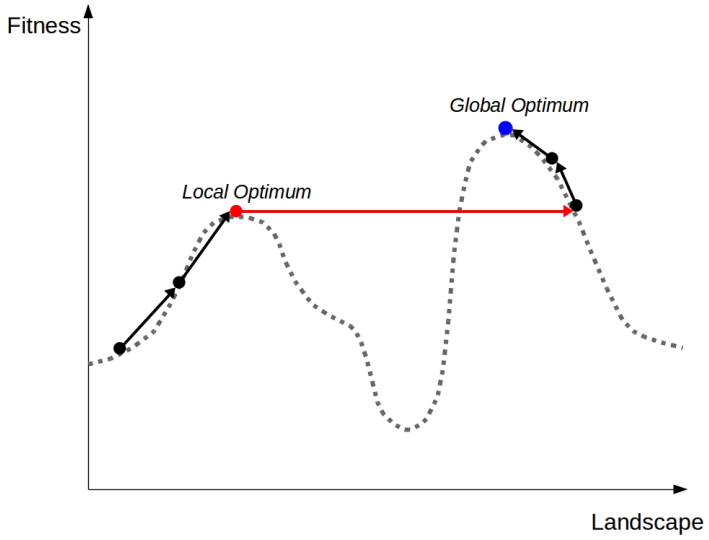


Figure 8.1: A simple visualisation of SND. Individuals exist in one dimension along the x-axis with their associated fitness on the y-axis. Normal mutations (black arrows) allow the EA to hill-climb by sampling from adjacent points. A semantics-preserving mutation (red arrow) allows the EA to leave a local optimum to move to a different slope where it can then climb to the global optimum.

crossover operators offer positive improvements with respect to the fitness function. It may be the case that a solution exists elsewhere in the landscape that is equally fit but has a neighbourhood with positive mutations available. By applying a semantics-preserving mutation to transform the best found solution into this other, semantically equivalent, solution, the evolutionary process gains access to this better neighbourhood to continue its search. Hence the proposed benefit of SND is the same as conventional neutral drift: that by transforming discovered solutions we gain access to different parts of the landscape that may allow the population to escape local optima. The distinction here is that we are employing domain knowledge to deliberately preserve semantics, rather than accessing neutral drift as a byproduct of other evolutionary processes. We investigate the hypothesis that this deployment of domain knowledge yields more meaningful neutral mutations than simple rewrites of intronic code, and that this leads the EA to more varied, and therefore useful, neighbourhoods.

A simple visualisation of SND is given in Figure 8.1. Here the landscape exists in one dimension, the x-axis, with fitness of individuals given in the y-axis. In this illustration, the individual has reached a local optimum, then a semantics-preserving mutation moves it to a

different 'hill' from which it is able to reach the global optimum.

While our experiments will focus on the role of SND when evolving graphs with EGGP, we argue that the underlying concept is extendable to other GP systems. For example, Koza noted the possibility of applying De Morgan's laws to GP trees [129, Ch.6] which, if used in a continuous process rather than as a solution optimiser, would induce SND. It is also plausible to apply similar operators to CGP [157] representations, although the ordering imposed on the representation raises some technical difficulties with respect to where newly created nodes should be placed. The potential for Embedded CGP [253] to effectively grow and shrink the overall size of the genotype offers some hope in this direction.

8.3.2 Designing Semantic Neutral Drift

We extend EGGP by applying semantics-preserving mutations to members of the population each generation. We focus on digital circuits as a case study, and design mutations which modify the individual's *active* components by exploiting domain knowledge of logical equivalence.

For the function set, {`AND`,`OR`,`NOT`}, there are a number of known logical equivalences. Here we use De Morgan's laws:

$$\begin{aligned}
\text{DeMorgan}_{F1}&: \neg(a \wedge b) = \neg a \vee \neg b;\\
\text{DeMorgan}_{F2}&: \neg(a \vee b) = \neg a \wedge \neg b;\\
\text{DeMorgan}_{R1}&: \neg a \vee \neg b = \neg(a \wedge b);\\
\text{DeMorgan}_{R2}&: \neg a \wedge \neg b = \neg(a \vee b),
\end{aligned}$$

and the identity and double negation laws:

$$\begin{aligned}
\text{ID-AND}_{F}&: a = a \wedge a;\\
\text{ID-AND}_{R}&: a \wedge a = a;\\
\text{ID-OR}_{F}&: a = a \vee a;\\
\text{ID-OR}_{R}&: a \vee a = a;\\
\text{ID-NOT}_{F}&: a = \neg\neg a;\\
\text{ID-NOT}_{R}&: \neg\neg a = a.
\end{aligned}$$

Here we investigate different subsets of these semantics-preserving rules. We encode them as graph transformation rules to apply to the active component of an individual. In the context of the $1+\lambda$ EA, we apply one of the rules from the subset to the surviving individual of each generation.

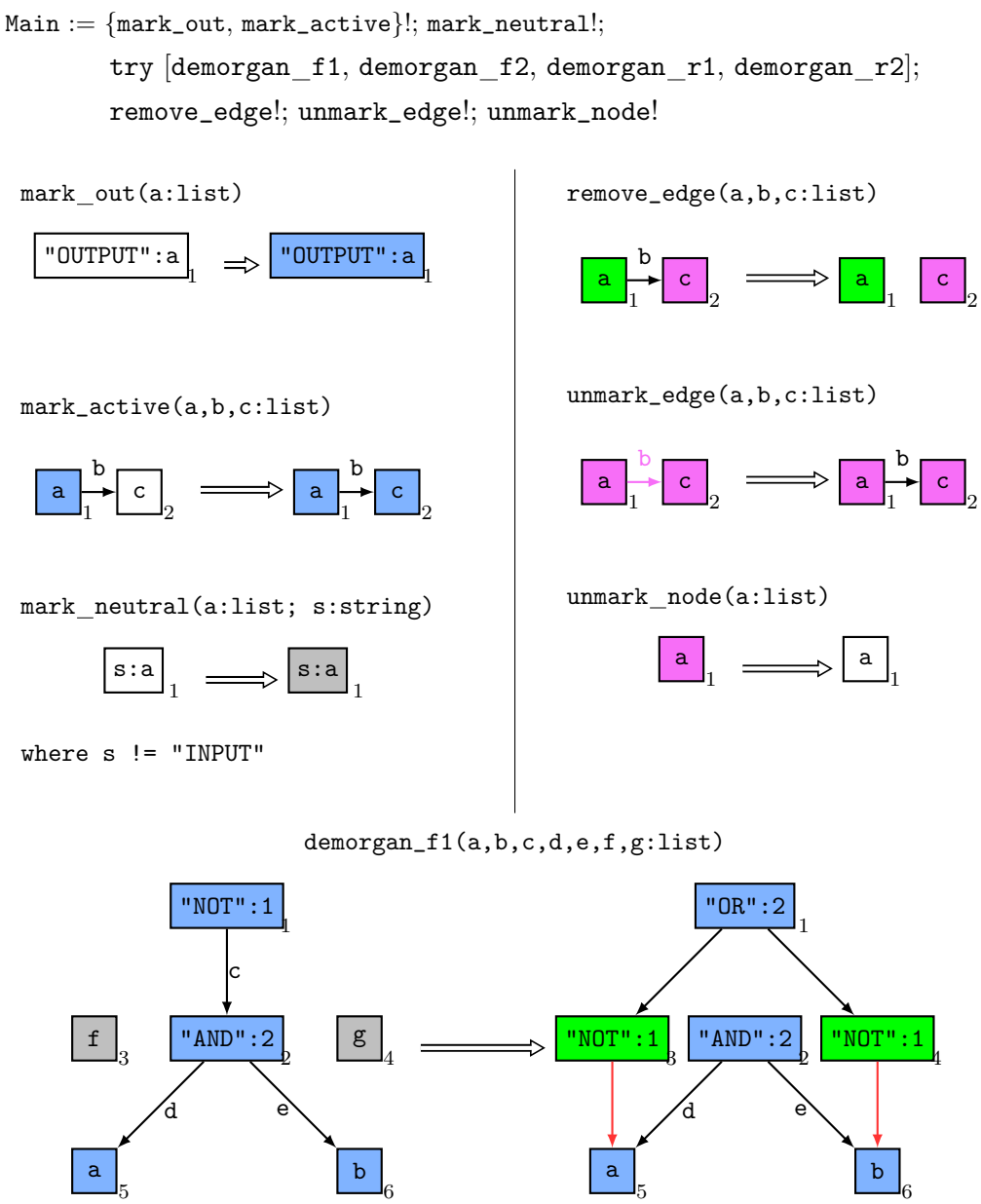


Figure 8.2: A P-GP 2 program for performing semantics-preserving mutations to digital circuits.

Encoding these semantics-preserving rules is non-trivial for our individuals as they incorporate sharing; multiple nodes may use the same node as an input, and therefore rewriting or removing that node, e.g. as part of De Morgan's, may disrupt the semantics elsewhere

in the individual. To overcome this, we need a more sophisticated rewriting program. The graph program in Figure 8.2 is designed for the logical equivalence laws $\text{DeMorgan}_{F1|F2}$ and $\text{DeMorgan}_{R1|R2}$; analogous programs are used for other operators. The program `Main` in Figure 8.2 works as follows:

{`mark_out`, `mark_active`}! : Mark all active nodes with the given rule-set applied as long as possible. Once this rule-set has no matches, all inactive nodes must be unmarked: these are 'neutral' nodes that do not contribute to the semantics of the individual.

`mark_neutral`! : Mark these neutral nodes grey with the rule applied as long as possible. We can then rewrite the individual while preserving semantics with respect to shared nodes by incorporating neutral nodes into the active component rather than overwriting existing nodes.

`try` [`demorgan_f1`, `demorgan_f2`, `demorgan_r1`, `demorgan_r2`] : pick some rule with uniform probability from the subset of the listed rules that have valid matches. When a rule has been chosen, a match is chosen for it from the set of all possible matches with uniform probability. The probabilistic rule-set call is surrounded by a `try` statement to catch the fail case that none of the rules have matches.

In Figure 8.2 we show one of the 4 referenced rules, `demorgan_f1`, which corresponds to the logical equivalence law DeMorgan_{F1}; the others may be given analogously. On the left hand side is a match for the pattern $\neg(a \wedge b)$ in the active component and 2 neutral nodes. If the matched pattern were directly transformed, any nodes sharing use of the matches for node 2 or node 3 could have their semantics disrupted. Instead, the right-hand-side of `demorgan_f1` changes the syntax of node 1 to correspond to $\neg a \vee \neg b$ by absorbing the matched neutral nodes (preserving the graph's semantics) without rewriting nodes 1 or 2 and disrupting their semantics. Nodes 3 and 4 are marked green and their newly created outgoing edges are marked red. These marks are used later in the program to clean up any previously existing outgoing edges they have to other parts of the graph.

`remove_edge`: once a semantics-preserving rule has been applied, the rule is applied as long as possible to remove the other outgoing edges of green marked absorbed nodes.

`unmark_edge!; unmark_node!`: return the graph to an unmarked state, where nodes and edges with any mark (indicated by magenta edges and nodes in the rules) have their marks removed.

This program highlights the helpfulness of graph programming for this task. The proba-

Set	Rules
De Morgan (DM)	DeMorgan_{F1}, DeMorgan_{F2}, DeMorgan_{R1}, DeMorgan_{R2}
De Morgan and Negation (DMN)	DeMorgan_{F1}, DeMorgan_{F2}, DeMorgan_{R1}, DeMorgan_{R2}, ID-NOT_F, ID-NOT_R
Identity (ID)	ID-AND_F, ID-AND_R, ID-OR_F, ID-OR_R, ID-NOT_F, ID-NOT_R
Collapse/Copy (CC)	collapse_1, collapse_2, copy_1, copy_2

Table 8.1: The studied semantics-preserving rule-sets.

bilistic application of complex transformations, such as De Morgan's law, to only the active components of a graph-like program with sharing is non-trivial, but can be concisely described by a graph program.

8.3.3 Variations on our approach

We identify 3 sets of logical equivalence rules to study, alongside another example of semantics-preserving transformation taken from term-rewriting theory. These sets are detailed in Table 8.1. The first 3 sets comprise the logical equivalence laws already discussed. The last, CC, refers to collapsing and copying from term graph rewriting (see [84]). Collapsing is the process of merging semantically equivalent subgraphs, and copying is the process of duplicating a subgraph.

The rules $\texttt{collapse}_2$ and $\texttt{copy}_2$ are shown in Figure 8.3. These collapse and copy, respectively, function nodes of arity 2 without garbage collection. We only require rules for arity 1 and arity 2 as our function sets in experiments are limited to arity 2. This final set is included for several reasons: it takes a different form from the domain-specific logical equivalence laws in the other 3 sets; it allows us to investigate if the apparent overlap between term-graph rewriting and EAs bears fruit; it appears to resemble gene duplication, which is a natural biological process believed to aid evolution [267].

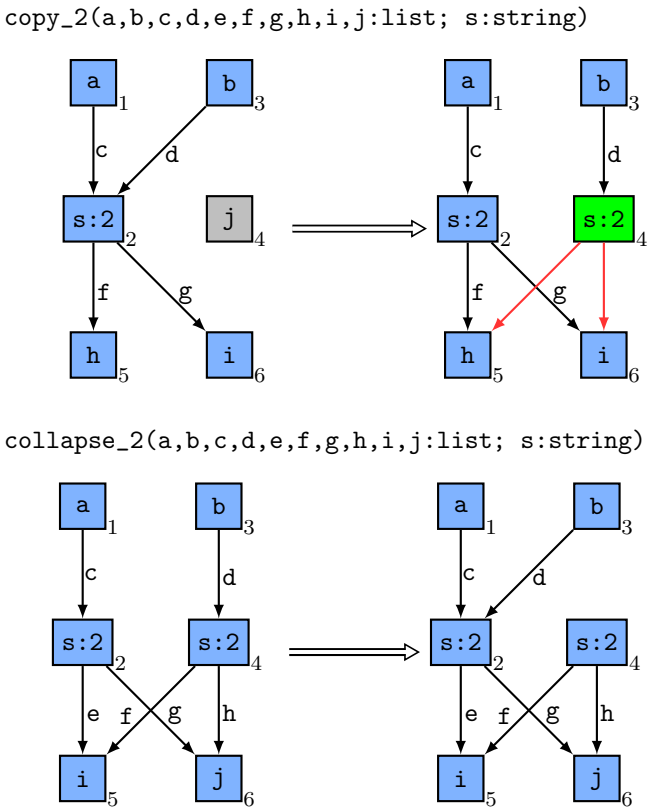


Figure 8.3: The rules `copy_2` and `collapse_2`. The rule `copy_2` matches a 2-arity function node that is shared by 2 active nodes and absorbs a neutral node to effectively copy that 2-arity function node and redirect one of the original node's shared incoming edges to that copy. The rule `collapse_2` attempts the reverse of `copy_2` by matching 2 active identical 2-arity function nodes and redirecting one of those nodes' incoming edges to the other. The node which has lost an incoming edge, if it was shared by no other nodes, may now become neutral.

Digital Circuit	No. Inputs	No. Outputs
1-bit Adder (1-Add)	3	2
2-bit Adder (2-Add)	5	3
3-bit Adder (3-Add)	7	4
2-bit Multiplier (2-Mul)	4	4
3-bit Multiplier (3-Mul)	6	6
3:8-bit De-Multiplexer (DeMux)	3	8
4×1-bit Comparator (COMP)	4	18
3-bit Even Parity (3-EP)	3	1
4-bit Even Parity (4-EP)	4	1
5-bit Even Parity (5-EP)	5	1
6-bit Even Parity (6-EP)	6	1
7-bit Even Parity (7-EP)	7	1

Table 8.2: Digital circuit benchmark problems.

8.4 Digital Circuit Experiments

To evaluate our approach, we study a subset of digital circuit benchmark problems used in Chapter 6, listed in Table 8.2. We perform 100 runs of each of our 4 neutral drift sets (Table 8.1) on each problem (Table 8.2). We use the $1+\lambda$ EA with $\lambda = 4$. We use a mutation rate of 0.01 and fix all individuals to use 100 function nodes. The fitness function used is the number of incorrect bits in an individual's truth table compared to the target truth table, hence we are minimising the fitness. We are able to achieve 100% success rate in finding global optima in our evolutionary runs, so we compare the number of evaluations required to find perfect fitness.

The function set used here is $\{\texttt{AND}, \texttt{OR}, \texttt{NOT}\}$, rather than the set $\{\texttt{AND}, \texttt{OR}, \texttt{NAND}, \texttt{NOR}\}$ used in Chapter 5 and [155, Ch.2]. Our function set is chosen to directly correspond to the logical equivalence laws used. To give context to the results in Section 8.5, and to highlight that the chosen function set is the harder of the two, we run EGGP with both function sets and detail the results in Table 8.3. For additional context, the comparative study in Chapter 6 has shown EGGP to perform favourably in comparison to CGP on these problems with the

	EGGP					
Problem	{AND, OR, NOT}		{AND, OR, NAND, NOR}			
	ME	IQR	ME	IQR	p	A
1-Add	15,538	18,963	7,495	8,764	10^{-7}	0.71
2-Add	162,003	172,781	82,688	79,333	10^{-8}	**0.73**
3-Add	742,948	679,040	309,570	288,865	10^{-16}	**0.83**
2-Mul	21,733	28,319	14,263	13,801	10^{-4}	0.65
3-Mul	1,326,880	907,544	932,430	643,529	10^{-6}	0.68
DeMux	28,123	17,450	17,100	10,763	10^{-9}	**0.75**
COMP	408,448	275,581	147,343	128,304	10^{-17}	**0.85**
3-EP	7,403	8,051	4,295	5,500	10^{-4}	0.66
4-EP	26,715	20,430	16,445	13,568	10^{-9}	**0.73**
5-EP	76,608	57,518	42,778	29,454	10^{-10}	**0.75**
6-EP	175,908	120,504	80,940	56,283	10^{-15}	**0.83**
7-EP	380,600	237,965	157,755	118,065	10^{-19}	**0.87**

Table 8.3: Baseline results from digital circuit benchmarks for EGGP on the {AND, OR, NOT} and {AND, OR, NAND, NOR} function sets. ME/IQR: the median/inter-quartile range of the number of evaluations used to solve the problem. The p value is from the two-tailed Mann–Whitney U test. Where $p < \frac{0.05}{12}$, the effect size from the Vargha–Delaney A test is shown; large effect sizes ($A > 0.71$) are shown in **bold**.

{AND, OR, NAND, NOR} function set.

We use a two-tailed Mann–Whitney U test to establish a statistically significant difference between the median number of evaluations using the two different function sets. When a result is statistically significant ($p < \frac{0.05}{12}$) we also use a Vargha–Delaney A test to measure the effect size. On every problem, using {AND, OR, NOT} takes significantly ($p < \frac{0.05}{12}$) more effort (in terms of evaluations) than when using {AND, OR, NAND, NOR}, and on all but the easiest problems, the effect size is large ($A > 0.71$). This justifies our assertion that the former function set is 'harder' to evolve.

Circuit	Neutral Rule-set											
	DM			DMN			ID			CC		
	ME	p	A	ME	p	A	ME	p	A	ME	p	A
1-Add	8,950	10^{-7}	**0.72**	9,893	10^{-5}	0.68	9,093	10^{-7}	0.71	8,275	10^{-7}	**0.72**
2-Add	65,692	10^{-14}	**0.81**	49,200	10^{-21}	**0.88**	73,275	10^{-12}	**0.79**	103,393	10^{-5}	0.68
3-Add	255,003	10^{-19}	**0.87**	186,647	10^{-25}	**0.93**	279,140	10^{-18}	**0.86**	592,815	0.09	–
2-Mul	19,853	0.36	–	16,680	0.01	–	13,312	10^{-7}	0.71	19,995	0.29	–
3-Mul	955,418	10^{-3}	0.63	678,403	10^{-11}	**0.77**	591,748	10^{-22}	**0.89**	975,558	10^{-4}	0.65
DeMux	19,633	10^{-5}	0.68	16,678	10^{-12}	**0.79**	29,700	0.59	–	19,098	10^{-5}	0.67
COMP	542,290	10^{-3}	0.63	453,730	0.44	–	298,758	10^{-4}	0.66	576,263	10^{-4}	0.64
3-EP	6,283	0.05	–	5,248	10^{-3}	0.61	5,990	10^{-3}	0.61	5,860	0.08	–
4-EP	23,828	0.06	–	20,278	10^{-5}	0.66	18,745	10^{-6}	0.69	20,295	10^{-3}	0.62
5-EP	57,333	0.01	–	58,408	10^{-3}	0.62	43,313	10^{-10}	**0.76**	60,087	0.01	–
6-EP	129,910	10^{-5}	0.67	134,770	0.03	–	104,392	10^{-9}	**0.74**	113,037	10^{-6}	0.68
7-EP	232,735	10^{-9}	**0.75**	330,572	0.05	–	221,790	10^{-12}	**0.78**	219,237	10^{-12}	**0.78**

Table 8.4: Results from digital circuit benchmarks for the various proposed neutral rule-sets. The p value is from the two-tailed Mann–Whitney U test. Where $p < \frac{0.05}{12}$, the effect size from the Vargha–Delaney A test is shown; large effect sizes ($A > 0.71$) are shown in **bold**.

8.5 Digital Circuit Results

The results from our experiments are given in Table 8.4. Each neutral rule-set is listed with the Median Evaluations (MEs) required to solve each benchmark problem.

We use a two-tailed Mann–Whitney U test to demonstrate statistical significance in the difference of the MEs for these runs and the unmodified EGGP results given in Table 8.3.

For most problems and neutral rule-sets, the inclusion of SND yields statistically significant improvements in performance. There are some exceptions: for the 4×1-bit comparator (COMP) problem, the inclusion of neutral rule-sets leads either to insignificant differences or to significantly worse performance for every rule-set except the Identity (ID), which performs significantly better. The De Morgan's rule-set (DM) and Copy/Collapse rule-set (CC) appear to yield the smallest benefit, finding significant improvement on only 8 of the 13 benchmark

Problem	DMN		ID		EGGP		p		
	MAS	IQR	MAS	IQR	MAS	IQR	DMN vs. ID	DMN vs. EGGP	ID vs. EGGP
3-Add	96.9	1.3	92.3	1.2	50.8	2.6	10^{-33}	10^{-34}	10^{-34}
COMP	99.3	95.6	92.3	0.5	67.0	2.3	10^{-34}	10^{-34}	10^{-34}

Table 8.5: Observed average solution size of the surviving population for the DMN rule-set, ID rule-set and EGGP without a neutral rule-set. Results are for the 3-Bit Adder (3-Add) and 4×1-Bit Comparator (COMP) problems. For each result, the Median Average Size (MAS) and Interquartile Range (IQR) are given. The p value is from the two-tailed Mann–Whitney U test.

problems respectively. Additionally, both of these rule-sets yield significantly worse performance for the 4×1-bit COMP problem. The De Morgan's and Negation rule-set (DMN) also finds significant improvement on only 8 of the 13 benchmark problems, but we see no statistical differences on the 4×1-bit COMP problem. Further, the DMN rule-set offers the best performance on the 2-bit and 3-bit adder problems (2-Add and 3-Add), in terms of MEs, p value and effect size. The ID rule-set achieves the best performance on the 2-bit and 3-bit multiplier problems (2-Mul and 3-Mul) but fails to achieve significant improvements on the 3:8-bit de-multiplexer problem (DeMux).

Our results show that, for some problems and certain neutral rule-sets, the inclusion of neutral drift may improve performance with respect to the effort (measured by the number of evaluations) required. Additionally, they offer strong evidence for the claim that there are some neutral rule-sets which may generally improve performance for a wide range of problems, particularly evidenced by the DMN and ID rule-sets.

We identify ID as the best performing rule-set and DMN as the second best performing rule-set. For this reason, these rule-sets are the subject of further analysis in Section 8.6.

8.6 Analysis

8.6.1 Neutral Drift or Neutral Growth?

Analysis of the run-time of EGGP augmented with the DMN and ID neutral rule-sets reveals their bias towards searching the space of larger solutions. When we refer to larger solutions, given that EGGP uses fixed-size representations, we refer to the proportion of the individual graph which is active, defined by the number of nodes to which there is a path from an output node. We demonstrate this with the results given in Table 8.5. Here, we measure the average (mean) size of the single surviving member throughout evolutionary runs on the 3-Add and COMP problems and give the median and IQR of these average sizes over 100 runs. The size of an individual is the number of active function nodes (those which are reachable from output nodes) contained within it. We give these values for DMN, ID and EGGP alone. We use a two-tailed Mann–Whitney U test to measure for statistical differences between these observations. On both problems, DMN has a higher Median Average Size (MAS) than both ID and EGGP alone ($p < \frac{0.05}{2}$) and ID also has a higher MAS than EGGP alone ($p < \frac{0.05}{2}$).

This observation challenges existing ideas that increasing the proportion of inactive code aids evolution [156]. We are able to achieve improvements in performance while effectively reducing the proportion of inactive code. It may be the case that high proportions of inactive code are helpful only when other forms of neutral drift are not available.

The result that DMN and ID increase the active size of individuals initially appears to challenge our hypothesis that it is SND that aids evolution. An alternative explanation could be that it is 'neutral growth', where our neutral rule-sets increase the size of individuals, that biases search towards larger solutions, which then happen to be better candidates for the problems we study. However, the CC neutral rule-set exclusively features neutral growth and neutral shrinkage, exploiting no domain knowledge beyond the notion that identical nodes in identical circumstances perform the same functionality, and featuring no meaningful semantic rewriting. We therefore compare how CC and DMN perform with different numbers of nodes available, to determine whether larger solutions are indeed better candidates for the studied problems.

We run DMN, CC and standard EGGP on the 2-Add, 3-Add and COMP problems, with fixed representation sizes of 50, 100 and 150 nodes. If it is the case that larger solutions are better candidates, and that our neutral rule-sets bias towards neutral growth, then we would expect to see degradation of performance (more evaluations needed) with a size of 50, and improvements (fewer evaluations needed) with a size of 150, over a baseline size of 100.

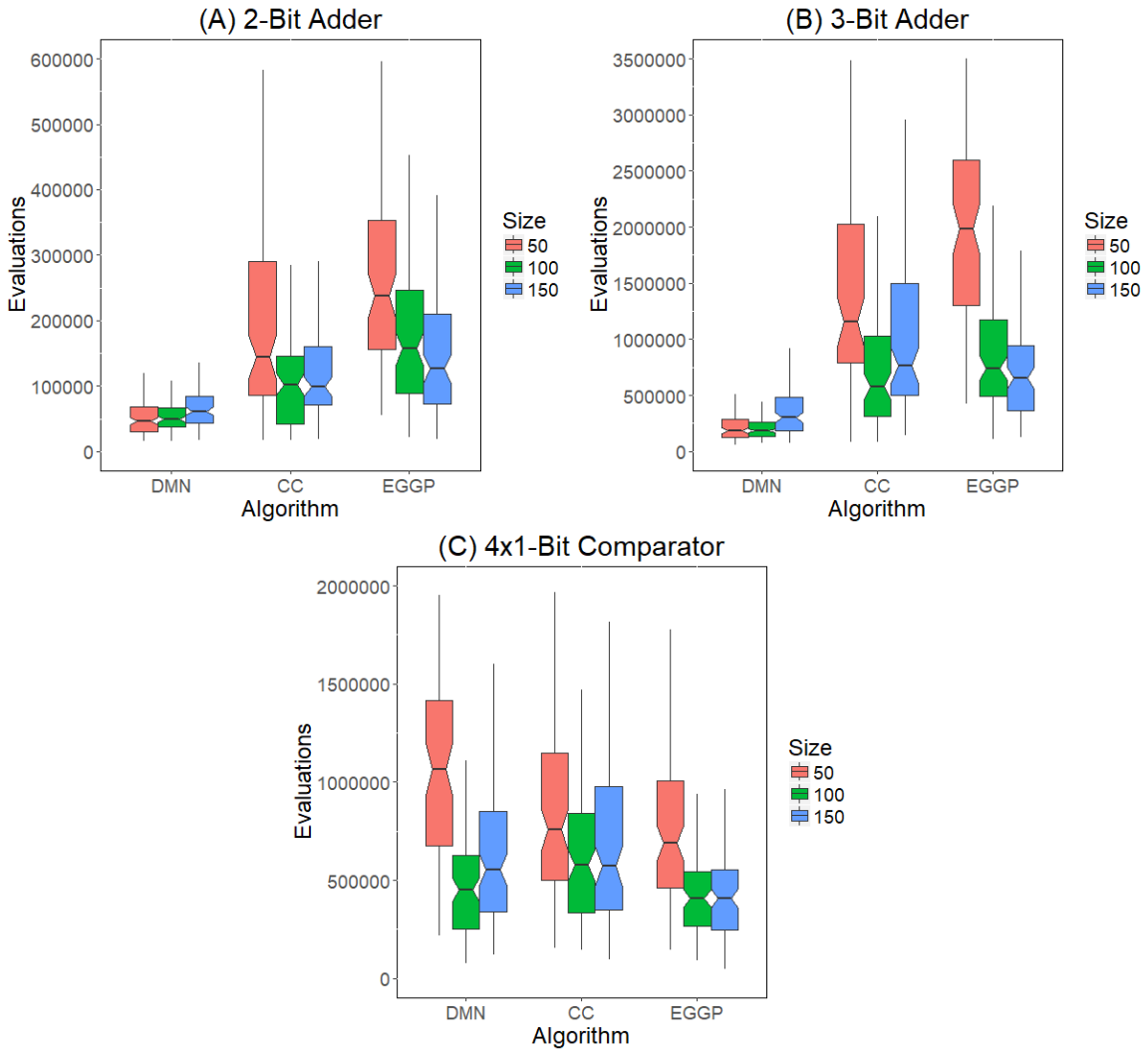


Figure 8.4: Results of running DMN, CC and EGGP on (A) 2-Add, (B) 3-Add and (C) COMP problems. The y-axis gives the MEs required to solve each problem across 100 runs. The x-axis groups setups by algorithm and then lists the observed MEs when running that algorithm with 50, 100 or 150 nodes as the fixed representation size.

The results of these runs are shown in Figure 8.4. For 2-Add and 3-Add with the DMN neutral rule-set, performance actually degrades when increasing the fixed size from 100 to 150, while remaining relatively similar when decreasing the size to 50. For EGGP alone and for the CC neutral rule-set, performance remains relatively similar when increasing the fixed

size from 100 to 150, but degrades when decreasing the size to 50. These observations imply that the DMN rule-set is not simply growing solutions to a more beneficial search space, since it performs better when limited to a smaller space. Therefore, on these problems, there is some other property of the DMN rule-set that is benefiting performance.

For the COMP problem, trends remain similar for EGGP alone and the CC neutral rule-set. However, the performance of the DMN rule-set degrades when the fixed size is decreased from 100 to 50. This suggests that the COMP problem is in some way different from the other problems. Further, when DMN is run on the COMP problem, the average proportion of active code is nearly 100%. This may offer an explanation to why the DMN rule-set struggles to outperform standard EGGP on the COMP problem, which has more than twice as many outputs (18) as the next nearest problem (8, DeMux). DMN's bias towards growth paired with the high number of outputs may give some of the problem's many outputs little room to change and configure to a correct solution.

8.6.2 DMN and ID in Combination

We investigate the effect of using DMN and ID, our two best performing neutral rule-sets, in combination. This combined set, which we refer to as DMID, consists of the following logical equivalence laws:

$$\text{DeMorgan}_{F1},\ \text{DeMorgan}_{F2},\ \text{DeMorgan}_{R1},\ \text{DeMorgan}_{R2},$$
$$\text{ID}-\text{AND}_F,\ \text{ID}-\text{AND}_R,\ \text{ID}-\text{OR}_F,\ \text{ID}-\text{OR}_R,\ \text{ID}-\text{NOT}_F \text{ and } \text{ID}-\text{NOT}_R.$$

We use this set under the same experimental conditions described in Section 8.4 to produce the results given in Table 8.6. In Table 8.6 we provide p and A values in comparison to the DMN and ID results in Table 8.4 and the EGGP results in Table 8.3.

The DMID rule-set significantly outperforms DMN on 5 of the 12 problems, and shows no significant difference for the other 7 problems. DMID significantly outperforms ID on 4 problems (notably the 2 and 3-Bit Adder problems), shows no significant difference on 3 problems, and is significantly outperformed by ID on 4 problems (notably the 3-Mul, COMP and 7-EP). DMID significantly outperforms EGGP without neutral rule-sets on all but 2 problems, with the exception being the COMP and 7-EP problems that DMN also fails to find significant benefits on. These results position DMID and ID on a Pareto front of studied problems, with DMID effectively dominating DMN but neither DMID nor ID universally outperforming each other.

Problem	DMID		vs. DMN		vs. ID		vs. EGGP	
	ME	IQR	p	A	p	A	p	A
1-Add	7,415	5,756	10^{-4}	0.64	0.02	-	10^{-12}	**0.78**
2-Add	43,633	29,065	0.13	–	10^{-8}	**0.73**	10^{-23}	**0.91**
3-Add	162,568	112,074	0.02	-	10^{-11}	**0.77**	10^{-28}	**0.95**
2-Mul	12,020	8,761	10^{-3}	0.63	0.30	–	10^{-8}	**0.73**
3-Mul	604,480	471,956	0.51	–	0.04	0.59	10^{-13}	**0.80**
DeMux	20,938	11,040	10^{-3}	0.63	10^{-6}	0.69	10^{-5}	0.68
COMP	399,140	315,459	0.45	–	10^{-4}	0.66	0.95	–
3-EP	3,930	3,105	10^{-3}	0.60	10^{-3}	0.61	10^{-7}	0.71
4-EP	16,778	10,730	0.02	-	0.13	–	10^{-9}	**0.75**
5-EP	52,868	31,445	0.29	–	10^{-3}	0.61	10^{-5}	0.66
6-EP	121,978	90,429	10^{-3}	0.61	0.11	–	10^{-6}	0.68
7-EP	326,040	224,121	0.95	–	10^{-7}	0.70	0.05	-

Table 8.6: Results from digital circuit benchmarks for the DMID neutral rule-set. The p value is from the two-tailed Mann–Whitney U test. Where $p < 0.0514$, the effect size from the Vargha–Delaney A test is shown; large effect sizes ($A > 0.71$) are shown in **bold**. Statistics are given in comparison to the DMN and ID neutral rule-sets and EGGP.

8.6.3 {AND, OR, NOT}: A Harder Function Set?

In Table 8.3 we show that solving problems with the function set {AND, OR, NOT} is significantly more difficult than when using the function set {AND, OR, NAND, NOR}. We justify using the former function set over the latter in our experiments as it lends itself to known logical equivalence laws despite costing performance. When we introduce these logical equivalence laws to the evolutionary process with the {AND, OR, NOT} function set, this 'cost' no longer universally holds. We identify 3-Add, 3-Mul, COMP and 7-EP as the 4 hardest problems, based on the MEs required to solve them, Table 8.3. EGGP with the {AND, OR, NOT} function set and augmented with the DMID neutral rule-set significantly ($p < 0.05$) outperforms EGGP with the {AND, OR, NAND, NOR} function set on two of the problems.

These two are the 3-Add ($p = 10^{-10}$, $A = 0.76$) and 3-Mul problems ($p = 10^{-5}$, $A = 0.68$).

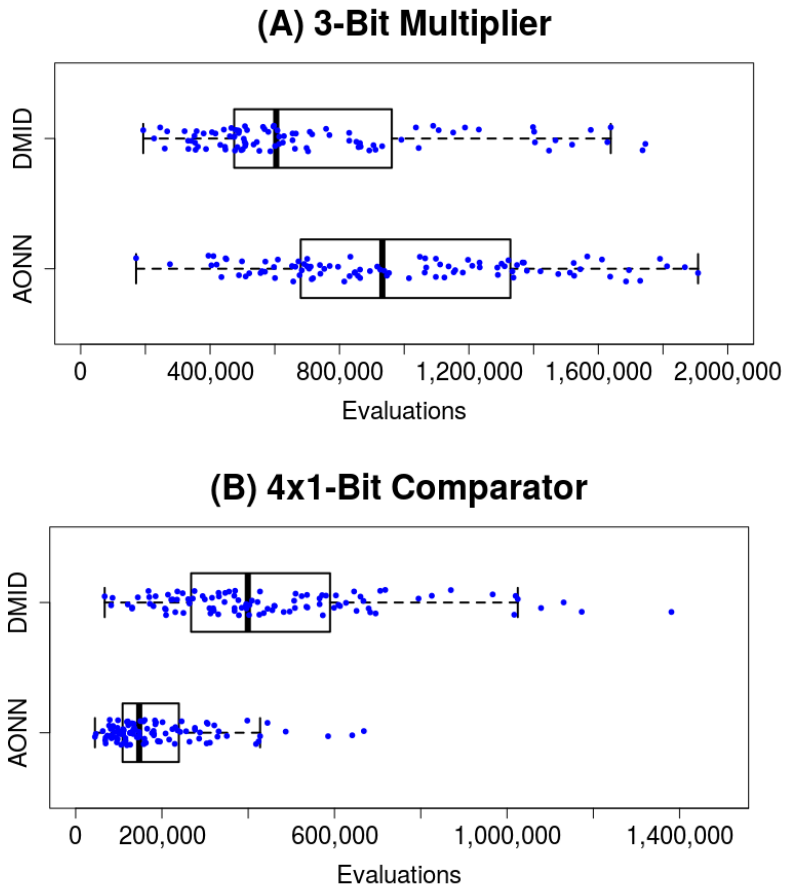


Figure 8.5: Box-plots showing observed evaluations required to solve (A) 3-Bit Multiplier and (B) 4×1-Bit COMP problems using EGGP augmented with the DMID neutral rule-set (DMID) and EGGP with the {`AND`, `OR`, `NAND`, `NOR`} function set (AONN). Vertical jitter is included for visual clarity.

In contrast, the reverse holds for COMP ($p = 10^{-18}$, $A = 0.85$) and 7-EP ($p = 10^{-14}$, $A = 0.80$). Note that for 3 of these circumstances (excluding 3-Mul), the significant difference occurs with large effect size ($A > 0.71$).

Figure 8.5 shows the number of evaluations across 100 runs for the 3-Mul and COMP problems, for (A) EGGP with the {`AND`, `OR`, `NOT`} function set and augmented with the DMID neutral rule-set and (B) EGGP with the {`AND`, `OR`, `NAND`, `NOR`} function set. Here the difference in medians and IQRs for these two EAs can be clearly seen; with EGGP with the DMID neutral rule-set requiring MEs outside of the IQR of EGGP with the {`AND`, `OR`, `NAND`,

NOR} function set for the 3-Mul problem. In stark contrast, the third quartile of evaluations required for the COMP problem lies below the first quartile of EGGP with the DMID neutral rule-set.

This offers an interesting secondary result: there are circumstances and problems where it may be beneficial to choose representations that on their own would yield detrimental results, if that decision then facilitates the inclusion of SND, which may in combination provide enhanced performance over the original representation.

8.7 Conclusions and Future Work

We have investigated the augmentation of EGGP for learning digital circuits with SND. From our experimental results, we can draw a number of conclusions both for our own specific setting and for the broader evolutionary community.

Firstly, we offer further evidence that there are circumstances where neutral drift aids evolution, building upon existing works that offer evidence in this direction. Additionally, the precise nature of our neutral drift by design offers evidence that neutral drift on the active component of individuals, rather than the intronic components, can aid evolution. For every benchmark problem studied, at least one neutral rule-set was able to yield significant improvements in performance.

Secondly, we have shown that by using graphs as a representation and graph programming as a medium for mutation, it is possible to directly inject domain knowledge into an evolutionary system to improve performance. The application of De Morgan's logical equivalence laws to graphs with sharing is non-trivial, but becomes immediately accessible in our graph evolution framework. Our ability to design complex domain-specific mutation operators supports the view that that the choice of representation of individuals in an EA matters. This injection of domain knowledge has been shown to offer benefits beyond simple 'neutral growth'.

Thirdly, while the approach we have proposed here offers promising results, the specific design of neutral drift matters. There are neutral rule-sets that appear to dominate each other, as is found comparing the DMID rule-set to the DMN rule-set. There are also neutral rule-sets which outperform each other on different problems, as is demonstrated comparing the DMID rule-set to the ID rule-set. As we highlighted in comparing DMID to EGGP with what initially appeared to be a preferential function set, there are circumstances where a GP

practitioner may want to deliberately degrade the representation in order to access beneficial neutral drift techniques. There are also other circumstances where the cost of incorporating these techniques may outweigh their immediate benefits.

There are a number of immediate extensions to our work that we believe should be investigated. Firstly, the use of the complete function set, {`AND`, `OR`, `NAND`, `NOR`, `NOT`}, alongside the DMID semantics-preserving mutations and additional mutations for converting between `AND` and `OR` gates and their negations, via `NOT`, should be investigated. It may be the case that this overall combination yields better results than either of the function sets and semantics-preserving mutations we have covered in this work. Additionally, while semantics-preserving mutations have generally improved performance with respect to the number of evaluations required to solve problems, it would be worthwhile to measure the clock-time cost of executing these transformations in every generation. Then it would be possible to study the trade-off between gained efficiency and additional overhead. Future work should also investigate the potential use of our proposed approach in CGP and TGP as discussed in Section 8.1.

While we do not address theoretical aspects of SND here, it may be possible to prove convergence of EAs equipped with SND under certain properties, such as the completeness of the semantics-preserving mutations used with respect to equivalence classes.

There are a number of application domains to investigate for future work: hard search problems where individual solutions may be represented by graphs and where there are known semantics-preserving laws. A primary candidate is the evolution of Bayesian Network topologies, a well-studied field [133], as there are known equivalence classes for Bayesian Network topologies [39]. A secondary candidate is learning quantum algorithms using the ZX-calculus, which represents quantum computations as graphs [42], and is equipped with graphical equivalence laws that preserve semantics.

9 Evolving Graphs with Horizontal Gene Transfer

Abstract

In this chapter we introduce a form of neutral Horizontal Gene Transfer (HGT) to Evolving Graphs by Graph Programming (EGGP). We introduce the $\mu \times \lambda$ Evolutionary Algorithm (EA), where μ parents each produce λ children who compete only with their parents. HGT events then copy the entire active component of one surviving parent into the inactive component of another parent, exchanging genetic information without reproduction. Experimental results from symbolic regression problems show that the introduction of the $\mu \times \lambda$ EA and HGT events improve the performance of EGGP. Comparisons with Tree-Based Genetic Programming (TGP) and Cartesian Genetic Programming (CGP) strongly favour our proposed approach. We also investigate the effect of using HGT events in neuroevolution tasks. We again find that the introduction of HGT improves the performance of EGGP, demonstrating that HGT is an effective cross-domain mechanism for recombining graphs.

Relevant Publications

Content from the following publications is used in this chapter:

[10] T. Atkinson, D. Plump, and S. Stepney, "Evolving graphs with horizontal gene transfer," in *Proc. Genetic and Evolutionary Computation Conference, GECCO 2019*, ACM, 2019, pp. 968-–976.

[12] T. Atkinson, D. Plump, and S. Stepney, "Horizontal gene transfer for recombining graphs," *Genetic Programming and Evolvable Machines,* 2020.

9.1 Introduction

Recombination of genetic material is commonly viewed as a key component of a successful Genetic Programming (GP) system. Koza [129] recommends that most offspring be produced by crossover, rather than by asexual reproduction and mutation. In contrast, Cartesian Genetic Programming (CGP) [155] traditionally uses the elitist $1+\lambda$ Evolutionary Algorithm (EA), where all offspring are produced by asexual reproduction and mutation; variation and the ability to leave local optima are a byproduct of neutral drift in the neutral parts of the genome [156].

Existing work on Evolving Graphs by Graph Programming (EGGP) has used only asexual reproduction and mutation. Here we extend EGGP to incorporate Horizontal Gene Transfer (HGT) 'events', where the genetic information of one parent is shared with another. Our system operates using the elitist '$\mu \times \lambda$' EA, such that in each generation there are μ parents, which each produce λ children, which compete only with their own parent. This is effectively μ parallel $1 + \lambda$ EAs, with genetic information shared horizontally between elite individuals. To avoid disrupting elitism (by modifying the active components of individuals) or sharing junk (by copying neutral components of individuals), we copy only the active components of one parent onto the neutral component of another; it may later be activated through mutation.

Here we replace neutral components with new material directly. This is inspired by horizontal gene transfer[1] (or lateral gene transfer) found in nature. Biological horizontal gene transfer is the movement of genetic material between individuals without mating, and is distinct from normal 'vertical' movement from parents to offspring [119]. Horizontal gene transfer plays a key role in the spread of anti-microbial resistance in bacteria [83] and evidence has been found of plant-plant horizontal gene transfer [265] and plant-animal horizontal gene transfer [200]. The mechanism of horizontal gene transfer in transferring a segment of DNA into another individual's DNA may have a clear analogy when considering bit-string based Genetic Algorithms (GAs) such as the Microbial GA [92], the equivalent analogy is not as obvious when dealing with graphs. Hence we use the term metaphorically: when we refer to HGT, we mean the movement of genetic material between individual graphs without mating. This is the new mechanism we present in this work.

Our approach is not the first work to recombine and share genetic information in graph-like programs. Parallel Distributed Genetic Programming (PDGP) uses Subgraph Active-

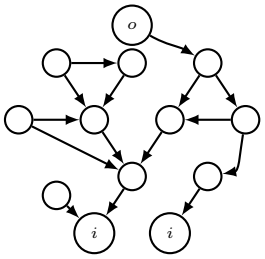
[1] Here we make a distinction between the computational and the biological.

Active Node (SAAN) crossover [188] to share material within a population of Cartesian grid-based programs. A number of crossover operators have been used in CGP, including uniform crossover [154], arithmetic crossover on a vector representation [41], and subgraph crossover [112]. Empirical comparison [106] shows that these crossover operators do not always aid performance, and that CGP with mutation only can sometimes be the best performing approach. Current advice [155, 243] is that the 'standard' CGP approach is to use mutation alone. Our recombination features no modification of active components and does not produce children; nevertheless HGT events followed by edge mutations may perform operations very similar to PDGP SAAN crossover [188] and CGP subgraph crossover [112]. However, our precise mechanism, where active components are passed into neutral components without any limitations to accessibility, does not obviously translate to PDGP and CGP, which are limited to Cartesian grids.

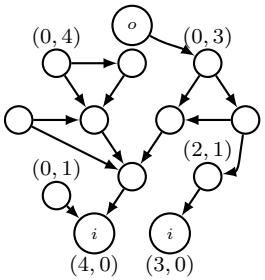
We perform comparative symbolic regression experiments and find that HGT improves performance on many of the studied problems. Comparisons with Tree-Based GP (TGP) and CGP strongly favour EGGP with HGT. We demonstrate the cross-domain effectiveness of HGT by synthesising neural networks for pole balancing problems. We find that in both Markovian and non-Markovian settings, HGT aids the efficiency of search. We strengthen the argument that HGT works across domains by deliberately choosing to evolve much smaller, more dense graphs in our neuroevolution experiments in comparison to our symbolic regression experiments.

The rest of this work is organised as follows. In Section 9.2 we introduce EGGP with a new feature: depth control. In Section 9.3 we describe our HGT approach, and the $\mu \times \lambda$ EA. In Section 9.4 we describe experimental settings for comparing our HGT approach to the existing EGGP approach, and to CGP and TGP on various symbolic regression problems. In Section 9.5 we present the results of our symbolic regression experiments. In Section 9.6 we describe the dynamics of the pole balancing problems, the genetic operators used for neuroevolution and the parameters used in these experiments. In Section 9.7 we present the results of our neuroevolution experiments. Finally, in Section 9.8, we summarise our findings and set out directions for future work.

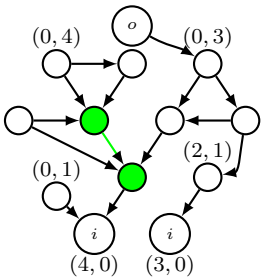
This individual is to undergo an edge mutation preserving acyclicity and a maximum depth $D = 4$.



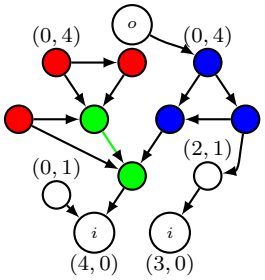
(1) The individual is annotated with depth information. Each node has an associated 'depth up' value u indicating the length of the longest path to a root node (excl. outputs), and a 'depth down' value d indicating the length of the longest path to a leaf node. These are listed as a pair (u, d) for each node.



(2) An edge to mutate is chosen at random and marked (green) alongside its source node s and target node t.



(3) Invalid candidate nodes for redirection are identified. If a node v has a directed path to s it is marked invalid (red), as targeting it would introduce a cycle. If the depth down value of a node v is d_v and the depth up value of s is u_s, when $u_s + d_v + 1 > D$, v is marked invalid (blue), as targeting it would exceed the maximum depth.



(4) The edge e (now shown in red) is mutated to target some randomly chosen unmarked (non-output) node, preserving acyclicity and maximum depth D. Finally, all annotations are removed.

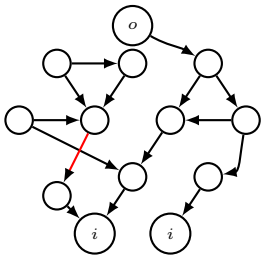


Figure 9.1: An example of edge mutation preserving acyclicity and depth. Some annotations from step (1) are omitted for visual clarity.

9.2 Depth Control

Here we introduce the notion of *depth control* to EGGP. The motivation for this is that, as we will see in Section 9.3, we have a desire to keep our solutions relatively small. Additionally, investigation into the use of depth control may help clarify our discussions of bloat in Section 6.8. Depth control prevents mutations that would cause a child to exceed a given maximum depth, D. We annotate individuals with information regarding the depth associated with each node. The 'depth up' u (or 'depth down' d) of a node is the length of the longest path from that node to a root (or leaf) node. We label each node v with the values (u, d). An exception is made for output nodes, which have $u = -1$ as their outgoing edges are not considered part of the 'depth' of the individual.

Once an individual has been annotated, we can identify pairs of nodes that, if an edge were inserted between them, would cause the individual to exceed the maximum depth, D. If we wish to insert an outgoing edge for node v_1, then we eliminate any other node v_2 as a viable candidate on the following criteria: If the depth up value of v_1 is u_1, and the depth down value of v_2 is d_2, then it is impossible to insert an edge and preserve the maximum depth D if $u_1 + d_2 + 1 > d$: we have a path of length u_1 from v_1 to a root node, and a path of length d_2 from v_2 to a leaf node, hence the overall path from a root to a leaf would be $u_1 + d_2 + 1$, which exceeds D. If $u_1 + d_2 + 1 \leq d$, inserting an edge from v_1 to v_2 would preserve D.

We use this strategy in both edge mutation and node mutation. In edge mutation, we use annotations to identify invalid targets for the mutating edge. In node mutation, we use annotations to identify invalid targets for new edges to be inserted for the mutating node. We give an example of depth preserving edge mutation in Figure 9.1; an edge of an individual is mutated, but all possible targets that would break acyclicity or a maximum depth $D = 4$ are ignored.

Note that we omit the P-GP 2 programs that do this, as the annotation programs are relatively simple and then our existing mutation operators described in Chapter 5 are modified with simple rule conditions that forbid edge re-directions or edge insertions based on these annotations as we have described.

9.3 Horizontal Gene Transfer in Evolving Graphs by Graph Programming

In this Section we describe the introduction of HGT events to EGGP. HGT events involve the transfer of active material from a donor to the neutral region of a recipient (Section 9.3.1). To accommodate the need for multiple surviving individuals, we introduce the $\mu \times \lambda$ EA (Section 9.3.2) as an alternative to the $1 + \lambda$ EA previously used in EGGP.

9.3.1 Active-Neutral Transfer

HGT involves the movement of genetic material between individuals of a population without reproduction. Given a population P, we choose a donor and recipient individual. We copy the entire active component of the donor (excluding output nodes); we remove sufficient neutral material at random from the recipient to fit this active component within the fixed representation size. The copied active component is inserted into the recipient's neutral component, where it remains neutral until it is activated by some mutation. This type of HGT, which we refer to as 'Active-Neutral Transfer', is guaranteed to preserve the fitness of both the donor and recipient, preventing it from disrupting the elitism of the EA. The intention is to promote the production of higher quality offspring by the recipient, by activating its received genetic material through mutation. This process is mutually beneficial; the donor has a mechanism for propagating its genes, while the recipient stands to improve the survivability of its offspring. Once material has been transferred, there are a number of possible consequences: the neutral donor material can drift, or become active, through mutation. In this way it is possible for processes such as SAAN crossover in PDGP [188] or block-based crossover in CGP [112] to arise out of Active-Neutral transfer followed by mutation.

Our strategy for choosing a donor and recipient is as follows. A recipient is first chosen based on a uniform distribution over the population, P, excluding the best performing member. We refer to this best performing member as the 'leader', which we exclude from receiving genetic material so that it can undergo neutral drift without any disruption. Throughout the evolutionary process, it is likely that the leader will change several times, meaning that the entire population is likely to receive genetic material at some point. Once a recipient is chosen, a donor is selected from the population excluding the recipient based on a roulette wheel. The donor may be the leader, allowing the leader to propagate its own genes to other members of the population. The use of a roulette wheel means that any individual can donate material, but the better performing individuals are more likely to do so.

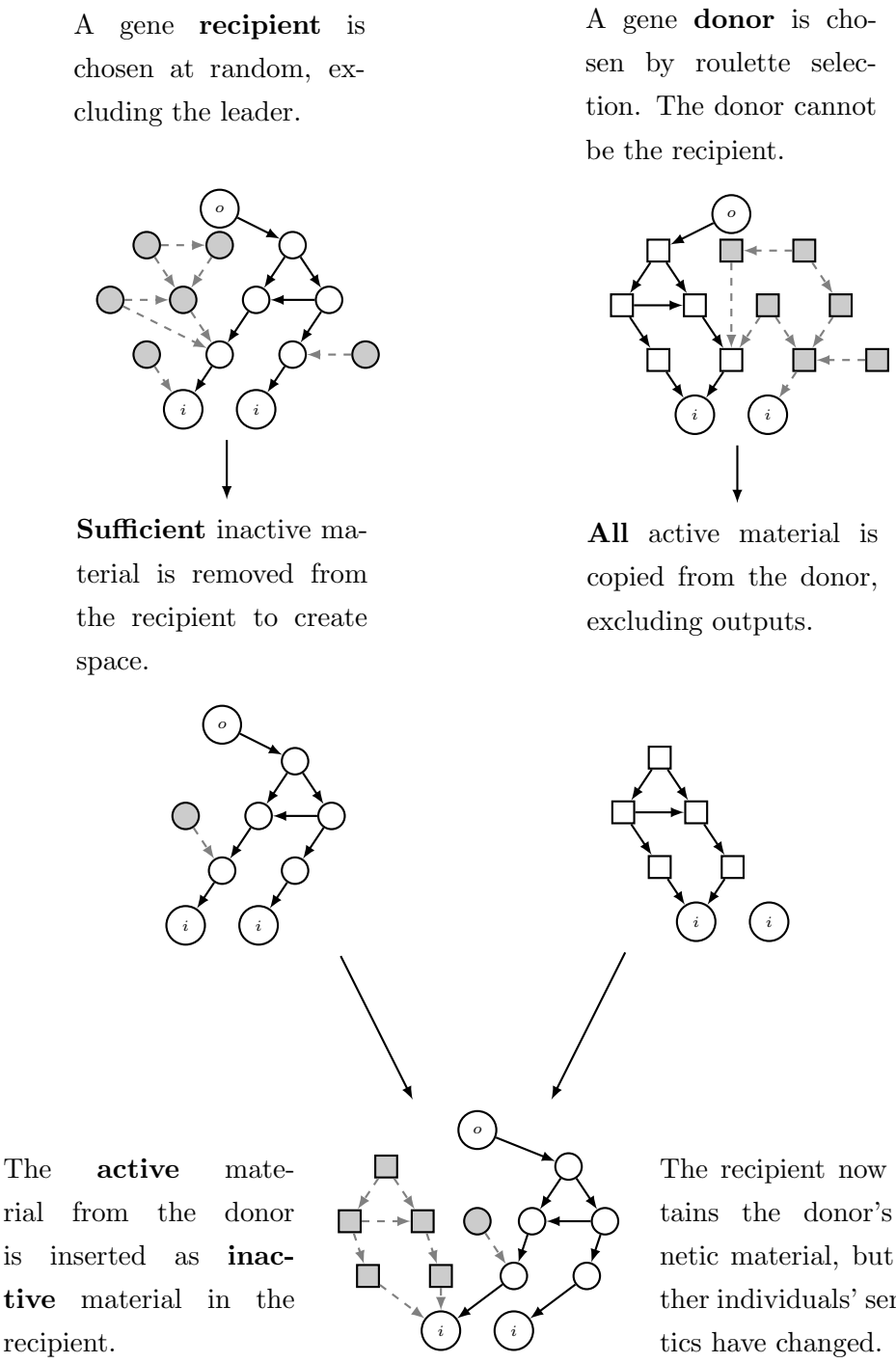


Figure 9.2: An example of Active-Neutral transfer. The active material of a donor is copied into the neutral material of a recipient. Neither individuals' semantics are changed by this process. Grey nodes and dashed edges indicate the neutral material of individuals; they do not indicate any actual information stored on the individual. The donor's function nodes are shown as squares for clarity.

We give an example of Active-Neutral transfer in Figure 9.2. The entire active component of a gene donor is copied into the neutral material of the recipient while maintaining the overall representation size.

Again note that we do not include the P-GP 2 programs to execute the Active-Neutral transfer event. The reason for this is that these programs are quite simple and easy to conceptualise. Firstly, a program is applied to the donor, removing outputs and inactive material. This can be done with a simple depth-first search from the outputs of the donor to identify the active components of the donor. Similarly, a program is applied to the recipient, removing inactive material equal to the active components of the donor. The number of nodes to remove can be calculated ahead-of-time, again by a depth first search from the outputs of the donor. Finally, the two graphs are merged via a disjoint union, which is not a native feature of P-GP 2 and must be implemented externally, and a final program is applied which merges the input nodes so that there are no duplicates.

We take the view that Active-Neutral transfer is distinct from traditional recombination operators. While both approaches map a pair of individuals to a single individual, the intention and behaviour are different. In recombination it is hoped that a child will be produced which is an approximate midpoint between its parents, introducing *immediate variation* to the search process. In contrast, our Active-Neutral transfer operator *does not vary* the gene recipient but instead biases *future* mutations to promising areas of the landscape.

9.3.2 The $\mu \times \lambda$ Evolutionary Algorithm

We cannot use Active-Neutral transfer with the $1 + \lambda$ algorithm except for sharing genetic material between the offspring; this is likely to be ineffective as direct offspring have much material in common. We therefore introduce the $\mu \times \lambda$ EA, a special case of the $\mu + \lambda$ EA. In each generation of the $\mu \times \lambda$ EA, there are μ parents. Each of the μ parents generates λ offspring, and compete for survival only with their own offspring. Without HGT, this effectively creates multiple parallel $1 + \lambda$ algorithms. The same cannot be said of a $\mu + \lambda$ EA, where children of one of the μ parents may replace *any* of the parents in the next generation.

In each generation we perform a single Active-Neutral transfer operation with probability p_{HGT}. We then follow the procedure set out in Section 9.3.1 by selecting a gene recipient from the μ parents (ignoring the best performing parent, the 'leader') and selecting a donor from the remaining $\mu - 1$ parents by roulette selection.

9.4 Symbolic Regression Experiments

To evaluate the effect of HGT, we return to the 14 symbolic regression problems studied in Chapter 6. We choose these problems as EGGP did not particularly outperform TGP or CGP on them as described in Section 6.7. In comparison we already have effective digital circuit synthesis and have found that EGGP outperforms CGP on digital circuit benchmarks as described in Section 6.4; we therefore have less motivation to improve upon these results.

We compare EGGP_{HGT} to: standard EGGP; the depth control variant EGGP_{DC}; the depth control variant using the $\mu \times \lambda$ EA (and no HGT), $\text{EGGP}_{\mu\times\lambda}$. These experiments allow us to test the following null hypotheses:

- H_1: there are no statistical differences when using the depth control variant EGGP_{DC} in comparison to standard EGGP.
- H_2: there are no statistical differences when using the $\mu\times\lambda$ EA for EGGP in comparison to the $1 + \lambda$ EA, with both approaches using depth control.
- H_3: there are no statistical differences when using the HGT approach for EGGP in comparison to using the $\mu \times \lambda$ EA without HGT, with both approaches using depth control.
- H_4: there are no statistical differences when using the HGT approach for EGGP in comparison to standard EGGP.

We test these null hypotheses for each benchmark problem. From these tests, we build an image of how the various features contribute to the performance of EGGP_{HGT}, and clarify whether the added HGT feature is truly improving performance by isolating it from the other new features.

We also compare our HGT approach to two other approaches from the literature: TGP [129] and CGP [155]. These experiments allow us to test the following null hypotheses:

- H_5: there are no statistical differences when using EGGP_{HGT} in comparison to GP.
- H_6: there are no statistical differences when using EGGP_{HGT} in comparison to CGP.

Again, we test each of these null hypotheses for each benchmark problem. H_5 and H_6 allow us to measure the progress made by introducing HGT to EGGP in comparison to other approaches in literature.

9.4.1 Experimental Settings

We again evaluate all individuals using the Mean Square Error (MSE) fitness function.We measure statistics taken over 100 independent runs of each approach on each dataset.

For all EGGP variants, we use a fixed 100 nodes and a mutation rate $m_r = 0.03$. For EGGP and EGGP$_{DC}$ we use the $1 + \lambda$ EA with $\lambda = 4$; for EGGP$_{\mu\times\lambda}$ and EGGP$_{HGT}$ we use $\mu = 3$ and $\lambda = 1$. This induces a 'minimal' version of the $\mu \times \lambda$ EA with $\mu = 3$ being the minimal value we could choose for μ such that HGT occurs not only from the 'leading' thread, but also between threads, and $\lambda = 1$ being the minimal value for λ. For EGGP$_{DC}$, EGGP$_{\mu\times\lambda}$ and EGGP$_{HGT}$ we set the maximum depth, $D = 10$, and limit the maximum size to 50 active nodes. The maximum active size is ensured by removing and replacing any generated individual that exceeds the maximum size; it is necessary to prevent errors in the HGT approach where, for example, the size of the donor's active component exceeds that of the recipient's neutral component (causing the overall number of nodes to grow when copying the entire active component over). In practice, this condition is used in very few instances, as depth control constrains the size. The rate p_{HGT} is 0.5.

For TGP and CGP we follow the experimental conditions described in Section 6.6. For CGP, we use 100 fixed nodes, and a mutation rate of 0.03. We use the $1 + \lambda$ EA with $\lambda = 4$. We do not use any of the published CGP crossover operators. We also use no form of depth control with CGP, as the approach is known to have inherent anti-bloat biases [237].

For TGP, the population size is 500, with 1 elite individual surviving in each generation. Subtree crossover is used with a probability of 0.9, and when it is not used, the 'depth steady' subtree replacement mutation operator is used which, when replacing a subtree of depth d, generates a new subtree of depth between 0 and d [169]. Tournament selection is used to select reproducing individuals, with a tournament size of 4, and the maximum depth allowed of any individual is 10. We add each new individual to the population one-by-one, discarding one of the children produced by each crossover operator. This allows us to immediately replace invalid individuals with respect to the maximum depth, guaranteeing that every individual in a new population is valid and should be evaluated. To initialise the population, we use the ramped half-and-half technique [129], with a minimum depth of 1 and a maximum depth of 5.

For all experiments, the maximum number of evaluations allowed is 24 950. In TGP this is achieved by allowing the search to run for 50 generations. In EGGP and CGP, we use the optimisation from [155, Ch.2], where individuals are evaluated only when their active

components are mutated; there is no fixed number of mutations, and the search continues until the total number of evaluations is performed. There is no analogous optimisation for TGP, as TGP individuals contain no neutral material. Again, we stress that this optimisation makes a large difference to the depth of search; for example, in CGP running on F_1, the median number of generations is 12 385, but if all individuals are evaluated (rather than only those with active region mutations), the number of generations would be capped at 6237 (assuming elite individuals are never re-evaluated).

9.4.2 Implementation

Our CGP experiments are based on the publicly available CGP library [243] with modifications made to accommodate the 'active evaluations only' optimisation and the use of validation and training sets. Our TGP experiments are based on the Distributed Evolutionary Algorithms in Python (DEAP) evolutionary computation framework [70] with modifications made to accommodate our crossover strategy, mutation operator, and use of validation and training sets.

9.5 Symbolic Regression Results

Table 9.1 lists the Median Fitness (MF) and Interquartile Range (IQR) of each approach on each dataset over 100 runs. Overall, the lowest MF score is achieved by EGGP$_{HGT}$ in 10 cases, EGGP$_{DC}$ in 2 cases and TGP in 2 cases. There are no cases where EGGP, EGGP$_{\mu\times\lambda}$ or CGP achieve the lowest MF score.

To test for statistical significance we use the two-tailed Mann–Whitney U test. We use a significance threshold of 0.05 and perform a Bonferroni procedure for each hypothesis giving a corrected significance threshold of $\alpha = \frac{0.05}{14}$. Where we get a statistically significant result ($p < \alpha$), we also calculate the effect size, using the non-parametric Vargha–Delaney A Test. $A \geq 0.71$ corresponds to a large effect size. These results of these statistical tests for all hypotheses are given in Table 9.2.

9.5.1 Building EGGP$_{HGT}$: H_1, H_2, H_3, H_4

The introduction of depth control (H_1) appears to have relatively little effect and is sometimes detrimental. In 12 of our benchmark problems, we observe no significant difference when introducing the feature. On 2 problems, standard EGGP achieves a statistically significant

	EGGP		**EGGP**$_{DC}$		**EGGP**$_{\mu\times\lambda}$		**EGGP**$_{HGT}$		**TGP**		**CGP**	
F	MF	IQR	MF	IQR	MF	IQR	MF	IQR	MF	IQR	MF	IQR
F1	4.45E-3	7.35E-3	6.26E-3	6.45E-3	3.59E-3	1.39E-3	**2.47E-3**	1.79E-3	5.77E-3	3.40E-3	6.74E-3	4.30E-3
F2	8.17E6	6.05E6	1.41E7	9.95E6	8.06E6	5.02E6	**5.94E6**	3.06E6	1.28E7	7.86E6	1.73E7	2.54E6
F3	1.18E-2	7.34E-3	1.48E-2	4.27E-3	9.92E-3	3.82E-3	**7.22E-3**	4.00E-3	1.04E-2	3.56E-3	1.48E-2	4.39E-3
F4	2.58E13	1.05E9	**2.58E13**	3.57E8	2.58E13	7.51E10	2.58E13	1.96E9	3.55E13	8.35E13	2.58E13	2.35E9
F5	3.96E0	3.56E0	4.48E0	4.30E0	2.30E0	2.61E0	**6.90E-1**	2.08E0	5.13E0	3.81E0	7.17E0	1.47E0
F6	1.69E1	2.24E1	2.11E1	3.99E1	7.23E0	1.18E1	4.46E0	6.24E0	**2.61E0**	6.86E0	9.28E0	2.03E1
F7	3.06E2	7.40E2	4.16E2	6.76E2	2.20E2	1.53E2	**1.51E2**	9.62E1	4.20E2	3.50E2	5.76E2	4.39E2
F8	3.91E-2	7.43E-2	1.03E-1	1.13E-1	2.85E-2	2.00E-2	**2.19E-2**	1.21E-2	1.09E-1	4.99E-2	4.49E-2	9.59E-2
F9	7.09E2	5.40E3	2.59E3	1.36E4	1.81E2	3.68E2	1.57E2	3.53E2	**1.46E2**	3.04E1	1.71E2	1.11E3
F10	1.52E-1	2.05E-1	2.36E-1	2.22E-1	1.07E-1	8.30E-2	**7.69E-2**	5.75E-2	3.22E-1	5.62E-2	1.66E-1	1.42E-1
F11	3.93E1	7.26E1	4.53E1	6.33E1	2.43E1	1.37E1	**1.59E1**	1.20E1	3.88E1	3.37E1	4.96E1	4.73E1
F12	1.21E3	5.25E2	1.22E3	5.20E2	6.95E2	1.19E2	**6.83E2**	1.44E2	1.25E3	5.02E1	7.08E2	5.19E2
F18	4.07E4	9.27E3	4.08E4	3.91E4	4.40E3	3.86E4	**3.69E-1**	2.07E4	4.13E4	3.54E2	1.20E2	4.10E4
F21	1.07E0	6.16E-4	**1.07E0**	1.38E-5	1.07E0	7.74E-4	1.07E0	6.88E-4	1.07E0	4.90E-4	1.07E0	1.53E-5

Table 9.1: Results from symbolic regression benchmarks as described in Section 9.4. MF indicates the **M**edian **F**itness over observed runs; the lowest (best) MF result across all algorithms is highlighted in **bold**. IQR indicates the Inter-quartile range in fitness.

	H_1		H_2		H_3		H_4		H_5		H_6	
F	p	A	p	A	p	A	p	A	p	A	p	A
F1	0.08	-	$< \alpha$	**0.76**	$< \alpha$	**0.71**	$< \alpha$	**0.76**	$< \alpha$	**0.92**	$< \alpha$	**0.91**
F2	$< \alpha$	0.70	$< \alpha$	**0.76**	$< \alpha$	0.68	$< \alpha$	**0.71**	$< \alpha$	**0.87**	$< \alpha$	**0.95**
F3	$< \alpha$	0.68	$< \alpha$	**0.82**	$< \alpha$	0.70	$< \alpha$	**0.72**	$< \alpha$	**0.75**	$< \alpha$	**0.91**
F4	0.98	-	0.33	-	0.08	-	0.52	-	$< \alpha$	0.68	0.89	-
F5	0.06	-	$< \alpha$	**0.76**	$< \alpha$	0.70	$< \alpha$	**0.84**	$< \alpha$	**0.86**	$< \alpha$	**0.99**
F6	0.26	-	$< \alpha$	**0.78**	$< \alpha$	0.63	$< \alpha$	**0.84**	0.37	-	$< \alpha$	0.63
F7	0.12	-	$< \alpha$	**0.74**	$< \alpha$	**0.71**	$< \alpha$	**0.76**	$< \alpha$	**0.93**	$< \alpha$	**0.94**
F8	$\geq \alpha$	-	$< \alpha$	**0.75**	$< \alpha$	0.62	$< \alpha$	**0.77**	$< \alpha$	**0.95**	$< \alpha$	**0.79**
F9	0.02	-	$< \alpha$	**0.78**	0.77	-	$< \alpha$	0.69	0.23	-	0.17	-
F10	0.01	-	$< \alpha$	**0.74**	$< \alpha$	0.65	$< \alpha$	**0.76**	$< \alpha$	**0.99**	$< \alpha$	**0.81**
F11	0.57	-	$< \alpha$	**0.76**	$< \alpha$	**0.73**	$< \alpha$	**0.85**	$< \alpha$	**0.90**	$< \alpha$	**0.89**
F12	0.85	-	$< \alpha$	**0.76**	0.12	-	$< \alpha$	**0.81**	$< \alpha$	**0.89**	0.15	-
F18	0.84	-	$< \alpha$	**0.71**	$< \alpha$	0.68	$< \alpha$	**0.85**	$< \alpha$	**0.91**	$< \alpha$	0.62
F21	$\geq \alpha$	-	$< \alpha$	0.66	0.11	-	0.57	-	0.32	-	$< \alpha$	0.62

Table 9.2: Statistical tests for hypotheses H_1 - H_6. The p value is from the two-tailed Mann–Whitney U test. The corrected threshold for statistical significance is $\alpha = \frac{0.05}{14}$. Where $p < \alpha$, the effect size from the Vargha–Delaney A test is shown; large effect sizes ($A > 0.71$) are shown in **bold**. Where $\alpha \leq p < 0.005$, p is listed as $\geq \alpha$.

lower (better) median fitness than EGGP$_{DC}$, but never with large effect. These results indicate that depth control is not necessarily a helpful feature for EGGP, but never causes EGGP to outperform EGGP$_{DC}$ with large effect, and in many cases makes no significant difference to performance. This implies that the performance of EGGP$_{HGT}$ (discussed later) cannot be explained by its new depth control feature alone. We suggest that these results may be due to neutral material contributing to active nodes' 'depth up' values, preventing the active component from undergoing certain mutations even if these mutations would produce an active component of a valid depth. There may be circumstances where this restriction of the landscape hinders the performance of EGGP$_{DC}$.

Comparing EGGP$_{\mu\times\lambda}$ and EGGP$_{DC}$ (H_2) we find that the introduction of the $\mu \times \lambda$ EA yields a statistically significant lower median fitness and a large effect size on 12 of the 14 problems. On 1 problem (F4) there is no significant difference, and on 1 problem (F21)

EGGP$_{DC}$ achieves a statistically significant lower median fitness, but without large effect. Overall, our study of H_2 provides substantial evidence that the $\mu \times \lambda$ EA aids the performance of EGGP, and should potentially be adopted generally.

The differences between EGGP$_{HGT}$ and EGGP$_{\mu\times\lambda}$ (H_3) are more subtle than the comparison of H_2, but there is a prevalent trend. The introduction of HGT yields a statistically significant lower median fitness in 10 problems, 3 of which occur with large effect, and no significant differences on the other 4. These results suggest that HGT is, generally, a beneficial feature capable of yielding major differences in performance. We observe no instances where HGT leads to a significant decrease in performance.

Overall, the results from studying our hypotheses H_1, H_2 and H_3 allow us to explain the success of EGGP$_{HGT}$ in comparison to TGP and CGP (discussed in Section 9.5.2) as a composition of the core EGGP approach, the use of the $\mu \times \lambda$ EA and the introduction of Active-Neutral HGT events. Each of our 3 new features has been added to our approach in isolation, allowing us to isolate the beneficial properties of $\mu \times \lambda$ and HGT events. The role of depth control remains unclear; alone, it appears to be unhelpful but may interact with the HGT process with respect to maintaining smaller individuals. An extended investigation into the role of depth control in our designed approach is desirable in the future.

H_4 compares our final proposed approach, EGGP$_{HGT}$, to our original EGGP approach. The proposed approach achieves a statistically significant lower median fitness in 12 of the 14 problems; 11 of which occur with large effect. On the 2 remaining problems, we observe no significant differences. Therefore the combination of our 3 features – depth control, $\mu \times \lambda$ and HGT – lead to a marked improvement over standard EGGP for the studied problems.

9.5.2 EGGP$_{HGT}$ vs. TGP & CGP: H_4, H_6

EGGP$_{HGT}$ achieves a statistically significant lower median fitness in comparison to TGP (H_5) on 11 problems, 10 of which show a large effect. On the other 3 problems, we observe no statistical differences. On a clear majority of the studied problems, EGGP$_{HGT}$ significantly outperforms a standard TGP system, and is never outperformed by that TGP system.

EGGP$_{HGT}$ achieves a statistically significant lower median fitness in comparison to CGP (H_6) on 11 problems, 9 of which show a large effect. On 3 of the other 4 problems, there is no significant difference, and on only 1 problem (F21) is there a statistical difference favouring CGP, but without large effect. Hence we have EGGP$_{HGT}$ significantly outperforming CGP under similar conditions on a majority of benchmark problems, and was itself only

outperformed on 1 problem.

Collectively, these results place EGGP$_{HGT}$ favourably in comparison to the literature. Although our experiments are not exhaustive – they are not the product of full parameter sweeps, but rather are testing approaches under standard conditions – they demonstrate that EGGP with HGT is a viable and competitive approach for symbolic regression problems.

9.6 Neuroevolution Experiments

We also evaluate the HGT mechanism for a very different class of graphs; Artificial Neural Networks (ANNs). With small modifications, our EGGP system and our HGT mechanism together form a neuroevolution system.

There are a number of significant differences between the types of graphs we are studying in this section and those of the previous symbolic regression experiments. Firstly, the FGs we study in this section utilise the full FG representation, with recurrence, weights and biases. Secondly, the graphs seen in the previous experiments have a large number of nodes (100) and are relatively sparse (1-2 edges per node). In comparison the graphs in these experiments have less nodes (10) but are much more dense (10 edges per node). In Section 9.6.1 we explain the Pole Balancing Benchmark problems we study. In Section 9.6.1 we describe our experimental configuration.

9.6.1 Pole Balancing Benchmarks

Pole balancing problems have a long and extensive history of use as benchmarking problems for neural network training. The form of problem we use here is described in detail in [258]. The main concept of a pole balancing problem is that there exists a cart upon which N poles are attached. The cart is restricted to moving left or right along a single dimension of a 2-dimensional plane, and its movements, alongside gravity, affect the angles of the poles with respect to the vertical. If any of the poles fall outside a certain angle from the vertical, or if the cart moves beyond a certain distance from its starting point, the simulation is considered a failure. The neural network being evaluated controls the cart by applying horizontal forces to it. This enables the network to accelerate the cart to the left or the right, thereby balancing the poles and keeping the cart within a given distance from its starting points. The equations of motion governing the dynamics of the N-pole pole balancing problem are as follows:

The displacement of the cart from the origin, 0, is x and we denote the cart's velocity and

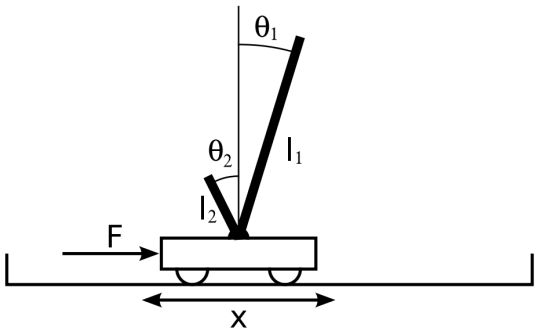


Figure 9.3: Pole balancing simulations. Figure taken from [128]

acceleration by $\dot{x}$ and $\ddot{x}$, respectively. The acceleration of the cart may be calculated by

$$\ddot{x} = \frac{F - \mu_c sign(\dot{x}) + \Sigma_{i=1}^{N} \tilde{F}_i}{M + \Sigma_{i=1}^{N} \tilde{m}_i}, \tag{9.1}$$

where we have introduced the effective force, $\tilde{F}_i$, associated with the ith pole, given by

$$\tilde{F}_i = m_i l_i \dot{\theta}_i^2 \sin\theta_i + \frac{3}{4}\cos(\theta_i)(\frac{\mu_{pi}\dot{\theta}_i}{m_i l_i} + g\sin\theta_i), \tag{9.2}$$

and the effective mass $\tilde{m}_i$ associated with the ith pole, given by

$$\tilde{m}_i = m_i \left(1 - \frac{3}{4}\cos^2\theta_i\right), \tag{9.3}$$

where $i = 1, \ldots, N$.

Once the cart's acceleration, $\ddot{x}$, has been calculated, it is then possible to calculate the angular acceleration of the ith pole. We denote the angle of each pole by θ_i, measured in radians, with 0 being vertical. Thus $\dot{\theta}_i$ is the angular velocity of the ith pole, and $\ddot{\theta}_i$ is the angular acceleration of the ith pole, with the latter is given by

$$\ddot{\theta}_i = -\frac{3}{4l_i}\left(\ddot{x}\cos(\theta_i) + g\sin\theta_i + \frac{\mu_{pi}\dot{\theta}_i}{m_i l_i}\right). \tag{9.4}$$

In our experiments we consider 2-pole problems such that $N = 2$. Variables used in these equations are listed in Table 9.3. Constants used in these equations are listed in Table 9.4 In general, we take constant values from [80].

The initial configuration and simulation of the system is taken from [80]. This is done to maximise the strength of our comparisons with other approaches from the literature; a

Symbol	Units	Description
x	m	Horizontal displacement of the cart from 0.
$\dot{x}$	m/s	Velocity of the cart.
$\ddot{x}$	m/s^2	Acceleration of the cart.
θ_i	rad	Angle of the ith pole from vertical.
$\dot{\theta}_i$	rad/s	Angular velocity of the ith pole.
$\ddot{\theta}_i$	rad/s^t	Angular acceleration of the ith pole.
F	N	The force applied to the cart by the controller.

Table 9.3: Variables used in pole balancing experiments.

Symbol	Value (units)	Description
μ_c	5×10^{-4} (-)	Friction between the cart and the track.
μ_{pi}	$\mu_{p1} = \mu_{p2} = 2 \times 10^{-6}(-)$	Friction between the ith pole and the cart.
M	1.0 (kg)	Mass of the cart.
m_i	$m_1 = 0.1, m_2 = 0.01$ (kg)	Mass of the ith pole.
l_i	$l_1 = 0.5, l_2 = 0.05$ (m)	Length of the ith pole.
g	-9.81 (m/s^2)	Acceleration due to gravity.

Table 9.4: Constants used in pole balancing experiments. These values are taken from [80].

number of techniques are evaluated on these tasks in [80]. The initial state of the system is defined by

$$x = 0, \quad \dot{x} = 0, \quad \theta_1 = \frac{4\pi}{180}, \quad \dot{\theta}_1 = 0, \quad \theta_2 = 0, \quad \dot{\theta}_2 = 0. \tag{9.5}$$

The cart starts in the centre of the track with the 1st, longer pole 4 degrees from vertical, and the 2nd, shorter pole inline with the vertical. The limits, beyond which a simulation ends, are that displacement x is bounded to the range $[-2.4, 2.4]$ and that both pole angles, θ_1 and θ_2, are bounded to the ranges $[\frac{-36\pi}{180}, \frac{36\pi}{180}]$, i.e., they cannot fall beyond 36 degrees from the vertical. The system is simulated using the 4th order Runge–Kutta approximation and a time-step of 0.1s. The neural network is updated every 2 time steps, and its output is scaled to the range $[-10, 10]$N which is then used as the force F applied to the cart. A solution is considered successful if it is able to keep both poles upright, and the cart within the bounds of the track, for $100,000$ simulated time-steps. Otherwise, the fitness assigned to a network

is equal to $100,000$ minus the number of time steps the network was able to keep the poles upright and the cart within the track. We are therefore minimising the fitness value, and the evolutionary run will successfully terminate once we find a network with a fitness of 0.

In our experiments, we study 2 problems; Markovian and non-Markovian. In the Markovian case, the network is presented with the full state of the system, with 6 input variables made up of the position and velocity of the cart and the angles and angular velocities of both poles. In the non-Markovian case, the network is only presented with the position of the cart and the angles of both poles. The latter problem is generally believed to be more difficult as it requires the network to internally account for the velocities of the cart and the poles based on observations. We rescale these values to present to the neural network, by dividing x by 1.2, $\dot{x}$ by 1.5, each θ_i by $\frac{36\pi}{180}$ and each $\dot{\theta}_i$ by $\frac{115\pi}{180}$.

9.6.2 Representation and Genetic Operators

The FGs we study in this section are similar to the cyclic FGs studied in Chapter 7, except that their edges are also labelled with weights, which are represented as integers and converted to rationals by dividing by 1000, as was done in the neural network examples of Section 4.2.4. We make further simplifications to the networks by preventing direct loops and ignoring node biases (all of which are assumed to be 0).

Our topological operators are the same as those used in Chapter 7 with minor modifications to prevent direct looping edges. We therefore use edge mutation, which may produce recurrent edges with probability p_{rec}. We fix our nodes' functions to be the bi-sigmoidal activation function given by

$$bisig(x) = \frac{1 - e^{-x}}{1 + e^{-x}}, \tag{9.6}$$

and therefore do not require function mutations. We do, however, require new mutation operators to modify weights. This is implemented with a single-rule P-GP 2 program that matches an edge uniformly at random and rewrites its weight to a uniformly chosen value from the specified weight range using the `rand_int` syntax. We can therefore distinguish between mutation rates; edge redirections may be applied according to a binomial distribution with edge mutation rate m_{re}, and weight mutations may be applied according to a binomial distribution with weight mutation rate m_{rw}.

For HGT to be viable we require that the number of active function nodes in solutions be at most half the total function nodes. However, we find that when initialising our relatively dense neural networks with recurrent connections, it may take exceptionally long to find a

viable starting point that satisfies this constraint. We therefore modify the initialisation procedure of Chapter 7. Firstly, we modify it to take into account our new restriction of no direct looping edges. Secondly, recurrent edges are added immediately after nodes are added, rather than after all nodes are added. This change reduces the average size of generated individuals, making our implementation more viable, but also prevents cycles from existing in the initial graphs. Cycles can be introduced throughout the evolutionary process via mutation.

9.6.3 Experimental Settings

We deliberately choose representation parameters that cause the graphs we study here to be topologically distinct from the graphs we have studied for symbolic regression in Section 9.4. By doing this, we further verify HGT as a cross-domain technique that is applicable in a variety of scenarios.

We use a fixed representation size of 10 nodes, with a maximum permitted number of active nodes of 5. Hence, in terms of the number of function nodes, the graphs we study here are much smaller than the 100-function node graphs we studied earlier. Each function node has an arity of 10, that is, there are 10 connections per neuron. Therefore the graphs we study here are significantly more dense, with respect to the number of edges, than the graphs we studied earlier where each function node had 1 or 2 outgoing edges. We are learning potentially cyclic graphs with recurrent edges, and set the probability of recurrent edges, $p_{rec} = 0.1$. In contrast, the graphs studied earlier were acyclic. Finally, our edges are associated with weights, with a weight range of $[-2.0, 2.0]$. In contrast, the edges we studied earlier did not feature edge weights. Overall, the graphs we study in these experiments are distinct from those studied in Section 9.4 in that they are much smaller, much more dense, may contain recurrent edges and cycles and also utilise edge weights.

In all experiments we again use the $\mu \times \lambda$ EA with $\mu = 3$ and $\lambda = 1$. Whenever we generate an individual that exceeds the permitted size of 5, we discard it and immediately generate a new one. We set the edge mutation rate $m_{re} = 0.05$, and the weight mutation rate $m_{rw} = 0.1$. We find that very occasional runs take a long time to terminate due to local optima. This is likely because of the small representation size that we have deliberately opted for, which allows for very little inactive material. To make our experiments computationally tractable while still having every evolutionary run terminate, we therefore introduce a restarting procedure; if an evolutionary run has not seen improvement in 1000 generations, its population is randomised.

	$\mathbf{EGGP}_{HGT}$		$\mathbf{EGGP}_{\mu\times\lambda}$			
Problem	ME	IQR	ME	IQR	p	A
Markovian	812	848	1,194	1,478	10^{-4}	0.61
Non-Markovian	6,230	8,928	10,577	17,074	10^{-6}	0.63

Table 9.5: Results from pole balancing benchmarks for EGGP_{HGT} and $\text{EGGP}_{\mu\times\lambda}$. The p value is from the two-tailed Mann–Whitney U test. The effect size A from the Vargha–Delaney A test is shown.

We study 2 variants of EGGP:

1. EGGP_{HGT} is the $\mu \times \lambda$ EA with HGT as described and $p_{HGT} = 1$.
2. $\text{EGGP}_{\mu\times\lambda}$ is simply the $\mu \times \lambda$ EA without HGT. This variant is used as a control for HGT.

We run each algorithm on each problem 200 times. These experiments allow us to test the null hypotheses that there are no statistical differences when using the HGT mechanism in comparison to the $\mu \times \lambda$ EA alone. We carry out statistical tests to test for significant differences introduced by the HGT mechanism on the studied problems. If our statistical tests reject the null hypothesis, and we see lower Median Evaluations (MEs) required for each problem when using HGT, then we can infer that the HGT mechanism is indeed improving performance for these neuroevolution tasks.

9.7 Neuroevolution Results

The results from our neuroevolution experiments are given in Table 9.5. For each problem and algorithm, we list the MEs and IQRs in evaluations. To test for statistical significance we use the two-tailed Mann–Whitney U test [147], which (essentially) tests the null hypothesis that two distributions have the same medians. We use a significance threshold of 0.05 and perform a Bonferroni procedure for each hypothesis giving a corrected significance threshold of $\alpha = \frac{0.05}{2}$. Where we get a statistically significant result ($p < \alpha$), we also calculate the effect size, using the non-parametric Vargha–Delaney A Test [248]. $A \geq 0.71$ corresponds to a large effect size.

As we can see in Table 9.5, on both problems we record lower MEs for EGGP_{HGT} in

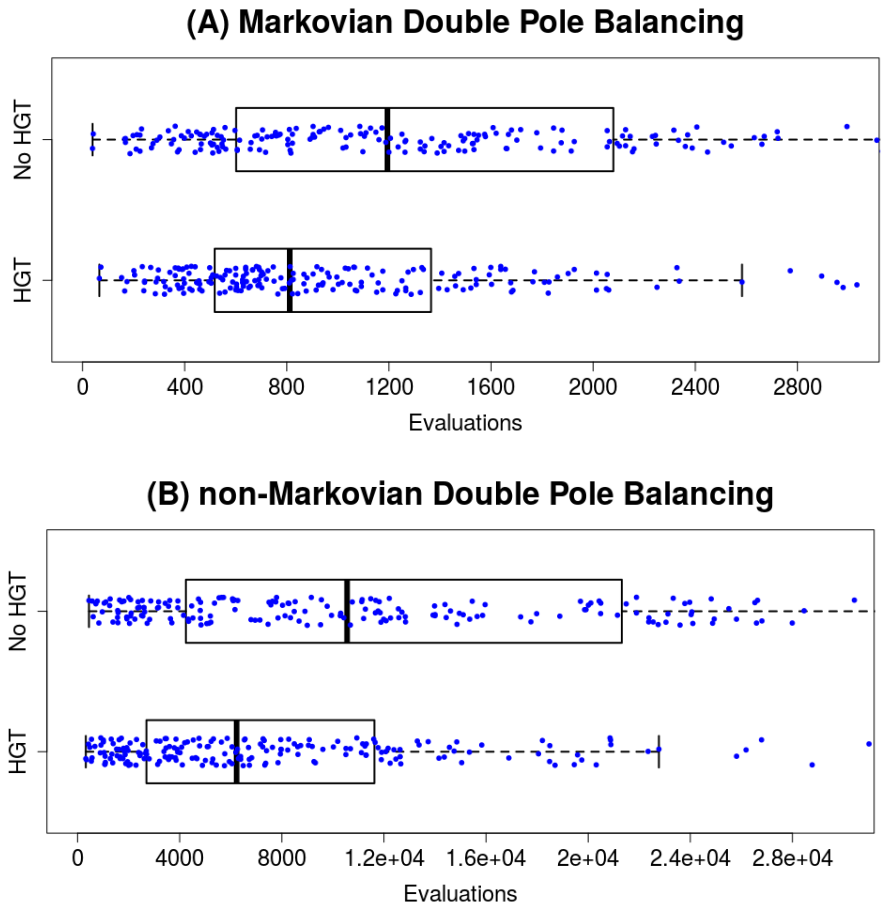


Figure 9.4: Box-plots with data overlayed for both neuroevolution problems. We give results for EGGP$_{HGT}$ (HGT) and EGGP$_{\mu\times\lambda}$ (No HGT); (A) Markovian, (B) non-Markovian. Overlayed data is jittered for visual clarity.

comparison to EGGP$_{\mu\times\lambda}$. Our Mann–Whitney U test reveals both results to be statistically significant ($p < \frac{0.05}{2}$), although without large effect. We give box-plots of the results of both problems in Figure 9.4, highlighting the degree to which HGT improves the efficiency of search. Taking into account the MEs and statistical significance, we can infer that HGT is indeed improving performance for these neuroevolution tasks. However, that we observe no large effect suggests that the change in MEs as a result of HGT is not large. This lack of large effect is in line with our statistical tests comparing EGGP$_{HGT}$ and EGGP$_{\mu\times\lambda}$ in Section 9.5.

Empirical comparison with other neuroevolution techniques on these problems is a difficult task. When these problems have been studied in the literature, they have not been standard-

	Mean Evaluations	
Technique	Markovian	Non-Markovian
CNE [80] [259]	22,100	76,906
SANE [80] [163]	12,600	262,700
RPG [80] [260]	4,981	5,649
ESP [80] [79]	3,800	7,374
NEAT [223]	3,600	20,918
NEVa [235]	2,177	-
EGGP$_{HGT}$	**1,175**	**8,891**
CGPANN [238]	1,111	-
CoSyNE [80]	954	1,249
CMA-ES [80] [107]	895	3,521

Table 9.6: MEs reported from various literature. Where a result is given, the publication it is taken from is referenced. A number of results are taken from comparative experiments in [80], in which case we also provide a reference for the approach after the reference to [80]. Results are ordered by MEs on Markovian pole balancing.

ised in many respects. For example, some implementations use Euler integration [125, 238], whereas others use Runge-Kutta integration [80, 223]. In some cases the longer pole starts at 1 degree from vertical [223, 238] and in others it starts at 4 degrees from vertical [80]. Some publications use 'bang-bang' force (where the network outputs ± 10 newtons) [125], whereas others have networks output continuous force [80, 238] as we have done. These distinctions, in combination with a general lack of publicly available implementations and that even standardising these conditions may unfairly bias against certain approaches chosen parameters, make a conventional statistical comparison difficult. For a more detailed discussion of problems drawing in drawing comparisons between methods on these tasks, see [236, Chapter 11].

However, the intention of our experiments is not to propose a state-of-the-art neuroevolution technique. Instead, we are investigating whether HGT works for graphs very different to those studied for symbolic regression. Nevertheless, we do list in Table 9.6 the MEs used by EGGP$_{HGT}$ in comparison to results reported in literature. While not a direct empirical comparison, this does give some notion of how the proposed algorithm compares. To this effect, results in [80] are helpful in that they have standardised comparisons over a number of approaches. In Table 9.6 we can see that EGGP$_{HGT}$ does quite well on Markovian pole

balancing, outperforming a number of techniques and performing similarly to CGPANN [238] which used much larger representation and had the longest pole starting at 1 degree from the vertical. However, the non-Markovian results are less impressive, with EGGP$_{HGT}$ being outperformed by all but 3 techniques in literature. We do take some reassurance from the fact that, on both problems, EGGP$_{HGT}$ outperforms the popular neuroevolution technique, 'Neuroevolution of Augmenting Topologies' (NEAT) [223].

The cause of the disparity between the two studied problems in comparison with the literature may be a result of our chosen parameters. We chose a recurrent edge rate of $p_{rec} = 0.1$, and it is generally believed that solutions to the non-Markovian problem are more dependent of memory than solutions to the Markovian problem. Therefore increasing p_{rec} and thereby increasing the amount of memory usage in the network may improve performance. Additionally, the non-Markovian problem is generally viewed as harder, and we may have hampered our search process by choosing such small, dense graphs. This may have reduced the evolvability of the system and the effect of this may be more prevalent on the harder problem, particularly if it has more local optima. Clearly, additional experiments with respect to parameterisation are required to establish the cause of this and improve EGGP$_{HGT}$'s performance on the non-Markovian task.

9.8 Conclusions and Future Work

In this work we have introduced a new and effective form of neutral HGT in the EGGP approach. Our approach utilises Active-Neutral transfer to copy the active components of one elite parent into the neutral material of another. Experimental results show that both HGT and the introduction of the $\mu \times \lambda$ EA lead to improvements in performance on benchmark symbolic regression problems. Comparing the final approach, EGGP$_{HGT}$, to TGP and CGP yields highly favourable results on a majority of problems.

We have also carried out neuroevolution experiments with HGT. Empirical comparisons on double pole balancing problems reveal that, for both Markovian and non-Markovian tasks, HGT improves the efficiency of search. This result is particularly interesting for two reasons. Firstly, we have evidence of positive effect of HGT for both symbolic regression and neuroevolution problems suggesting that this technique may function as a cross-domain recombination operator. Secondly, we deliberately chose to evolve very small, dense, graphs in our neuroevolution experiments to make the differences with our symbolic regression benchmarks more stark. That HGT remained beneficial reinforces the idea that it is useful.

These results have implications for broader research in EAs and GP. The reuse and recombination of genetic material is generally assumed to be a useful feature of an evolutionary system (e.g. TGP crossover [129]), but our Active-Neutral HGT events achieve reuse without altering the active components of individuals. Hence our approach contributes evidence to the notion that neutral drift aids evolutionary search [72]. Active-Neutral HGT events move beyond neutrality through mutation; we are effectively biasing the neutral components of individuals towards areas of the landscape we know to be 'good' with respect to the fitness function. While this is empirically beneficial here, it remains unknown whether this neutral biasing is helpful outside of the EGGP approach. Our favourable comparisons with TGP and CGP support this direction of thought; TGP offers recombination without neutral drift, whereas (vanilla) CGP offers neutral drift without recombination.

Our work here opens up a number of avenues for further research. It is desirable to investigate the influence of population parameters μ, λ and the HGT rate p_{HGT} on the performance of the described approach. Here, we have chosen small values of μ and λ and relatively high values of p_{HGT}; it is therefore interesting to consider whether larger values of μ and λ help or hinder the HGT process, and whether it is necessary to introduce multiple HGT events in a single generation when using larger populations. A possible way to investigate this could be through a graph equivalent of the Microbial Genetic Algorithm [92] as this could work as a minimal extension that supports the use of a larger population. Additionally, an investigation isolating depth control from HGT would help clarify whether HGT is more useful when individuals are smaller or larger.

There are two variants of HGT that should be investigated further. The first is a 'partial' HGT mechanism, where only a small subgraph of the active component of the donor is copied into the recipient. With such a mechanism, it would even be possible to take fragments of genetic material from several donors during a HGT event, thereby increasing the variance in the recipients received genetic material. However, empirical comparisons would certainly be necessary to clarify whether this is a preferable approach, and it is not yet clear how such a mechanism should be parameterised. Open questions are how should a subgraph be selected? how large should a subgraph be? how many subgraphs should be copied into the recipient, and how many donors should they come from?

Another interesting variant of HGT is 'headless chicken' HGT where the donor is substituted with a randomly generated individual. In this case, we would be replacing neutral material with randomly generated material. An empirical comparison between this variant and standard HGT could reveal any side effects caused by the HGT mechanism; if the head-

less chicken mechanism is effective, then we may have to reconsider our explanations for the effectiveness of HGT. However, we doubt that the headless chicken mechanism would compete with or outperform HGT, particularly in the symbolic regression problems, as we already have a large degree of neutral material in the genotype which undergoes neutral drift thereby achieving a similar randomising effect.

10 Conclusions and Future Work

This chapter is arranged as follows. In Section 10.1 we conclude the findings of this thesis and draw together conclusions from our different chapters. Finally, in Section 10.2 we set out areas for future work.

10.1 Overall Conclusions

In this thesis we have shown that rule-based graph programming can be used to design effective and novel Evolutionary Algorithms (EAs) over graphs. By designing genetic operators in this way, we have been able to create new systems which contribute new knowledge to the field of evolutionary computation with respect to representation of solutions, representation of genetic operators, new EAs, empirical studies and the effect of neutrality. In this section we review the findings of this thesis.

As we discuss the findings of this thesis, we relate them to the thesis aims set out in Section 1.2. These aims are:

1. To extend the graph programming language GP 2 to a probabilistic variant capable of expressing probabilistic transformations of graphs necessary to implement genetic operators for evolution.
2. To investigate whether and how these probabilistic graph programs can be used to design genetic operators for learning graphs.
3. To establish the benefits of using probabilistic graph programs as genetic operators, through empirical comparisons and theoretical discussion.
4. To investigate how probabilistic graph programs can be used to implement complex domain-specific rewrites in the context of evolution.
5. To empirically study the benefits and costs of using such rewrites throughout an evolutionary process.

6. To investigate how graphs can be recombined through probabilistic graph programs.
7. To empirically study the benefits and costs of using such recombinations throughout an evolutionary process.

Probabilistic graph programming

The first issue identified in this thesis was that the available rule-based graph programming language GP 2 did not have native support for the probabilistic constructs necessary to design stochastic genetic operators over graphs. To overcome this, we proposed P-GP 2 in Chapter 3 which extends GP 2's syntax, allowing a programmer to specify probability distributions over outcome graphs. Through numerous examples of randomised graph algorithms and P-GP 2 based genetic operators, we have seen that P-GP 2 is highly practical with specific application in evolutionary computation, and broader application beyond the scope of this thesis. P-GP 2 has been used in 5 of the technical chapters of this thesis at both descriptive and implementation levels, and is therefore a fundamental contribution that enables the other findings of the work presented.

By extending GP 2 to P-GP 2, which is capable of expressing probabilistic transformations of graphs, we have directly addressed aim 1. However, while GP 2 is computationally complete [184] it is not clear whether P-GP 2 is complete in the sense of describing any probability distribution over graphs. Given the relatively simple syntax available in P-GP 2, it appears likely that this is not the case. There may be domains where we desire genetic operators over graphs whose probability distributions we cannot express in P-GP 2's current form. So while aim 1 has been met in the context of this thesis, there is clearly more work to be done in this endeavour. An additional consideration of the contribution of P-GP 2 is that we have used the implementation of P-GP 2 to perform empirical experiments addressing aims 3, 5 and 7.

Representation of solutions

The issue of representation of solution was resolved through the introduction of Function Graphs (FGs) in Chapter 4. By describing a class of graphs capable of expressing both stateful and stateless programs in a variety of domains, we are able to design genetic operators over those graphs that are sufficiently generic and cross-domain. This contribution, taken alongside the generic representations of Cartesian Genetic Programming (CGP) [157], Parallel Distributed Genetic Programming (PDGP) [188] and Neuroevolution of Augmenting Topologies (NEAT) [222], demonstrates the importance of the choice of representation

in evolutionary computation and the benefits of using the generic representation of graphs. Further, it can be argued that it is the representation itself that enables our genetic operators, making the case that representation of solutions must be carefully designed as it may have significant consequences for the overall EA.

While these findings do not explicitly address any of the thesis aims we have set out, they are fundamental to many other findings of this thesis. Our choice of representation was a central consideration in the design of our genetic operators and, in particular, enabled our extensions to Semantic Neutral Drift (SND) and Horizontal Gene Transfer (HGT). As a result, these findings have contributed to addressing aims 2, 4 and 6.

Representation of genetic operators

Representation of genetic operators is an aspect of evolutionary computation that has seen little attention. Often when a genetic operator is proposed it is described in text or through diagrams e.g. [129,157,188]. In this thesis we have consistently described our genetic operators as P-GP 2 programs, which has a number of interesting consequences. Firstly, by choosing P-GP 2 as a representation, we are able to straightforwardly reason about our genetic operators, as seen in our argument of the correctness of edge mutation in Section 5.3.1 or in our argument of Evolving Graphs by Graph Programming (EGGP), generalising the landscape of CGP in Section 5.6.1. Secondly, the choice of representation of genetic operators may significantly aid the practitioner in the proposal of new ideas, as demonstrated in Chapter 8 where we are able to concisely describe complex graph rewrites to yield more efficient evolutionary search. That is not to say that it would be impossible to implement these rewrites without P-GP 2 - ultimately all P-GP 2 code used compiles to C code. However, in this thesis it is the chosen representation of the operators that has facilitated design, prototyping and implementation of our proposed ideas. We therefore argue that appropriate choice of representation of genetic operators can significantly aid both the evolutionary computation practitioner's ambitions and their audience's understanding. In particular we have seen that P-GP 2 is a particularly practical and effective paradigm for the representation of genetic operators over graphs.

Our findings on the representation of genetic operators directly addresses aim 2: we have proposed many probabilistic graph programs that can be used as genetic operators for learning graphs. Through our use of P-GP 2 we have seen that probabilistic programs can be used to design a variety of genetic operators which can be used to efficiently evolve graphs. Additionally, it is through our choice of representation of genetic operators that we gain access to

the complex SND rewrites, addressing aim 4, and the HGT mechanism, addressing aim 6.

New Evolutionary Algorithms

In Chapter 5 we combined FGs with P-GP 2 programs to propose the first EA with genetic operators described through rule-based graph programming. Our approach, EGGP, comes with initialisation, atomic edge mutation and atomic node mutation operators, all of which are described and implemented as P-GP 2 programs. We have seen arguments for the correctness of our mutation operators, and made the case that EGGP's landscape strictly generalises that of CGP in Section 5.6.1. EGGP has demonstrated an impressive ability to work across domains; through this thesis we have seen EGGP and its variants used to evolve digital circuits, symbolic expressions, digital counters, mathematical sequences, generalising digital parity checks and Artificial Neural Networks (ANNs).

A remarkable aspect of EGGP is how readily it can be extended. In Chapter 7 we gave minor modifications of the initialisation and mutation operators that allowed us to learn stateful programs. This variant is named Evolving Recurrent Graphs by Graph Programming (R-EGGP). In Chapter 8 we implemented SND through new genetic operators, based on known logical equivalence laws, to equip EGGP with additional pathways for neutral drift to occur. The implementation of this was achieved simply in the creation of new P-GP 2 programs which were applied to the surviving members of the population in each generation. In Chapter 9 we implemented HGT as a mechanism for passive sharing of genetic information. This was implemented through some simple P-GP 2 programs in combination with a disjoint union operator, which together were applied once per generation with a given probability. In Chapter 9 we also saw that our operators from Chapter 7 in combination with HGT and new weight mutation operators make it possible to evolve ANNs for control tasks.

The extendable nature of EGGP has allowed us to investigate many new ideas as we have just described. In the design and evaluation of these extensions, we have achieved a number of favourable comparisons with approaches from the literature. We therefore conclude that EGGP and its variants provide useful tools for the evolution of graphs with applications in many fields.

Our new evolutionary techniques address a number of our thesis aims. By proposing EGGP and R-EGGP we have directly addressed aim 2, demonstrating that probabilistic graph programs can be used to design generic operators for learning graphs. Our discussion of the landscapes induced by EGGP's genetic operators, both with respect to correctness and

generality, addresses aim 3 by highlighting theoretical benefits of using probabilistic graph programming to describe genetic operators. By extending EGGP through the incorporation of SND, we have investigated how probabilistic graph programs can be used to implement domain-specific rewrites, directly addressing aim 4. Similarly, by extending EGGP to allow for HGT, we have investigated how graphs can be recombined through probabilistic graph programs, directly addressing aim 6.

Empirical results

The conclusions of this thesis are founded on rigorous empirical study. To demonstrate the effectiveness of the proposed approaches, we have performed extensive comparisons with approaches from the literature. In Chapter 6 we found that EGGP significantly outperforms CGP on many digital circuit synthesis tasks with respect to the evaluations required. Additional experiments with Ordered EGGP (O-EGGP) suggest that the improvement in performance is a result of the generalised landscape of EGGP. However we also found fewer statistical differences between EGGP, CGP and Genetic Programming (GP) on symbolic regression tasks. This disparity was in some sense overcome in Chapter 9 where we found that EGGP along with HGT significantly outperforms EGGP, CGP and GP on many of these same problems with respect to the quality of solutions found. We also established, in Chapter 7, that R-EGGP significantly outperforms Recurrent CGP (RCGP) on many digital counter synthesis and n-bit parity check synthesis tasks, although we observed fewer differences on mathematical sequence synthesis tasks. An interesting intersection between our digital circuit comparisons and our digital counter comparisons is that the difference in performance between the EGGP variant and the CGP variant increases with problem difficulty. Overall, on a majority of problems that we have studied, we have that EGGP or some variant is the best performing algorithm in comparison to the alternatives we have taken from the literature.

We have also performed extensive comparisons between our EGGP variants to verify whether our individual proposals are responsible for yielding meaningful improvements in performance. In Chapter 8 we performed experiments comparing various semantics-preserving rule-sets allowing us to identify the best performing rule-sets for further study. By performing comparative experiments between the EGGP system equipped with SND and the standard EGGP system with an 'easier' function set, we were able to find cases where choosing otherwise detrimental parameters may yield statistically significant improvements in performance if that choice then facilitates neutral drift. In Chapter 9 we performed experiments comparing the various components of the EGGP extension that supported HGT. Through studying the

results of this decomposition we concluded that the observed improvements in performance were due to a mixture of our new $\mu \times \lambda$ EA and our HGT mechanism. We also performed experiments comparing R-EGGP with and without HGT for the evolution of ANNs. We intentionally chose a different solution space of particularly small and dense graphs and again found that the inclusion of HGT aided the evolutionary search. These results together strengthen the case that HGT is a general mechanism for recombination of FGs.

Through empirical evaluation of EGGP and R-EGGP, we have directly addressed aim 3. Through our comparisons of several different neutral rule-sets and our analysis of SND, we have empirically studied the benefits of using complex domain-specific rewrites implemented as probabilistic graph programs, addressing aim 5. Similarly, through our comparisons of EGGP with HGT, we have empirically studied the benefits of using recombinations implemented as probabilistic graph programs.

Effect of neutrality

An emergent theme in this thesis has been in the study of the effect of neutral drift on the evolution of graphs. In Chapter 8 we found that by introducing known logical equivalence laws to the evolutionary system we could improve the efficiency of search in many cases. We have described a technique whereby domain knowledge is used to build *additional* neutral drift directly into the landscape, which can significantly improve performance. In Chapter 9 we exploited the neutral components of the individual to achieve a form of genetic recombination that does not affect the fitness of either parents, but allows meaningful code reuse to occur as a byproduct of the HGT mechanism and our existing atomic mutations. In this case, we have found a way to bias the neutral components of the individual towards genetic material that we believe to be effective according to the fitness function. We are in effect *biasing* neutral drift towards 'good' parts of the landscape.

We have therefore presented 2 distinct instances where adding new mechanisms for neutral drift and increasing the occurrence of neutral drift may significantly improve the efficiency of search over graphs. These findings build upon those of various works which assert the benefits of neutral drift [103, 156, 266] and also go further; we are presenting evidence that the specific design of the mechanisms of neutral drift may influence and improve upon the performance of the system. These findings have an interesting interaction with our earlier discussion of representation, both with respect to solutions and genetic operators, as the techniques that provide these new insights into neutral drift are based on exploitation of

the solution representation and depend on complex rewrites which can readily be described, implemented and reasoned about through P-GP 2.

Our findings on the effect of neutrality do not directly address any of our thesis aims. However, the fact that we have established these conclusions through our use of genetic operators designed in P-GP 2 using complex domain-specific rewrites and graph recombinations does add further evidence for the benefits of each of these in relation to thesis aims 3, 5 and 7, respectively.

In summary

The findings of this thesis are summarised as follows:

1. Probabilistic graph programming is a practical paradigm for designing genetic operators over graphs and has further applications beyond evolutionary computation.
2. FGs are a suitably generic class of graphs for the evolution of solutions to various problems of interest.
3. EGGP is an intuitive and extendable EA that uses probabilistic graph programming to describe genetic operators.
4. EGGP is effective in comparison to approaches from the literature, particularly on digital circuit synthesis tasks.
5. R-EGGP is an extension to EGGP that facilitates the evolution of stateful programs.
6. R-EGGP is effective in comparison to approaches from the literature, particularly on digital counter synthesis and n-bit parity check tasks.
7. EGGP can be extended to incorporate SND. In particular, we have shown that logical equivalence laws implemented as probabilistic graph programs can be applied throughout the evolutionary process to build neutral pathways into the evolutionary landscape.
8. SND may improve the efficiency of search. In particular, we have found that there are circumstances where it is preferable to choose parameters which would otherwise be detrimental to performance if that facilitates the implementation of SND.
9. EGGP can be extended to incorporate HGT. HGT allows for genetic recombination of FGs without modifying the active components of any individuals.
10. HGT may improve the efficiency of search. In particular, we have found that EGGP with HGT often outperforms other approaches from the literature on symbolic regres-

sion problems.

11. By taking R-EGGP in combination with HGT and weight mutations, it is possible to efficiently solve neuroevolution tasks.

10.2 Future Work

Our work on EGGP opens up a number of directions for future work. While we have already set out areas for future work in each chapter that are specific to the findings of that chapter, here we describe some particularly promising areas of future work. In Section 10.2.1 we describe a number of potential application areas, and discuss the necessary extensions to support some of these areas. In Section 10.2.2 we describe the extension of EGGP to hierarchical graphs, and the potential consequences for learning programs with meaningful abstraction of data types. Finally, in Section 10.2.3 we present a possible mechanism for meta-learning of landscapes based on higher-order graph transformation.

10.2.1 New Domains

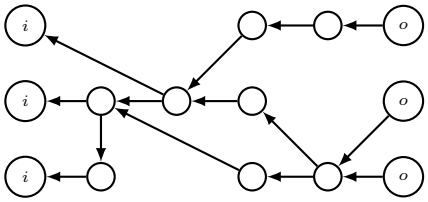
As we have described in Chapter 6 there are a number of application areas where our existing EGGP system may be directly applied:

1. Approximate circuits, as in [165, 249, 250], by introducing a multi-objective EA such as SPEA2 [268] as a replacement of the $1 + \lambda$ algorithm.
2. Cryptographic circuits, as in [179, 180].
3. Image processing, as in [88, 89, 201].
4. Multi-step forecasting, as in [58].
5. A number of other plausible problems are discussed in [253], including the lawnmower problem and the hierarchical if-and-only-if problem.

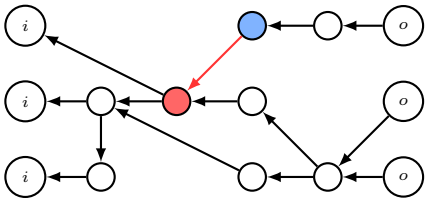
Additionally, as we described in Chapter 7, by setting $p_{rec} = 1$ in R-EGGP, it may be possible to learn topologies in many interesting domains, for example, in the search for a topology of an echo state network [108] or a random Boolean network [211].

There are, however, a number of domains where our systems cannot be applied in their current forms. In this section we describe 3 particularly appealing problem domains and discuss how EGGP may be modified to support them.

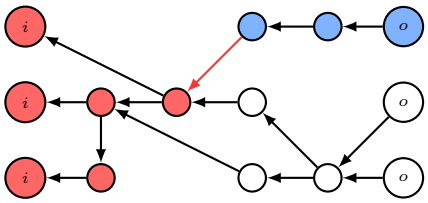
This individual is to undergo an edge mutation preserving acyclicity and balance of nodes' numbers of inputs and outputs.



(1): An edge to mutate is chosen at random and marked (red) alongside its source node s (blue) and target node t (red).



(2): Invalid candidate nodes for redirection are identified. If a node v has a directed path to s it is marked blue, or a directed path from t it is marked red. Swapping edges in either case may introduce a cycle.



(3) The edge e is swaps targets with another edge e' who's source has not been marked. This transformation cannot introduce a cycle and maintains the number of inputs and outputs of each gate.

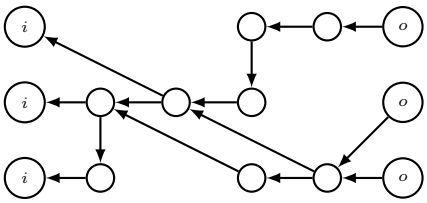


Figure 10.1: A suggestion for how edges may be swapped in a quantum circuit while preserving acyclicity and the constraint that the number of inputs to a quantum gate must equal the number of outputs.

Quantum circuits

Many attempts have been made to design evolutionary systems capable of automatically synthesising quantum circuits; see [225] for a detailed review. Many of these systems also propose a domain specific language which can then be decoded into a quantum circuit [149, 214,217]. The main reason for this is that quantum circuits are significantly more constrained than their classical counterparts. A quantum circuit models the time-evolution of a fixed dimension quantum state. For this reason a quantum gate is interpreted as a square matrix and must have the same number of inputs and outputs. Further, both inputs and outputs must be explicitly ordered to prevent any ambiguity. On top of these constraints, quantum circuits are in general acyclic. It would be possible to use EGGP to evolve solutions in these domain specific languages. Perhaps more interesting is to attempt to evolve quantum circuits directly but doing this demands great care in the design of genetic operators as to respect these constraints.

There are many open issues in the extension of EGGP to the evolution of direct representations of quantum circuits. While we do not resolve all of them here, we do make a suggestion with respect to atomic edge mutation. EGGP's atomic edge mutation cannot be directly applied to quantum circuits because it makes no guarantee all quantum gates will have equal numbers of inputs and outputs. In fact it appears that, under the assumption that the new target of the mutating edge is distinct from the previous target, this mutation operator is guaranteed to break this constraint when applied to any valid quantum circuit. We suggest that a more appropriate approach is to 'swap' edges while preserving acyclicity. In Figure 10.1, we give a suggestion of how such a mutation could work.

If an extension to EGGP that supported quantum circuits was proposed, it would need evaluating. The approach to benchmarking in [6] set out a possible way this could be done. Additional experiments clarifying the difference between evolution of a direct representation and evolution in a domain specific language would provide further insight into the general field of quantum circuit synthesis.

Bayesian networks

As we discussed in Section 2.4 there are many approaches in the literature which attempt to optimise Bayesian network structure via a graph-based EA (see [133]). It is natural therefore for us to consider the same task through a variant of EGGP. Interestingly, the atomic edge mutation of EGGP is directly applicable to Bayesian networks as a central constraint of

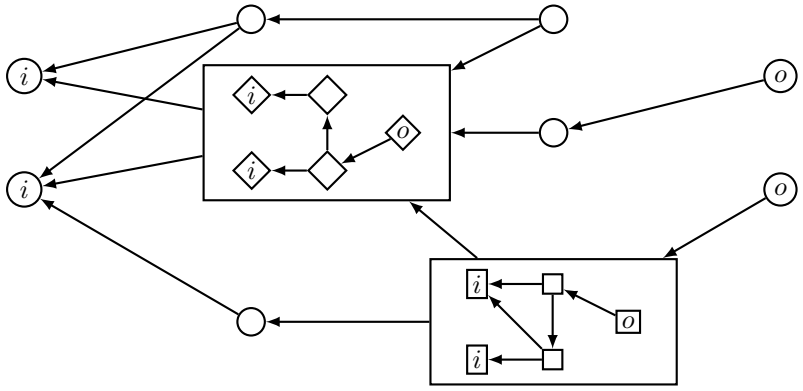


Figure 10.2: A plausible model of a HFG. Modules, contained within nodes, could then be modified, copied and deleted with hierarchical graph transformations [56].

Bayesian networks is acyclicity.

A particular intersection between the work we have covered and the evolution of Bayesian networks is that Bayesian networks are already equipped with equivalence classes [39]. This additional intersection, alongside our results demonstrating the effectiveness of SND in improving the efficiency of evolutionary search, suggest that a graph programming-based approach to Bayesian network evolution may be particularly effective.

Deep neural network architectures

Evolution of Deep Neural Network (DNN) architectures has seen a surge of interest in recent years [136,153,226]. Evolution of these architectures offers a potential avenue to higher quality deep learning implementations and novel structures which provide new insights into the design of deep learning architectures. Often the problem is framed as one of learning an architecture for a particular task, in particular in image recognition tasks [153,226]. Architectures for these tasks are in general acyclic, and the EGGP framework is highly applicable to these problems with respect to the evolution of structure. However a significant roadblock to this is in our representation of labels. Often it is desirable to treat an architecture layer as a node and have multiple parameters associated with that layer, such as the number of filters, the dropout rate, the kernel size and the use of pooling [153]. It is more difficult to express genetic operators over these features if they are represented as a P-GP 2 list. Clearly more thought must be given to the representation of layer parameters for such a system to be successful.

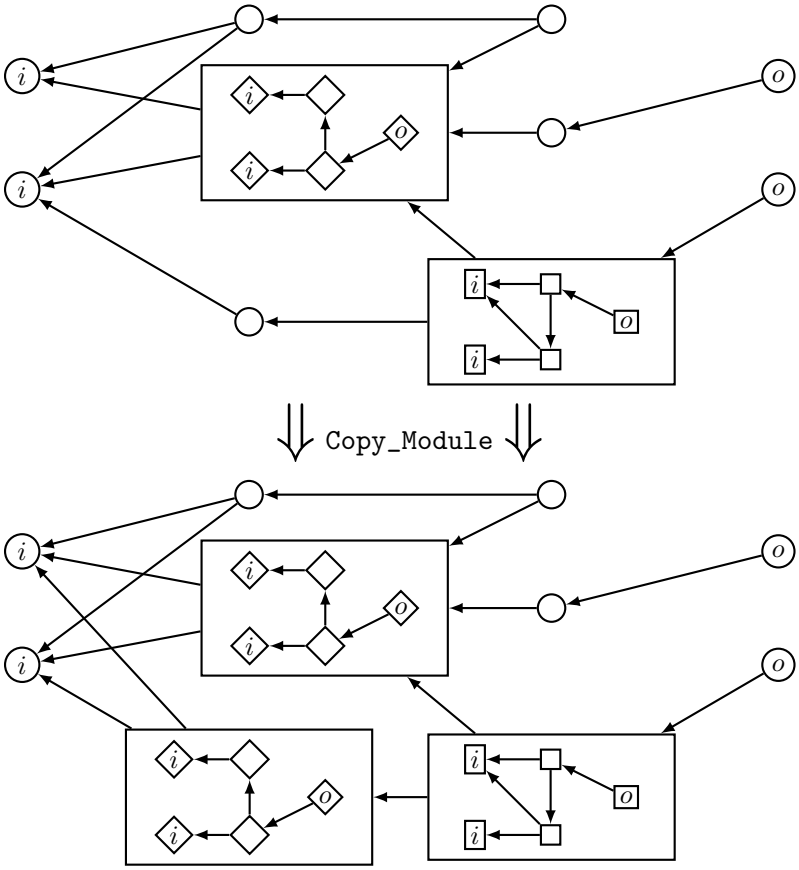


Figure 10.3: A notion of how subgraph copying (see [56]) might be utilised in a modular extension of EGGP to copy entire modules between function nodes.

10.2.2 Evolving Hierarchical Graphs

In Chapters 4 and 5 we presented ideas as to how EGGP could be extended to hierarchical graphs and therefore gain access to a model of modularity. A particularly appealing model of hierarchy is shown in Figure 10.2, where function nodes may then contain other EGGP graphs. This model of hierarchy in principle allows for arbitrary levels of nesting and does not require parameters specifying the number of modules. Components of the Hierarchical Function Graphs (HFGs) may be modified through existing techniques of hierarchical graph transformation [31,56,176], in the same way that we have used graph programming to modify 'flat' FGs. For example, it would be possible to use the formalism of [56] whereby a graph contained within a node may be copied as a way of duplicating a module within a HFG. We give a visual example of how this may work in Figure 10.3.

An intriguing direction of thought is to use these structures to describe the decomposition of data types. For example, we might wish to learn a function over floats. A 32-bit float can be decomposed into 4 bytes, each of which can be decomposed into 8 bits. So it would in principle to be possible to learn a hierarchical structure with 3 layers of hierarchy. At the highest level, a function is expressed over floats. Inside each function node at the highest level there is a graph representing a function over bytes. Inside each function node in the second level, there is a graph representing a function over bits consisting of logic gates. This offers a possible avenue for an EGGP-based system to automatically learn functions over a higher-order data type as a composition of logic gates. As EGGP has been found to be particularly effective at learning digital circuits, it may then be possible to 'lift' this efficient form of search to more complex data. Whether or not this is more effective than simply learning functions over a higher-order function set, as we did with our symbolic regression experiments, is a matter for empirical study.

10.2.3 Meta-Learning of Landscapes

In this work we have proposed various genetic operators over graphs. More broadly, there are works which attempt to induce genetic operators entirely, rather designing them by hand. For example, [102] use a GP algorithm to evolve probability distributions for use as mutation operators in an EA. GP has been applied to register machines [263] to learn programmatic mutation operators for Genetic Algorithms (GAs) applied to a range of test problems. A core motivation for such works is that by automatically synthesising genetic operators we may discover landscapes which significantly aid the evolutionary process. Interestingly, there is a natural intersection with this line of work and existing theory on higher-order graph transformation. Higher-order Double-Pushout (DPO) rewriting [145] allows the manipulation of DPO rules in a manner analogous to DPO rules rewriting graphs. Figure 10.4 shows the commutative diagram for this concept; a pattern is matched in a rule, and that pattern is replaced with a new pattern through deletion and then addition of new elements.

An interesting line of work would be the development and implementation of meta-learning of mutation operators as P-GP 2 Programs. This could be achieved by applying higher-order rules to graph transformation rules in the same manner that we have applied graph transformation rules to graphs. It may then be possible to design meta-EAs that learn effective mutations for sets of problems, rather than optimising solutions for individual problems. The precise form that this should take remains unclear with numerous questions to be answered:

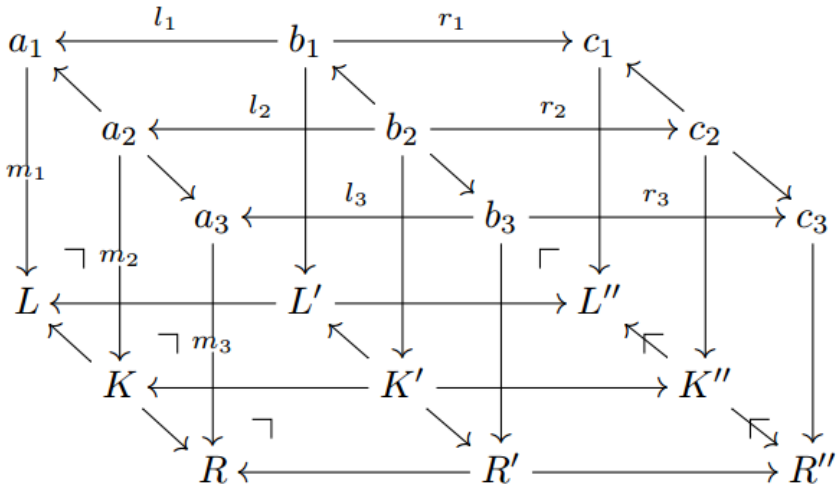


Figure 10.4: A higher-order DPO diagram, figure taken from [145]. A higher-order rule matches a pattern within a first order rule and updates that pattern, effectively allowing transformation of rules themselves.

- Should meta-learning be of only rules, or of both rules and P-GP 2 programs?
- How should these 2nd order transformations of rules be implemented? Is it possible to encode P-GP 2 rules as graphs which can then be manipulated with other P-GP 2 programs?
- How should this meta-evolution be structured? Is it more beneficial to evolve mutation operators alongside solutions (as a self-configuring EA), or as a parameter for EAs (as a generative hyper-heuristic, see [30])?
- How should an individual mutation operator be evaluated during the evolutionary process?

The potential result of this is a 2nd order EA which can be pointed towards different, difficult, domains and effectively induce landscapes which benefit search while respecting the constraints of a given problem. As we are discussing higher-order evolution of graph transformations and we know graphs to be a ubiquitous data structure, such an algorithm would in principle be applicable to a great many domains.

References

[1] A. Agapitos, M. O'Neill, A. Kattan, and S. M. Lucas, "Recursion in tree-based Genetic Programming," Genetic Programming and Evolvable Machines, vol. 18, no. 2, pp. 149–183, 2017, doi: 10.1007/s10710-016-9277-5.

[2] L. Altenberg, "Emergent phenomena in Genetic Programming," in Proc. 3rd International Conference on Evolutionary Programming. World Scientific, 1994, pp. 233–241.

[3] P. J. Angeline, "Subtree crossover: Building block engine or macromutation?" in Proc. Second Annual Conference on Genetic Programming. Morgan Kaufmann, 1997, pp. 9–17.

[4] P. J. Angeline, G. M. Saunders, and J. B. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," IEEE Trans. Neural Networks, vol. 5, no. 1, pp. 54–65, 1994, doi: 10.1109/72.265960.

[5] F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro, "Denser: deep evolutionary network structured representation," Genetic Programming and Evolvable Machines, vol. 20, no. 1, pp. 5–35, Mar 2019, doi: 10.1007/s10710-018-9339-y.

[6] T. Atkinson, A. Karsa, J. Drake, and J. Swan, "Quantum program synthesis: Swarm algorithms and benchmarks," in Proc. European Conference on Genetic Programming, EuroGP 2019, ser. LNCS, vol. 11451. Springer, 2019, pp. 19–34, doi: 10.1007/978-3-030-16670-0˙2.

[7] T. Atkinson, D. Plump, and S. Stepney, "Probabilistic graph programming," in Pre-Proc. Graph Computation Models, GCM 2017, 2017. [Online]. Available: http://www.cs.york.ac.uk/plasma/publications/pdf/AtkinsonPlumpStepney.GCM17.pdf

[8] ——, "Evolving graphs by graph programming," in Proc. European Conference on Genetic Programming, EuroGP 2018, ser. LNCS, vol. 10781. Springer, 2018, pp. 35–51, doi: 10.1007/978-3-319-77553-1˙3.

[9] ——, "Probabilistic graph programs for randomised and evolutionary algorithms," in Proc. International Conference on Graph Transformation, ICGT 2018, ser. LNCS, vol. 10887. Springer, 2018, pp. 63–78, doi: 10.1007/978-3-319-92991-0˙5.

[10] ——, "Evolving graphs with horizontal gene transfer," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2019. ACM, 2019, pp. 968–976, doi: 10.1145/3321707.3321788.

[11] ——, "Evolving graphs with semantic neutral drift," Natural Computing, 2019, doi: 10.1007/s11047-019-09772-4.

[12] ——, "Horizontal gene transfer for recombining graphs," Genetic Programming and Evolvable Machines, 2020, doi: 10.1007/s10710-020-09378-1.

[13] T. Bäck, F. Hoffmeister, and H. Schwefel, "A survey of evolution strategies," in Proc. International Conference on Genetic Algorithms, IGCA 1991, R. K. Belew and L. B. Booker, Eds. Morgan Kaufmann, 1991, pp. 2–9.

[14] P. Bahr, "Convergence in infinitary term graph rewriting systems is simple," Mathematical Structures in Computer Science, vol. 28, no. 8, pp. 1363–1414, 2018, doi: 10.1017/S0960129518000166.

[15] C. Bak, "GP 2: Efficient implementation of a graph programming language," Ph.D. dissertation, Department of Computer Science, University of York, 2015. [Online]. Available: http://etheses.whiterose.ac.uk/12586/

[16] C. Bak, G. Faulkner, D. Plump, and C. Runciman, "A reference interpreter for the graph programming language GP 2," in Proc. Graphs as Models (GaM 2015), ser. EPTCS, vol. 181, 2015, pp. 48–64.

[17] C. Bak and D. Plump, "Compiling graph programs to C," in Proc. International Conference on Graph Transformation, ICGT 2016, ser. LNCS, vol. 9761. Springer, 2016, pp. 102–117, doi: 10.1007/978-3-319-40530-8˙7.

[18] W. Banzhaf, "Genetic programming for pedestrians," Mitsubishi Electric Research Labs, MERL Technical Report 93-03, 1993.

[19] ——, "Genotype-phenotype-mapping and neutral variation — a case study in Genetic Programming," in Proc. 3rd International Conference on Parallel Problem Solving from Nature, PPSN III, ser. LNCS, vol. 866. Springer, 1994, pp. 322–332, doi: 10.1007/3-540-58484-6˙276.

[20] L. Barnett, "Ruggedness and neutrality; the NKp family of fitness landscapes," in Proc. 6th International Conference on Artificial Life. MIT Press, 1998, pp. 18–27.

[21] N. Behr, V. Danos, and I. Garnier, "Stochastic mechanics of graph rewriting," in Proc. 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016. ACM, 2016, pp. 46–55, doi: 10.1145/2933575.2934537.

[22] R. K. Belew, J. McInerney, and N. N. Schraudolph, "Evolving networks: using the genetic algorithm with connectionist learning," in Artificial Life II, ser. SFI Studies in the Sciences of Complexity: Proceedings. Addison-Wesley, 1992, vol. 10, pp. 511–547.

[23] G. Bergmann, Á. Horváth, I. Ráth, and D. Varró, "A benchmark evaluation of incremental pattern matching in graph transformation," in Proc. International Conference on Graph Transformation, ICGT 2008, ser. LNCS, vol. 5214. Springer, 2008, pp. 396–410, doi: 10.1007/978-3-540-87405-8˙27.

[24] C. M. Bishop, Pattern recognition and machine learning. Springer, 2006.

[25] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," ACM Comput. Surv., vol. 35, no. 3, pp. 268–308, 2003. [Online]. Available: http://doi.acm.org/10.1145/937503.937505

[26] M. Brameier, "On linear genetic programming," Ph.D. dissertation, Technical University of Dortmund, Germany, 2004. [Online]. Available: http://hdl.handle.net/2003/20098

[27] M. F. Brameier and W. Banzhaf, Linear Genetic Programming. Springer, 2010, doi: 10.1007/978-0-387-31030-5.

[28] S. Brave, "Evolving recursive programs for tree search," in Advances in Genetic Programming. MIT Press, 1996, pp. 203–219.

[29] Z. Buk, J. Koutnik, and M. Snorek, "NEAT in HyperNEAT substituted with Genetic Programming," in Proc. International Conference on Adaptive and Natural Computing Algorithms, ICANNGA 2009, ser. LNCS, vol. 5495. Springer, 2009, pp. 243–252, doi: 10.1007/978-3-642-04921-7˙25.

[30] E. K. Burke, M. R. Hyde, G. Kendall, and J. Woodward, "Automatic heuristic generation with genetic programming: Evolving a jack-of-all-trades or a master of one," in Proceedings Genetic and Evolutionary Computation Conference, GECCO 2007. ACM, 2007, pp. 1559–1565, doi: 10.1145/1276958.1277273.

[31] G. Busatto, H.-J. Kreowski, and S. Kuske, "Abstract hierarchical graph transformation," Mathematical Structures in Computer Science, vol. 15, no. 4, pp. 773–819, 2005, doi: 10.1017/S0960129505004846.

[32] E. Cantú-Paz, "A survey of parallel genetic algorithms," Calculateurs Paraleles, vol. 10, 1998.

[33] ——, "Master-slave parallel genetic algorithms," in Efficient and Accurate Parallel Genetic Algorithms. Springer, 2001, pp. 33–48, doi: 10.1007/978-1-4615-4369-5_3.

[34] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Evolving competitive car controllers for racing games with neuroevolution," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2009. ACM, 2009, pp. 1179–1186, doi: 10.1145/1569901.1570060.

[35] ——, "On-line neuroevolution applied to the open racing car simulator," in Proc. IEEE Congress on Evolutionary Computation, CEC 2009. IEEE, 2009, pp. 2622–2629, doi: 10.1109/CEC.2009.4983271.

[36] D. Carlton and M. Zhang, "Parallel linear genetic programming," in Proc. European Conference on Genetic Programming, EuroGP 2011, vol. 6621. Springer, 2011, pp. 178–189, doi: 10.1007/978-3-642-20407-4_16.

[37] Y. Chen, S. Mabu, K. Hirasawa, and J. Hu, "Trading rules on stock markets using genetic network programming with sarsa learning," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2007. ACM, 2007, pp. 1503–1503, doi: 10.1145/1276958.1277232.

[38] D. M. Chickering, "Learning Bayesian networks is NP-complete," in Learning from data, ser. LNS. Springer, 1996, vol. 112, pp. 121–130, doi: 10.1007/978-1-4612-2404-4˙12.

[39] ——, "Learning equivalence classes of Bayesian-network structures," J. machine learning research, vol. 2, pp. 445–498, 2002, doi: 10.1162/153244302760200696.

[40] C. Clack and T. Yu, "Performance enhanced Genetic Programming," in Proc. 6th International Conference on Evolutionary Programming, ser. LNCS, vol. 1213. Springer, 1997, pp. 85–100, doi: 10.1007/BFb0014803.

[41] J. Clegg, J. A. Walker, and J. F. Miller, "A new crossover technique for Cartesian Genetic Programming," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2007. ACM, 2007, pp. 1580–1587, doi: 10.1145/1276958.1277276.

[42] B. Coecke and R. Duncan, "Interacting quantum observables: categorical algebra and diagrammatics," New J. Physics, vol. 13, no. 4, p. 043016, 2011, doi: 10.1088/1367-2630/13/4/043016.

[43] M. Collins, "Finding needles in haystacks is harder with neutrality," Genetic Programming and Evolvable Machines, vol. 7, no. 2, pp. 131–144, 2006, doi: 10.1007/s10710-006-9001-y.

[44] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. MIT Press, 2009.

[45] A. Corradini, T. Heindel, F. Hermann, and B. König, "Sesqui-pushout rewriting," in Proc. International Conference on Graph Transformation, ICGT 2006, ser. LNCS, vol. 4178. Springer, 2006, pp. 30–45, doi: 10.1007/11841883˙4.

[46] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe, "Algebraic approaches to graph transformation — Part I: Basic concepts and double pushout approach," in Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific, 1997, vol. 1, ch. 3, pp. 163–245.

[47] C. Cotta and J. Muruzábal, "On the learning of Bayesian network graph structures via evolutionary programming," in Proc. 2nd Workshop on Probabilistic Graphical Models, 2004, pp. 65–72.

[48] N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in Proc. International Conference on Genetic Algorithms, ICGA 1985. L. Erlbaum Associates Inc., 1985, pp. 183–187.

[49] V. Danos and C. Laneve, "Formal molecular biology," Theoretical Computer Science, vol. 325, no. 1, pp. 69–110, 2004, doi: 10.1016/j.tcs.2004.03.065.

[50] T. E. Davis and J. C. Príncipe, "A markov chain framework for the simple genetic algorithm," Evolutionary Computation, vol. 1, no. 3, pp. 269–288, 1993, doi: 10.1162/evco.1993.1.3.269.

[51] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," IEEE Trans. Evolutionary Computation, vol. 6, no. 2, pp. 182–197, 2002, doi: 10.1109/4235.996017.

[52] S. Doncieux, N. Bredeche, J.-B. Mouret, and A. E. G. Eiben, "Evolutionary robotics: what, why, and where to," Frontiers in Robotics and AI, vol. 2, p. 4, 2015, doi:

10.3389/frobt.2015.00004.

[53] S. Doncieux, J.-B. Mouret, N. Bredeche, and V. Padois, "Evolutionary robotics: Exploring new horizons," in New horizons in evolutionary robotics, ser. SCI. Springer, 2011, vol. 341, pp. 3–25, doi: 10.1007/978-3-642-18272-3˙1.

[54] R. M. Downing, "Evolving binary decision diagrams using implicit neutrality," in Proc. IEEE Congress on Evolutionary Computation, CEC 2005. IEEE, 2005, pp. 2107–2113, doi: 10.1109/CEC.2005.1554955.

[55] ——, "Neutrality and gradualism: encouraging exploration and exploitation simultaneously with binary decision diagrams," in Proc. IEEE Congress on Evolutionary Computation, CEC 2006. IEEE, 2006, pp. 615–622, doi: 10.1109/CEC.2006.1688367.

[56] F. Drewes, B. Hoffmann, and D. Plump, "Hierarchical graph transformation," J. Computer and System Sciences, vol. 64, no. 2, pp. 249–283, 2002, doi: 10.1006/jcss.2001.1790.

[57] M. Duarte, V. Costa, J. Gomes, T. Rodrigues, F. Silva, S. M. Oliveira, and A. L. Christensen, "Evolution of collective behaviors for a real swarm of aquatic surface robots," PloS one, vol. 11, no. 3, p. e0151834, 2016, doi: 10.1371/journal.pone.0151834.

[58] I. Dzalbs and T. Kalganova, "Multi-step ahead forecasting using Cartesian Genetic Programming," in Inspired by Nature, ser. ECC. Springer, 2018, vol. 28, pp. 235–246, doi: 10.1007/978-3-319-67997-6˙11.

[59] B. Edmonds, "Meta-Genetic Programming: Co-evolving the operators of variation," Elektrik, vol. 9, no. 1, pp. 13–29, 2001.

[60] L. N. M. N. P. N. Eduard Lukschandl, Henrik Borgvall, "Distributed java bytecode genetic programming with telecom applications," in Proc. Europgean Conference on Genetic Programming, EuroGP2000, ser. LNCS, vol. 1802. Springer, 2000, pp. 316–325, doi: 10.1007/978-3-540-46239-2˙24.

[61] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, Fundamentals of Algebraic Graph Transformation, ser. Monographs in Theoretical Computer Science. Springer, 2006, doi: 10.1007/3-540-31188-2.

[62] H. Ehrig, C. Ermel, U. Golas, and F. Hermann, Graph and Model Transformation, ser. Monographs in Theoretical Computer Science. Springer, 2015.

[63] H. Ehrig, M. Pfender, and H. J. Schneider, "Graph-grammars: An algebraic approach,"

in 14th Annual Symposium on Switching and Automata Theory, 1973, pp. 167–180, doi: 10.1109/SWAT.1973.11.

[64] A. E. Eiben and J. E. Smith, Introduction to Evolutionary Computing, 2nd ed., ser. Natural Computing Series. Springer, 2015, doi: 10.1007/978-3-662-44874-8.

[65] P. Erdős and A. Rényi, "On random graphs," Publicationes Mathematicae (Debrecen), vol. 6, pp. 290–297, 1959.

[66] R. Etxeberria, P. Larranaga, and J. M. Picaza, "Analysis of the behaviour of genetic algorithms when learning Bayesian network structure from data," Pattern Recognition Letters, vol. 18, no. 11-13, pp. 1269–1273, 1997, doi: 10.1016/S0167-8655(97)00106-2.

[67] M. Fernández, H. Kirchner, and B. Pinaud, "Strategic port graph rewriting: an interactive modelling framework," Mathematical Structures in Computer Science, vol. 29, no. 5, p. 615–662, 2019, doi: 10.1017/S0960129518000270.

[68] F. Fernández de Vega, M. Tomassini, W. F. Punch III, and J. M. Sánchez-Pérez, "Experimental study of multipopulation parallel Genetic Programming," in Proc. European Conference on Genetic Programming, EuroGP 2010, 2000, pp. 283–293, doi: 10.1007/978-3-540-46239-2_21.

[69] D. B. Fogel, Evolutionary Computation: The Fossil Record, 1st ed. Wiley-IEEE Press, 1998.

[70] F.-A. Fortin, F.-M. D. Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," J. Machine Learning Research, vol. 13, no. Jul, pp. 2171–2175, 2012.

[71] E. Galvan-Lopez, "Efficient graph-based Genetic Programming representation with multiple outputs," International J. Automation and Computing, vol. 5, no. 1, pp. 81–89, 2008, doi: 10.1007/s11633-008-0081-4.

[72] E. Galván-López, R. Poli, A. Kattan, M. O'Neill, and A. Brabazon, "Neutrality in evolutionary algorithms... what do we know?" Evolving Systems, vol. 2, no. 3, 2011, doi: 10.1007/s12530-011-9030-5.

[73] J. Gauci and K. Stanley, "Generating large-scale neural networks through discovering geometric regularities," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2007. ACM, 2007, pp. 997–1004, doi: 10.1145/1276958.1277158.

[74] J. Gauci and K. O. Stanley, "Indirect encoding of neural networks for scalable go," in

Proc. 11th International Conference on Parallel Problem Solving from Nature, PPSN XI, ser. LNCS, vol. 6238. Springer, 2010, pp. 354–363, doi: 10.1007/978-3-642-15844-5˙36.

[75] E. N. Gilbert, "Random graphs," The Annals of Mathematical Statistics, vol. 30, no. 4, pp. 1141–1144, 1959. [Online]. Available: http://www.jstor.org/stable/2237458

[76] B. W. Goldman and W. F. Punch, "Reducing wasted evaluations in Cartesian Genetic Programming," in Proc. European Conference on Genetic Programming, EuroGP 2013, ser. LNCS, vol. 7831. Springer, 2013, pp. 61–72, doi: 10.1007/978-3-642-37207-0˙6.

[77] ——, "Analysis of Cartesian Genetic Programming's evolutionary mechanisms," IEEE Trans. Evolutionary Computation, vol. 19, no. 3, pp. 359–373, 2014, doi: 10.1109/TEVC.2014.2324539.

[78] J. Gomes, P. Urbano, and A. L. Christensen, "Evolution of swarm robotics systems with novelty search," Swarm Intelligence, vol. 7, no. 2-3, pp. 115–144, 2013, doi: 10.1007/s11721-013-0081-z.

[79] F. Gomez and R. Miikkulainen, "Incremental evolution of complex general behavior," Adaptive Behavior, vol. 5, no. 3-4, pp. 317–342, 1997, doi: 10.1177/105971239700500305.

[80] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Accelerated neural evolution through cooperatively coevolved synapses," J. Machine Learning Research, vol. 9, no. May, pp. 937–965, 2008.

[81] F. J. Gomez and R. Miikkulainen, "Solving non-markovian control tasks with neuroevolution," in Proc. 16th International Joint Conference on Artificial Intelligence, IJCAI'99, vol. 2. Morgan Kaufmann, 1999, pp. 1356–1361.

[82] A. J. Graham and D. A. Pike, "A note on thresholds and connectivity in random directed graphs," Atl. Electron. J. Math, vol. 3, no. 1, pp. 1–5, 2008.

[83] C. Gyles and P. Boerlin, "Horizontally transferred genetic elements and their role in pathogenesis of bacterial disease," Veterinary pathology, vol. 51, no. 2, pp. 328–340, 2014, doi: 10.1177/0300985813511131.

[84] A. Habel, H.-J. Kreowski, and D. Plump, "Jungle evaluation," in Recent Trends in Data Type Specification, WADT'87, Selected Papers, ser. LNCS, vol. 332. Springer, 1988, pp. 92–112, doi: 10.1007/3-540-50325-0˙5.

[85] A. Habel, J. Müller, and D. Plump, "Double-pushout graph transformation revisited," Mathematical Structures in Computer Science, vol. 11, no. 5, pp. 637–688, 2001, doi: 10.1017/S0960129501003425.

[86] A. Habel and D. Plump, "Relabelling in graph transformation," in Proc. International Conference on Graph Transformation , ICGT 2002, ser. LNCS, vol. 2505. Springer, 2002, pp. 135–147, doi: 10.1007/3-540-45832-8_12.

[87] M. W. Hahn, "Toward a selection theory of molecular evolution," Evolution, vol. 62, no. 2, pp. 255–265, 2007.

[88] S. Harding, "Evolution of image filters on graphics processor units using Cartesian Genetic Programming," in Proc. IEEE Congress on Evolutionary Computation, CEC 2008. IEEE, 2008, pp. 1921–1928, doi: 10.1109/CEC.2008.4631051.

[89] S. Harding, J. Leitner, and J. Schmidhuber, "Cartesian Genetic Programming for image processing," in Genetic Programming theory and practice X, ser. GEVO. Springer, 2013, pp. 31–44, doi: 10.1007/978-1-4614-6846-2˙3.

[90] S. Harding, J. F. Miller, and W. Banzhaf, "Developments in Cartesian Genetic Programming: self-modifying CGP," Genetic Programming and Evolvable Machines, vol. 11, no. 3-4, pp. 397–439, 2010, doi: 10.1007/s10710-010-9114-1.

[91] ——, "Self modifying Cartesian Genetic Programming: finding algorithms that calculate pi and e to arbitrary precision," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2010. ACM, 2010, pp. 579–586, doi: 10.1145/1830483.1830591.

[92] I. Harvey, "The microbial genetic algorithm," in European Conference on Artificial Life, ECAL 2009, ser. LNCS, vol. 5778. Springer, 2009, pp. 126–133, doi: 10.1007/978-3-642-21314-4˙16.

[93] I. Harvey and A. Thompson, "Through the labyrinth evolution finds a way: A silicon ridge," in Proc. Evolvable Systems: From Biology to Hardware, ICES 1996, ser. LNCS, vol. 1259. Springer, 1997, pp. 406–422, doi: 10.1007/3-540-63173-9˙62.

[94] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general atari game playing," IEEE Trans. Computational Intelligence and AI in Games, vol. 6, no. 4, pp. 355–366, 2014, doi: 10.1109/TCIAIG.2013.2294713.

[95] R. Heckel, "Stochastic analysis of graph transformation systems: A case study in P2P networks," in Proc. Theoretical Aspects of Computing, ICTAC 2005, ser. LNCS, vol.

3722, 2005, pp. 53–69, doi: 10.1007/11560647˙4.

[96] ——, "Graph transformation in a nutshell," Electronic notes in theoretical computer science, vol. 148, no. 1, pp. 187–198, 2006, doi: 10.1016/j.entcs.2005.12.018.

[97] R. Heckel, G. Lajios, and S. Menge, "Stochastic graph transformation systems," Fundamenta Informaticae, vol. 74, no. 1, pp. 63–84, 2006.

[98] R. Heckel and P. Torrini, "Stochastic modelling and simulation of mobile systems," in Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, ser. LNCS, vol. 5765. Springer, 2010, pp. 87–101, doi: 10.1007/978-3-642-17322-6˙5.

[99] T. Helmuth and L. Spector, "General program synthesis benchmark suite," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2015. ACM, 2015, pp. 1039–1046, doi: 10.1145/2739480.2754769.

[100] K. Hirasawa, T. Eguchi, J. Zhou, L. Yu, J. Hu, and S. Markon, "A double-deck elevator group supervisory control system using genetic network programming," IEEE Trans. Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 38, no. 4, pp. 535–550, 2008, doi: 10.1109/TSMCC.2007.913904.

[101] J. H. Holland, "Adaptation in natural and artificial systems. an introductory analysis with application to biology, control, and artificial intelligence," Ann Arbor, MI: University of Michigan Press, 1975.

[102] L. Hong, J. R. Woodward, J. Li, and E. Özcan, "Automated design of probability distributions as mutation operators for evolutionary programming using Genetic Programming," in Proc. European Conference on Genetic Programming, EuroGP 2013, ser. LNCS, vol. 7831. Springer, 2013, pp. 85–96, doi: 10.1007/978-3-642-37207-0˙8.

[103] T. Hu and W. Banzhaf, "Neutrality and variability: Two sides of evolvability in linear Genetic Programming," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2009. ACM, 2009, pp. 963–970, doi: 10.1145/1569901.1570033.

[104] ——, Neutrality, Robustness, and Evolvability in Genetic Programming. Springer, 2018, pp. 101–117, doi: 10.1007/978-3-319-97088-2˙7.

[105] L. Huelsbergen, "Learning recursive sequences via evolution of machine-language programs," in Proc. Second Annual Conference on Genetic Programming. Morgan Kaufmann, 1997, pp. 186–194.

[106] J. Husa and R. Kalkreuth, "A comparative study on crossover in Cartesian Genetic Programming," in Proc. European Conference on Genetic Programming, EuroGP 2018, ser. LNCS, vol. 10781. Springer, 2018, pp. 203–219, doi: 10.1007/978-3-319-77553-1˙13.

[107] C. Igel, "Neuroevolution for reinforcement learning using evolution strategies," in IEEE Congress on Evolutionary Computation, CEC 2003, vol. 4, IEEE. IEEE, 2003, pp. 2588–2595, doi: 10.1109/CEC.2003.1299414.

[108] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note," Bonn, Germany: German National Research Center for Information Technology GMD Technical Report, vol. 148, no. 34, p. 13, 2001.

[109] D. Jungnickel, Graphs, Networks and Algorithms, 4th ed. Springer, 2013, doi: 10.1007/978-3-540-72780-4.

[110] V. Kabanets and J.-Y. Cai, "Circuit minimization problem," in Proc. 32nd annual ACM symposium on Theory of computing. ACM, 2000, pp. 73–79, doi: 10.1145/335305.335314.

[111] R. Kalkreuth, "Towards advanced phenotypic mutations in Cartesian Genetic Programming," arXiv preprint arXiv:1803.06127, 2018. [Online]. Available: https://arxiv.org/abs/1803.06127

[112] R. Kalkreuth, G. Rudolph, and A. Droschinsky, "A new subgraph crossover for Cartesian Genetic Programming," in Proc. European Conference on Genetic Programming, EuroGP 2017, ser. LNCS, vol. 10196. Springer, 2017, pp. 294–310, doi: 10.1007/978-3-319-55696-3˙19.

[113] W. Kantschik and W. Banzhaf, "Linear-tree gp and its comparison with other gp structures," in Proc. European Conference on Genetic Programming, EuroGP 2001, ser. LNCS, vol. 2038. Springer, 2001, pp. 302–312, doi: 10.1007/3-540-45355-5˙24.

[114] ——, "Linear-graph GP-a new GP structure," in Proc. European Conference on Genetic Programming, EuroGP 2002, ser. LNCS, vol. 2278. Springer, 2002, pp. 83–92, doi: 10.1007/3-540-45984-7˙8.

[115] D. R. Karger, "Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm," in Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1993). Society for Industrial and Applied Mathematics, 1993, pp. 21–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=313559.313605

[116] ——, "Random sampling in matroids, with applications to graph connectivity and minimum spanning trees," in Proc. 34th IEEE Annual Symposium on Foundations of Computer Science, FOCS 1993. IEEE, 1993, pp. 84–93, doi: 10.1109/SFCS.1993.366879.

[117] Y. Kassahun and G. Sommer, "Efficient reinforcement learning through evolutionary acquisition of neural topologies," in Proc. 13th European Symposium on Artificial Neural Networks, ESANN 2005, 2005, pp. 259–266. [Online]. Available: https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2005-72.pdf

[118] H. Katagiri, K. Hirasama, and J. Hu, "Genetic network programming - application to intelligent agents," in IEEE International Conference on Systems, Man and Cybernetics, SMC 2000, vol. 5. IEEE, 2000, pp. 3829–3834, doi: 10.1109/ICSMC.2000.886607.

[119] P. J. Keeling and J. D. Palmer, "Horizontal gene transfer in eukaryotic evolution," Nature Reviews Genetics, vol. 9, no. 8, p. 605, 2008.

[120] S. Kelly and M. I. Heywood, "Emergent tangled graph representations for atari game playing agents," in Proc. European Conference on Genetic Programming, EuroGP 2017, ser. LNCS, vol. 10196. Springer, 2017, pp. 64–79, doi: 10.1007/978-3-319-55696-3˙5.

[121] ——, "Multi-task learning in atari video games with emergent tangled program graphs," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2017. ACM, 2017, pp. 195–202, doi: 10.1145/3071178.3071303.

[122] C. J. Kennedy and C. Giraud-Carrier, "A depth controlling strategy for strongly typed evolutionary programming," in Proc. Genetic and Evolutionary Computation Conference, GECCO '99, vol. 1. Morgan Kaufmann, 1999, pp. 879–885. [Online]. Available: http://dl.acm.org/citation.cfm?id=2933923.2934037

[123] A. Khan, R. Heckel, P. Torrini, and I. Ráth, "Model-based stochastic simulation of P2P voip using graph transformation system," in Proc. International Conference on Analytical and Stochastic Modeling Techniques and Applications, ASMTA 2010, ser. LNCS, vol. 6148. Springer, 2010, pp. 204–217, doi: 10.1007/978-3-642-13568-2˙15.

[124] G. M. Khan, J. F. Miller, and D. M. Halliday, "Developing neural structure of two agents that play checkers using Cartesian Genetic Programming," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2008. ACM, 2008, pp. 2169–2174, doi: 10.1145/1388969.1389042.

[125] M. M. Khan, G. M. Khan, and J. F. Miller, "Efficient representation of recurrent neural networks for markovian/non-markovian non-linear control problems," in Proc. 10th International Conference on Intelligent Systems Design and Applications, (ISDA 2010). IEEE, 2010, pp. 615–620. [Online]. Available: https://doi.org/10.1109/ISDA.2010.5687197

[126] ——, "Evolution of neural networks using Cartesian Genetic Programming," in Proc. IEEE Congress on Evolutionary Computation, CEC 2010. IEEE, 2010, pp. 1–8, doi: 10.1109/CEC.2010.5586547.

[127] M. Kimura, The neutral theory of molecular evolution. Cambridge University Press, 1983.

[128] J. Koutnik, F. Gomez, and J. Schmidhuber, "Evolving neural networks in compressed weight space," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2010. ACM, 2010, pp. 619–626, doi: 10.1145/1830483.1830596.

[129] J. R. Koza, Genetic Programming: on the programming of computers by means of natural selection. MIT Press, 1993.

[130] ——, "Genetic Programming as a means for programming computers by natural selection," Statistics and Computing, vol. 4, no. 2, pp. 87–112, Jun 1994.

[131] J. R. Koza, F. H. B. III, and O. Stiffelman, "Genetic Programming as a Darwinian invention machine," in Proc. European Conference on Genetic Programming, EuroGP 1999, ser. LNCS, vol. 1598. Springer, 1999, pp. 93–108, doi: 10.1007/3-540-48885-5˙8.

[132] C. Krause and H. Giese, "Probabilistic graph transformation systems," in Proc. International Conference on Graph Transformation, ICGT 2012, ser. LNCS, vol. 7562. Springer, 2012, pp. 311–325, doi: 10.1007/978-3-642-33654-6˙21.

[133] P. Larrañaga, H. Karshenas, C. Bielza, and R. Santana, "A review on evolutionary algorithms in Bayesian network learning and inference tasks," Information Sciences, vol. 233, pp. 109 – 125, 2013, doi: 10.1016/j.ins.2012.12.051.

[134] P. Larrañaga, M. Poza, Y. Yurramendi, R. H. Murga, and C. M. H. Kuijpers, "Structure learning of Bayesian networks by genetic algorithms: A performance analysis of control parameters," IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 18, no. 9, pp. 912–926, 1996, doi: 10.1109/34.537345.

[135] K. S. Leung, K. H. Lee, and S. M. Cheang, "Evolving parallel machine programs for

a multi-alu processor," in Proc. IEEE Congress on Evolutionary Computation, CEC 2002, vol. 2. IEEE, 2002, pp. 1703–1708, doi: 10.1109/CEC.2002.1004499.

[136] J. Liang, E. Meyerson, and R. Miikkulainen, "Evolutionary architecture search for deep multitask networks," in Proc. Genetic and Evolutionary Computation Conference. ACM, 2018, pp. 466–473, doi: 10.1145/3205455.3205489.

[137] Y. Liu, G. Tempesti, J. A. Walker, J. Timmis, A. M. Tyrrell, and P. Bremner, "A self-scaling instruction generator using Cartesian Genetic Programming," in Proc. European Conference on Genetic Programming, EuroGP 2011, ser. LNCS, vol. 6621. Springer, 2011, pp. 298–309, doi: 10.1007/978-3-642-20407-4_26.

[138] E. G. López and K. Rodríguez-Vázquez, "Multiple interactive outputs in a single tree: An empirical investigation," in Proc. European Conference on Genetic Programming, EuroGP 2007, ser. LNCS, vol. 4445. Springer, 2007, pp. 341–350, doi: 10.1007/978-3-540-71605-1˙32.

[139] L. A. Lorena, M. G. Narciso, and J. Beasley, "A constructive genetic algorithm for the generalized assignment problem," Evolutionary Optimization, vol. 5, pp. 1–19, 2002.

[140] M. Löwe, "Algebraic approach to single-pushout graph transformation," Theoretical Computer Science, vol. 109, no. 1&2, pp. 181–224, 1993, doi: 10.1016/0304-3975(93)90068-5.

[141] S. Luke, Essentials of Metaheuristics, 2nd ed. Lulu, 2013. [Online]. Available: http://cs.gmu.edu/$\sim$sean/book/metaheuristics/

[142] S. Luke and L. Panait, "A comparison of bloat control methods for Genetic Programming," Evolutionary Computation, vol. 14, no. 3, pp. 309–344, 2006, doi: 10.1162/evco.2006.14.3.309.

[143] S. Luke and L. Spector, "A comparison of crossover and mutation in Genetic Programming," in Proc. Second Annual Conference on Genetic Programming. Morgan Kaufmann, 1997, pp. 240–248.

[144] P. Machado, F. B. Pereira, J. Tavares, E. Costa, and A. Cardoso, "Evolutionary Turing machines: The quest for busy beavers," in Recent Developments in Biologically Inspired Computing. Idea Group Publishing, 2004, ch. 2, doi: 10.4018/978-1-59140-312-8.ch002.

[145] R. Machado, L. Ribeiro, and R. Heckel, "Rule-based transformation of graph rewriting rules: Towards higher-order graph grammars," Theoretical Computer Science, vol. 594,

pp. 1–23, 2015, doi: 10.1016/j.tcs.2015.01.034.

[146] K. L. Mak, Y. S. Wong, and X. X. Wang, "An adaptive genetic algorithm for manufacturing cell formation," The International J. Advanced Manufacturing Technology, vol. 16, no. 7, pp. 491–497, 2000, doi: 10.1007/s001700070057.

[147] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," Ann. Math. Statist., vol. 18, no. 1, pp. 50–60, 1947.

[148] M. Mascherini and F. M. Stefanini, "M-ga: A genetic algorithm to search for the best conditional gaussian Bayesian network," in Proc. International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, CIMCA-IAWTIC'06, vol. 2. IEEE, 2005, pp. 61–67, doi: 10.1109/CIMCA.2005.1631446.

[149] P. Massey, J. A. Clark, and S. Stepney, "Evolving quantum circuits and programs through Genetic Programming," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2004, ser. LNCS, vol. 3103. Springer, 2004, pp. 569–580, doi: 10.1007/978-3-540-24855-2˙66.

[150] P. McCombie and P. Wilkinson, "The use of the simple genetic algorithm in finding the critical factor of safety in slope stability analysis," Computers and Geotechnics, vol. 29, no. 8, pp. 699–714, 2002, doi: 10.1016/S0266-352X(02)00027-7.

[151] A. Meduna, Formal languages and computation: models and their applications. Auerbach Publications, 2014.

[152] E. Mezura-Montes and C. A. C. Coello, "A simple multimembered evolution strategy to solve constrained optimization problems," IEEE Trans. Evolutionary Computation, vol. 9, no. 1, pp. 1–17, 2005, doi: 10.1109/TEVC.2004.836819.

[153] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy et al., "Evolving deep neural networks," in Artificial Intelligence in the Age of Neural Networks and Brain Computing. Elsevier, 2019, pp. 293–312, doi: 10.1016/B978-0-12-815480-9.00015-3.

[154] J. F. Miller, "An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach," in Proc. Genetic and Evolutionary Computation Conference, GECCO 1999, vol. 2. Morgan Kaufmann, 1999, pp. 1135–

1142.

[155] J. F. Miller, Ed., Cartesian Genetic Programming. Springer, 2011, doi: 10.1007/978-3-642-17310-3.

[156] J. F. Miller and S. L. Smith, "Redundancy and computational efficiency in Cartesian Genetic Programming," IEEE Trans. Evolutionary Computation, vol. 10, no. 2, pp. 167–174, 2006, doi: 10.1109/TEVC.2006.871253.

[157] J. F. Miller and P. Thomson, "Cartesian Genetic Programming," in Proc. European Conference on Genetic Programming, EuroGP 2000, ser. LNCS, vol. 1802. Springer, 2000, pp. 121–132, doi: 10.1007/978-3-540-46239-2_9.

[158] J. F. Miller, "Cartesian Genetic Programming: its status and future," Genetic Programming and Evolvable Machines, 2019, doi: 10.1007/s10710-019-09360-6.

[159] M. Molloy and B. Reed, "A critical point for random graphs with a given degree sequence," Random structures & algorithms, vol. 6, no. 2-3, pp. 161–180, 1995.

[160] D. J. Montana, "Strongly typed Genetic Programming," Evolutionary computation, vol. 3, no. 2, pp. 199–230, 1995, doi: 10.1162/evco.1995.3.2.199.

[161] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," in Proc. 11th International Joint Conference on Artificial Intelligence, IJCAI'89, vol. 1, 1989, pp. 762–767. [Online]. Available: http://ijcai.org/Proceedings/89-1/Papers/122.pdf

[162] A. Moraglio, K. Krawiec, and C. G. Johnson, "Geometric semantic Genetic Programming," in Parallel Problem Solving from Nature, PPSN XII, ser. LNCS, vol. 7491. Springer, 2012, pp. 21–31, doi: 10.1007/978-3-642-32937-1˙3.

[163] D. E. Moriarty, "Symbiotic evolution of neural networks in sequential decision tasks," Ph.D. dissertation, University of Texas at Austin USA, 1997.

[164] R. Motwani and P. Raghavan, Randomized Algorithms. Cambridge University Press, 1995.

[165] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy, "Design of power-efficient approximate multipliers for approximate artificial neural networks," in Proc. IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2016. IEEE, 2016, pp. 1–7, doi: 10.1145/2966986.2967021.

[166] J. Muruzábal and C. Cotta, "A primer on the evolution of equivalence classes of Bayesian-network structures," in Parallel Problem Solving from Nature, PPSN VIII, ser. LNCS, vol. 3242. Springer, 2004, pp. 612–621, doi: 10.1007/978-3-540-30217-9˙62.

[167] A. Naidoo and N. Pillay, "Using Genetic Programming for Turing machine induction," in Proc. European Conference on Genetic Programming, EuroGP 2008, ser. LNCS, vol. 4971. Springer, 2008, pp. 350–361, doi: 10.1007/978-3-540-78671-9˙30.

[168] Q. U. Nguyen, X. H. Nguyen, and M. O'Neill, "Semantic aware crossover for Genetic Programming: The case for real-valued function regression," in Proc. European Conference on Genetic Programming, EuroGP 2009, ser. LNCS, vol. 5481. Springer, 2009, pp. 292–302, doi: 10.1007/978-3-642-01181-8˙25 .

[169] M. Nicolau, A. Agapitos, M. O'Neill, and A. Brabazon, "Guidelines for defining benchmark problems in Genetic Programming," in Proc. IEEE Congress on Evolutionary Computation, CEC 2015, May 2015, pp. 1152–1159, doi: 10.1109/CEC.2015.7257019.

[170] M. Nishiguchi and Y. Fujimoto, "Evolution of recursive programs with multi-niche Genetic Programming (mnGP)," in Proc. 1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE, 1998, pp. 247–252, doi: 10.1109/ICEC.1998.699720.

[171] P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," in Advances in Genetic Programming. MIT Press, 1994, ch. 14, pp. 311–331.

[172] P. Nordin and W. Banzhaf, "Evolving turing-complete programs for a register machine with self-modifying code." in Proc. International Conference on Genetic Algorithms, ICGA 1995. Morgan Kaufmann, 1995, pp. 318–325.

[173] J. R. Norris, Markov chains, ser. Cambridge series in statistical and probabilistic mathematics. Cambridge University Press, 1998.

[174] M. Oltean, "Evolving evolutionary algorithms using linear genetic programming," Evolutionary Computation, vol. 13, no. 3, pp. 387–410, 2005, doi: 10.1162/1063656054794815.

[175] M. O'Neill and C. Ryan, "Grammatical evolution," IEEE Trans. Evolutionary Computation, vol. 5, no. 4, pp. 349–358, 2001, doi: 10.1109/4235.942529.

[176] W. Palacz, "Algebraic hierarchical graph transformation." J. Computer and System

Science, vol. 68, no. 3, pp. 497–520, 2004, doi: 10.1016/S0022-0000(03)00064-3.

[177] F. B. Pereira, P. Machado, E. Costa, and A. Cardoso, "Graph based crossover – a case study with the busy beaver problem," in Proc. Genetic and Evolutionary Computation Conference, GECCO 1999. Morgan Kaufmann, 1999, pp. 1149–1155.

[178] T. A. Pham, Q. U. Nguyen, X. H. Nguyen, and M. O'Neill, "Examining the diversity property of semantic similarity based crossover," in Proc. European Conference on Genetic Programming, EuroGP 2013, ser. LNCS, vol. 7831. Springer, 2013, pp. 265–276, doi: 10.1007/978-3-642-37207-0˙23.

[179] S. Picek, C. Carlet, S. Guilley, J. F. Miller, and D. Jakobovic, "Evolutionary algorithms for boolean functions in diverse domains of cryptography," Evolutionary computation, vol. 24, no. 4, pp. 667–694, 2016.

[180] S. Picek, D. Jakobovic, J. F. Miller, L. Batina, and M. Cupic, "Cryptographic boolean functions: One output, many design criteria," Applied Soft Computing, vol. 40, pp. 635–653, 2016, doi: 10.1016/j.asoc.2015.10.066.

[181] D. Plump, "Term graph rewriting," in Handbook of Graph Grammars and Computing by Graph Transformation, H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds. World Scientific, 1999, vol. 2, ch. 1, pp. 3–61, doi: 10.1142/9789812815149_0001.

[182] ——, "The design of GP 2," in Proc. Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, ser. EPTCS, vol. 82, 2012, pp. 1–16, doi: 10.4204/EPTCS.82.1.

[183] ——, "Reasoning about graph programs," in Proc. Computing with Terms and Graphs, TERMGRAPH 2016, ser. EPTCS, vol. 225, 2016, pp. 35–44, doi: 10.4204/EPTCS.225.6.

[184] ——, "From imperative to rule-based graph programs," J. Logical and Algebraic Methods in Programming, vol. 88, pp. 154–173, 2017, doi: 10.1016/j.jlamp.2016.12.001.

[185] D. Plump and S. Steinert, "Towards graph programs for graph algorithms," in Proc. International Conference on Graph Transformation, ICGT 2004, ser. LNCS, vol. 3256. Springer, 2004, pp. 128–143, doi: 10.1007/978-3-540-30203-2˙11.

[186] R. Poli, Parallel distributed Genetic Programming. University of Birmingham, Cognitive Science Research Centre, 1996.

[187] ——, "Some steps towards a form of Parallel Distributed Genetic Programming," in

Proc. First On-line Workshop on Soft Computing, 1996, pp. 290–295.

[188] ——, "Evolution of graph-like programs with parallel distributed Genetic Programming," in Proc. International Conference on Genetic Algorithms, ICGA 1997. Morgan Kaufmann, 1997, pp. 346–353.

[189] ——, "Parallel Distributed Genetic Programming," in New Ideas in Optimization. McGraw-Hill, 1999, pp. 403–431.

[190] ——, "A simple but theoretically-motivated method to control bloat in genetic programming," in Proc. European Conference on Genetic Programming, EuroGP 2003, ser. LNCS, vol. 2610. Springer, 2003, pp. 204–217, doi: 10.1007/3-540-36599-0˙19.

[191] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, A field guide to Genetic Programming. Lulu. com, 2008. [Online]. Available: https://dces.essex.ac.uk/staff/rpoli/gp-field-guide/A_Field_Guide_to_Genetic_Programming.pdf

[192] J. C. F. Pujol and R. Poli, "Evolving the topology and the weights of neural networks using a dual representation," Applied Intelligence, vol. 8, no. 1, pp. 73–84, 1998, doi: 10.1023/A:1008272615525.

[193] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in Proc. the AAAI Conference on Artificial Intelligence, vol. 33, 2019, pp. 4780–4789, doi: 10.1609/aaai.v33i01.33014780.

[194] C. R. Reeves, "A genetic algorithm for flowshop sequencing," Computers & Operations Research, vol. 22, no. 1, pp. 5–13, 1995, doi: 10.1016/0305-0548(93)E0014-K.

[195] C. W. Rempis, "Evolving complex neuro-controllers with interactively constrained neuro-evolution," Ph.D. dissertation, University of Osnabrück, 2012. [Online]. Available: http://d-nb.info/1030399018

[196] F. Rothlauf and D. E. Goldberg, "Pruefer numbers and genetic algorithms: A lesson on how the low locality of an encoding can harm the performance of gas," in Proc. Parallel Problem Solving from Nature, PPSN VI, ser. LNCS, vol. 1917. Springer, 2000, pp. 395–404, doi: 10.1007/3-540-45356-3_39.

[197] G. Rozenberg and A. Salomaa, The mathematical theory of L systems. Academic press, 1980.

[198] G. Rudolph, "Global optimization by means of distributed evolution strategies," in Proc. Parallel Problem Solving from Nature, PPSN I, ser. LNCS, vol. 496. Springer,

1990, pp. 209–213, doi: 10.1007/BFb0029754.

[199] C. Ryan, J. J. Collins, and M. O'Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in Proc. European Conference on Genetic Programming, EuroGP 1998, ser. LNCS, vol. 1391. Springer, 1998, pp. 83–96, doi: 10.1007/BFb0055930.

[200] J. A. Schwartz, N. E. Curtis, and S. K. Pierce, "Fish labeling reveals a horizontally transferred algal (*Vaucheria litorea*) nuclear gene on a sea slug (*Elysia chlorotica*) chromosome," The Biological Bulletin, vol. 227, no. 3, pp. 300–312, 2014, doi: 10.1086/BBLv227n3p300.

[201] L. Sekanina, S. L. Harding, W. Banzhaf, and T. Kowaliw, "Image processing and CGP," in Cartesian Genetic Programming. Springer, 2011, pp. 181–215, doi: 10.1007/978-3-642-17310-3˙6.

[202] R. Serfozo, Basics of applied stochastic processes. Springer, 2009, doi: 10.1007/978-3-540-89332-5.

[203] S. Shirakawa and T. Nagao, "Evolution of sorting algorithm using graph structured program evolution," in 2007 IEEE International Conference on Systems, Man and Cybernetics, SMC 2007. IEEE, 2007, pp. 1256–1261, doi: 10.1109/ICSMC.2007.4413828.

[204] ——, "Graph structured program evolution with automatically defined nodes," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2009. ACM, 2009, pp. 1107–1114, doi: 10.1145/1569901.1570050.

[205] ——, "Graph structured program evolution: Evolution of loop structures," in Genetic Programming Theory and Practice VII, ser. GEVO. Springer, 2010, pp. 177–194, doi: 10.1007/978-1-4419-1626-6˙11.

[206] S. Shirakawa, S. Ogino, and T. Nagao, "Graph structured program evolution," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2007. ACM, 2007, pp. 1686–1693, doi: 10.1145/1276958.1277290.

[207] N. T. Siebel and G. Sommer, "Evolutionary reinforcement learning of artificial neural networks," International J. Hybrid Intelligent Systems, vol. 4, no. 3, pp. 171–183, 2007, doi: 10.3233/HIS-2007-4304.

[208] S. Silva and E. Costa, "Resource-limited Genetic Programming: the dynamic ap-

proach," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2005. ACM, 2005, pp. 1673–1680, doi: 10.1145/1068009.1068290.

[209] S. Skiena, The Algorithm Design Manual, 2nd ed. Springer, 2008.

[210] R. J. Smith and M. I. Heywood, "Scaling tangled program graphs to visual reinforcement learning in vizdoom," in Proc. European Conference on Genetic Programming, EuroGP 2018, ser. LNCS, vol. 10781. Springer, 2018, pp. 135–150, doi: 10.1007/978-3-319-77553-1˙9.

[211] D. Snyder, A. Goudarzi, and C. Teuscher, "Finding optimal random boolean networks for reservoir computing," in Proc. International Conference on Artificial Life, ALIFE 2012. MIT Press, 2012, pp. 259–266, doi: 10.7551/978-0-262-31050-5-ch035.

[212] L. Spector, "Autoconstructive evolution: Push, pushGP, and pushpop," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2001. Morgan Kaufmann, 2001, pp. 137–146.

[213] L. Spector, H. Barnum, H. J. Bernstein, and N. Swamy, "Quantum computing applications of Genetic Programming," Advances in Genetic Programming, vol. 3, pp. 135–160, 1999.

[214] L. Spector and J. Klein, "Machine invention of quantum computing circuits by means of Genetic Programming," Artificial Intelligence for Engineering Design, Analysis and Manufacturing, vol. 22, no. 3, pp. 275–283, 2008.

[215] L. Spector, B. Martin, K. Harrington, and T. Helmuth, "Tag-based modules in genetic programming," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2011. ACM, 2011, pp. 1419–1426, doi: 10.1145/2001576.2001767.

[216] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," IEEE Computer, vol. 27, no. 6, pp. 17–26, 1994, doi: 10.1109/2.294849.

[217] R. Stadelhofer, W. Banzhaf, and D. Suter, "Evolving blackbox quantum algorithms using Genetic Programming," Artificial Intelligence for Engineering Design, Analysis and Manufacturing, vol. 22, no. 3, pp. 285–297, 2008.

[218] K. O. Stanley, "Compositional pattern producing networks: A novel abstraction of development," Genetic Programming and evolvable machines, vol. 8, no. 2, pp. 131–162, 2007, doi: 10.1007/s10710-007-9028-8.

[219] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the

NERO video game," IEEE Trans. Evolutionary Computation, vol. 9, no. 6, pp. 653–668, 2005, doi: 10.1109/TEVC.2005.856210.

[220] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," Artificial life, vol. 15, no. 2, pp. 185–212, 2009, doi: 10.1162/artl.2009.15.2.15202.

[221] K. O. Stanley and R. Miikkulainen, "Efficient reinforcement learning through evolving neural network topologies," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2002. Morgan Kaufmann, 2002, pp. 569–577.

[222] ——, "Evolving neural networks through augmenting topologies," Evolutionary Computation, vol. 10, no. 2, pp. 99–127, 2002, doi: 10.1162/106365602320169811.

[223] K. O. Stanley, "Efficient evolution of neural networks through complexification," Ph.D. dissertation, The University of Texas at Austin, 2004. [Online]. Available: http://nn.cs.utexas.edu/?stanley:phd2004

[224] S. Steinert, "The graph programming language GP," Ph.D. dissertation, The University of York, 2007. [Online]. Available: http://www.cs.york.ac.uk/ftpdir/reports/2007/YCST/15/YCST-2007-15.pdf

[225] S. Stepney and J. A. Clark, "Searching for quantum programs and quantum protocols," J. Computational and Theoretical Nanoscience, vol. 5, no. 5, pp. 942–969, 2008, doi: 10.1166/jctn.2008.2535.

[226] M. Suganuma, S. Shirakawa, and T. Nagao, "A Genetic Programming approach to designing convolutional neural network architectures," in Proc. of the Genetic and Evolutionary Computation Conference, GECCO 2017. ACM, 2017, pp. 497–504, doi: 10.1145/3071178.3071229.

[227] J. Swan, P. De Causmaecker, S. Martin, and E. Özcan, A Re-characterization of Hyper-Heuristics. Springer, 2018, pp. 75–89, doi: 10.1007/978-3-319-58253-5˙5.

[228] J. Swan, K. Krawiec, and Z. A. Kocsis, "Stochastic synthesis of recursive functions made easy with bananas, lenses, envelopes and barbed wire," Genetic Programming and Evolvable Machines, vol. 20, no. 3, pp. 327–350, 2019, doi: 10.1007/s10710-019-09347-3.

[229] É. D. Taillard, L. M. Gambardella, M. Gendreau, and J. Potvin, "Adaptive memory programming: A unified view of metaheuristics," European J. Operational Research,

vol. 135, no. 1, pp. 1–16, 2001, doi: 10.1016/S0377-2217(00)00268-X.

[230] J. Tanomaru and A. Azuma, "Automatic generation of Turing machines by a genetic approach," in Proc. The First International Workshop on Machine Learning, Forecasting, and Optimization, MALFO96, 1996, pp. 173–184. [Online]. Available: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/tanomaru_1996_tm.pdf

[231] A. Teller and M. Veloso, "Pado: Learning tree structured algorithms for orchestration into an object recognition system." Carnegie-Mellon Univ Pittsburg PA Dept of Computer Science, Tech. Rep., 1995.

[232] ——, "Pado: A new learning architecture for object recognition," Symbolic visual learning, pp. 81–116, 1996.

[233] Tina Yu, "Structure abstraction and genetic programming," in Proc. IEEE Congress on Evolutionary Computation, CEC 1999, vol. 1. IEEE, 1999, pp. 652–659, doi: 10.1109/CEC.1999.781995.

[234] P. Torrini, R. Heckel, and I. Ráth, "Stochastic simulation of graph transformation systems," in Proc. Fundamental Approaches to Software Engineering, FASE 2010, ser. LNCS, vol. 6013. Springer, 2010, pp. 154–157, doi: 10.1007/978-3-642-12029-9˙11.

[235] Y. Tsoy and V. Spitsyn, "Using genetic algorithm with adaptive mutation mechanism for neural networks design and training," in Proce. 9th Russian-Korean International Symposium on Science and Technology, KORUS 2005. IEEE, 2005, pp. 709–714, doi: 10.1109/KORUS.2005.1507882.

[236] A. Turner, "Evolving artificial neural networks using Cartesian Genetic Programming," Ph.D. dissertation, University of York, 2015. [Online]. Available: http://etheses.whiterose.ac.uk/12035/1/thesis.pdf

[237] A. Turner and J. Miller, "Cartesian Genetic Programming: Why no bloat?" in Proc. European Conference on Genetic Programming, EuroGP 2014, ser. LNCS, vol. 8599. Springer, 2014, pp. 222–233.

[238] A. J. Turner and J. F. Miller, "Cartesian Genetic Programming encoded artificial neural networks: a comparison using three benchmarks," in Proc. Genetic and Evolutionary Computation Conference, GECCO 2013. ACM, 2013, pp. 1005–1012, doi: 10.1145/2463372.2463484.

[239] ——, "The importance of topology evolution in neuroevolution: A case study us-

ing Cartesian Genetic Programming of artificial neural networks," in Research and Development in Intelligent Systems XXX, SGAI 2013. Springer, 2013, pp. 213–226, doi: 10.1007/978-3-319-02621-3˙15.

[240] ——, "Cartesian Genetic Programming: Why no bloat?" in Proc. European Conference on Genetic Programming, EuroGP 2014, ser. LNCS, vol. 8599. Springer, 2014, pp. 222–233, doi: 10.1007/978-3-662-44303-3˙19.

[241] ——, "Recurrent Cartesian Genetic Programming," in Proc. Parallel Problem Solving from Nature, PPSN 2014, ser. LNCS, vol. 8672. Springer, 2014, pp. 476–486, doi: 10.1007/978-3-319-10762-2˙47.

[242] ——, "Recurrent Cartesian Genetic Programming applied to famous mathematical sequences," in Proc. Seventh York Doctoral Symposium on Computer Science & Electronics, YDS 2014, 2014, pp. 37–46.

[243] ——, "Introducing a cross platform open source Cartesian Genetic Programming library," Genetic Programming and Evolvable Machines, vol. 16, no. 1, pp. 83–91, 2015, doi: 10.1007/s10710-014-9233-1.

[244] ——, "Neutral genetic drift: an investigation using Cartesian Genetic Programming," Genetic Programming and Evolvable Machines, vol. 16, no. 4, pp. 531–558, 2015, doi: 10.1007/s10710-015-9244-6.

[245] ——, "Recurrent Cartesian Genetic Programming of artificial neural networks," Genetic Programming and Evolvable Machines, vol. 18, no. 2, pp. 185–212, 2017, doi: 10.1007/s10710-016-9276-6.

[246] L. Vanneschi, M. Castelli, and S. Silva, "A survey of semantic methods in Genetic Programming," Genetic Programming and Evolvable Machines, vol. 15, no. 2, pp. 195–214, Jun 2014, doi: 10.1007/s10710-013-9210-0.

[247] L. Vanneschi, Y. Pirola, G. Mauri, M. Tomassini, P. Collard, and S. Verel, "A study of the neutrality of boolean function landscapes in Genetic Programming," Theoretical Computer Science, vol. 425, pp. 34 – 57, 2012, doi: 10.1016/j.tcs.2011.03.011.

[248] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," J. Educational and Behavioral Statistics, vol. 25, no. 2, pp. 101–132, 2000.

[249] Z. Vasicek, "Cartesian GP in optimization of combinational circuits with hundreds of in-

puts and thousands of gates," in Proc. European Conference on Genetic Programming, EuroGP 2015, ser. LNCS, vol. 9025. Springer, 2015, pp. 139–150, doi: 10.1007/978-3-319-16501-1˙12.

[250] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," IEEE Trans. Evolutionary Computation, vol. 19, no. 3, pp. 432–444, 2014, doi: 10.1109/TEVC.2014.2336175.

[251] V. K. Vassilev and J. F. Miller, "The advantages of landscape neutrality in digital circuit evolution," in Proc. 3rd International Conference on Evolvable Systems, ICES 2000, ser. LNCS, vol. 1801. Springer, 2000, pp. 252–263, doi: 10.1007/3-540-46406-9˙25.

[252] M. D. Vose and A. H. Wright, "The simple genetic algorithm and the walsh transform: Part i, theory," Evolutionary Computation, vol. 6, no. 3, pp. 253–273, 1998, doi: 10.1162/evco.1998.6.3.253.

[253] J. A. Walker and J. F. Miller, "The automatic acquisition, evolution and reuse of modules in Cartesian Genetic Programming," IEEE Trans. Evolutionary Computation, vol. 12, no. 4, pp. 397–417, 2008, doi: 10.1109/TEVC.2007.903549.

[254] J. Walker and J. Miller, "Evolution and acquisition of modules in Cartesian Genetic Programming," Proc. European Conference on Genetic Programming, EuroGP 2004, vol. 3003, pp. 187–197, 2004, doi: 10.1007/978-3-540-24650-3˙17.

[255] E. A. Wan, "Time series prediction by using a connectionist network with internal delay lines," in Time Series Prediction, vol. 15. Addison-Wesley, 1993, pp. 195–195.

[256] D. Whitley, S. Rana, and R. B. Heckendorn, "The island model genetic algorithm: On separability, population size and convergence," CIT. J. computing and information technology, vol. 7, no. 1, pp. 33–47, 1999. [Online]. Available: http://cit.fer.hr/index.php/CIT/article/view/2919/1783

[257] L. D. Whitley, T. Starkweather, and C. Bogart, "Genetic algorithms and neural networks: optimizing connections and connectivity," Parallel Computing, vol. 14, no. 3, pp. 347–361, 1990, doi: 10.1016/0167-8191(90)90086-O.

[258] A. P. Wieland, "Evolving controls for unstable systems," in Connectionist Models. Elsevier, 1991, pp. 91–102.

[259] ——, "Evolving neural network controllers for unstable systems," in Seattle International Joint Conference on Neural Networks, IJCNN 91, vol. 2. IEEE, 1991,

pp. 667–673, doi: 10.1109/IJCNN.1991.155416.

[260] D. Wierstra, A. Foerster, J. Peters, and J. Schmidhuber, "Solving deep memory POMDPs with recurrent policy gradients," in Proc. Artificial Neural Networks, ICANN 2007, ser. LNCS, vol. 4668. Springer, 2007, pp. 697–706, doi: 10.1007/978-3-540-74690-4˙71.

[261] M. L. Wong, W. Lam, and K. S. Leung, "Using evolutionary programming and minimum description length principle for data mining of Bayesian networks," IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 21, no. 2, pp. 174–178, 1999, doi: 10.1109/34.748825.

[262] M. L. Wong and K. S. Leung, "Learning recursive functions from noisy examples using generic Genetic Programming," in Proc. First Annual Conference on Genetic Programming. MIT Press, 1996, pp. 238–246.

[263] J. R. Woodward and J. Swan, "The automatic generation of mutation operators for genetic algorithms," in Proc. Genetic and Evolutionary Computation Conference (GECCO 12). ACM, 2012, pp. 67–74, doi: 10.1145/2330784.2330796.

[264] X. Yao and Y. Liu, "A new evolutionary system for evolving artificial neural networks," IEEE Trans. neural networks, vol. 8, no. 3, pp. 694–713, 1997, doi: 10.1109/72.572107.

[265] S. Yoshida, S. Maruyama, H. Nozaki, and K. Shirasu, "Horizontal gene transfer by the parasitic plant *Striga hermonthica*," Science, vol. 328, no. 5982, p. 1128, 2010, doi: 10.1126/science.1187145.

[266] T. Yu and J. Miller, "Finding needles in haystacks is not hard with neutrality," in Proc. European Conference on Genetic Programming, EuroGP 2002, ser. LNCS, vol. 2278. Springer, 2002, pp. 13–25, doi: 10.1007/3-540-45984-7˙2.

[267] J. Zhang, "Evolution by gene duplication: an update," Trends in ecology & evolution, vol. 18, no. 6, pp. 292–298, 2003, doi: 10.1016/S0169-5347(03)00033-8.

[268] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength Pareto evolutionary algorithm," TIK-report, 2001, doi: 10.3929/ethz-a-004284029.