Random Graph Generation in Hyperedge Replacement Languages

Federico Vastarini

PhD

University of York

Computer Science

February 2024

Abstract

We present a novel approach for the random generation of graphs in context-free hypergraph languages. It is obtained by adapting both of Mairson's generation algorithms for context-free string grammars to the setting of hyperedge replacement grammars. It provides a concrete instrument for the generation of unbiased graph data where a solid mathematical proof is required for the validation of procedures, filling an important gap in the field of random testing. Our main result is that for non-ambiguous hyperedge replacement grammars, the method is guaranteed to efficiently generate hypergraphs uniformly at random in user-specified domains. It means that testing for the sought properties in the generated graph is no longer required since they are directly inferred by the grammar. The efficiency of the method is ensured by the proofs of polynomial time and space asymptotic behaviors. Our secondary result is that it greatly extends the range of either context-free and non-context-free string languages to sample from with a uniform distribution through the use of string graph grammars. We prove how ambiguous string grammars can be expressed with equivalent non-ambiguous hyperedge replacement grammars, overcoming the current limitation for the achievement of the uniformity of the sampling. Our contribution also proposes several case studies of relevant hyperedge replacement languages proving the existence of a representative grammar for a uniform sampling, or, otherwise, their inherent ambiguity by the analysis of particular structures. These languages form a basis for a proof by reduction for more complex languages, greatly improving the possible search for non-ambiguous solutions.

Contents

List of Figures

List of Tables

Acknowledgements

Thanks to my Supervisor, Professor Detlef Plump, for the guidance and patience and for proving that there is always some space for striving towards perfection.

Thanks to my Assessor, Professor Susan Stepney, for the listening and suggestions and for helping me reach the end of this tunnel called PhD.

...in loving memory of my father.

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 a degree or other qualification at this University or elsewhere. All sources are acknowledged as references.

A conference paper, referring to Chapter 3 on the adaptation of the first of Mairson's methods [57], presented at the 13th International Workshop on Developments in Computational Models in 2023, has been approved for publication and will appear soon in the Electronic Proceedings in Theoretical Computer Science.

Chapter 1

Introduction

This work presents a novel approach for the generation of random hypergraphs in user-specified domains. Our approach extends the methods of Mairson for generating strings in context-free languages [47] to the setting of context-free hypergraph languages specified by hyperedge replacement grammars [14, 33].

Graphs find their application in a vast variety of fields, their origin dating back to 17th century with the formulation of the famous Seven Bridges of Königsberg problem [23]. They may be used to represent any kind of abstract or concrete relations and connections among elements of any domain [21, 42].

The study of their random generation is constantly evolving [1, 6, 10]. Several algorithms already exist for the sampling of graphs addressing specific problems. We must consider three fundamental parameters when studying their stochastic processes:

- the range of graphs they can generate and if the properties of the resulting graph.
- the probability distribution of the sampling
- the termination of the generating algorithm

In [52], [58], [27] and [50] we may find powerful methods for the random generation of graphs, but none of them guarantee all these parameters at once. For example relying on edge rewriting drastically reduces the possibility of guaranteeing the sought properties of the generated graph, while relying only on the probability of a graph transformation rule to be applied does not guarantee the termination of the algorithm. We know that testing for a sought property may be in some cases expensive, even more than the generating process itself [44]. Obtaining a uniform distribution of the sampled graph instead of the applied rules requires additional computation that cannot be solved by randomly assigning a probability to the rules.

Our approach, instead, guarantees at the same time the termination of the algorithm, the properties of the generated graph and a uniform distribution of the sampling, according to the grammar and size of the generated graph specified by the user.

We know from [25, 36] that hypergraphs are decomposable structures, that is, they can be deconstructed in simpler parts, usually referred to as subgraphs. As explained later in this work, this is a fundamental notion for our methods to work, since the resulting graph of our generation process is constructively assembled according to a list of rules.

In [12] it is proven that the concept of context-freeness is applicable to hyperedge replacement languages. Our work focuses on context-free grammars, that is, a set of rules that are applied to single edges of a graph independent of their surrounding elements. The main advantage of working with such grammars is that it is possible to predict the available productions at each step of the derivation. This is an indispensable requirement for creating the data our generation process is based on.

This allows us to present a Chomsky normal form [9] similar to the one defined for strings, for the representation of the input grammar that suits the needs of the generation algorithm. We are reminded that a grammar is in normal form when only a definite set of rules is allowed. In case of strings, we know that such a form only allows for the productions to generate exactly 2 non-terminals or 1 terminal, with the exception of the starting symbol that can also allow an empty production if it is not present in the right hand side of any other production. In case of hypergraphs a similar Chomsky normal form, based instead on the number of hyperedges created by each production, ensures the termination of the generation algorithm.

We then focus our study on the probability distribution [45] of the results of our sampling methods. In order to ensure that a graph is generated over a uniform distribution a system is needed to discern among different graphs. Similarly to [18] we propose a system to produce ordered derivation trees that allows the extension of the notion of leftmost derivation to hyperedge replacement grammars. We can then relate the ambiguity [28] of a grammar to the uniqueness of its leftmost derivations and, consequently, the probability distribution of the sampling.

We prove that our methods have the same polynomial space and time complexity as their string counterparts. Most importantly a proof of uniformity of the probability distribution when using non-ambiguous grammars is given. This is crucial when a mathematical proof is required for the validation of procedures relying on graph data, filling an important gap in the field of random testing [8]. Similarly to the role of QuickCheck, a tool used to produce random strings for testing functions and procedures for the Haskell language [40], a random graph generator can be built for the purpose of graph based software testing, offering an unbiased inspection of the functionality with the utmost accuracy. For example, if we want to test the correctness or performance of a new program working with term graphs, we may want to provide a sufficient variety of samples. An insufficient number of test cases would not provide reliable information, but at the same time having a large number of graphs that includes too many duplicates could even give the misleading impression of the program operating correctly. Our method instead could efficiently provide a large enough number of samples that also guarantees their complete differentiation.

The original methods of Mairson are limited to non-ambiguous grammars if a uniform sampling distribution is sought. Instead we prove how by using graph representations of strings, in some cases it is possible to overcome these limitations, dramatically increasing the range of grammars to sample from. This represents an actual gap in the literature between grammars based on strings and their graph counterpart. We prove how the inherently-ambiguous prototype language presented in [25] has instead an equivalent non-ambiguous hyperedge replacement grammar and we also prove that there are still languages for which all representative grammars are ambiguous even in the setting of hypergraphs.

Considering the ambiguity problem, known to be undecidable for context-free string grammars [32], through the analysis of several cases, we introduce a novel notion of structural symmetry that helps solving the problem for hyperedge replacement grammars. Intuitively, we may think of a symmetry as a contextualized automorphism, a way to compare two hypergraphs that takes into account their surrounding elements. We prove how the structure of hypergraphs presenting such a feature can be directly related to the ambiguity of a grammar or even with the inherent ambiguity of a language. Moreover we show how this concept can be extended to prove the ambiguity of complex grammars.

Chapter 2

Background and Context

We begin giving the fundamental definitions needed to understand the generation process. The aim is to provide a solid background for the adaptation of Mairson's methods and the analysis of relevant cases. A running example, based on the language of term graphs, is presented to help with the comprehension of each step. For an exhaustive treatment of the theory of hyperedge replacement grammars and languages, please refer to the works of Drewes et al. [14], Habel [33] and Engelfriet [18], also see Courcelle [12] for a thorough analysis of context-freeness.

Analyzing the current literature, we notice that the concept of uniformity of a random sampling is often related to the operations underlying the stochastic process. It represents a rather static approach towards today's needs of strong mathematical and computational proofs. It is true, for example, that a pseudo-random number generator offers the most computationally unbiased choice of a number in a certain range, but, when considering more complicated processes, with several additional constraints, obtaining a uniform sampling may not be that straightforward. First of all, we have to identify what we want to apply the property of uniformity to. Then, we need to address the requirements of our sampling method. Ultimately we need to understand how the method manages to adapt to different scenarios.

A quick review of some of the known sampling methods is needed to help understanding the importance of our work:

- A very simple method to sample graphs is to start from a set of nodes and then apply a probability to each pair of them to be connected by an edge. Here the uniformity of the sampling is bound to the probability of each edge to be generated. If they all have the same probability we consider it uniform. The only requirements are the provision of a set of nodes as input and a random number generator to compute the probability for the generation of each edge. Beside the clear advantage of generating any kind of graph, there are several disadvantages such as the complete lack of control on the structure of the generated graph and the complexity of the method when considering hyperedges instead of simple undirected edges. Examples of this sampling logic can be found in [52] and [58]. We may also notice that these methods prevent the graph from growing dynamically during the generation, while this should be a common requirement for networks. These kinds of methods also rely on the identities of the nodes to discern among different results.

- Moving a step closer to our method we may consider a generation process such as the Gillespie algorithm [27] based on a Markov Chain Monte Carlo stochastic process. Originally proposed for the study of molecules reaction, this algorithm can also be applied to graph transformation rules. In this case the probability of a rule to be applied is dynamically calculated among all the possible applicable rules during the generation. This results in the advantage of having some sort of control to restrict the range of graphs and the evolution of the size of the graph during the process. On the other hand, this algorithm terminates when an arbitrary number of rules have been applied rather than reaching a point when we can consider a graph to be constructed. This also means that establishing a different goal, as for example the production of a particular subgraph, would not guarantee the termination of the algorithm. Also, it is not possible to decide which graph is the process converging to. It may indeed converge to a particular shape or diverge from others depending on the set of rules and how they are randomly applied. Ultimately, it is not trivial to calculate at each step how many rules, and for each of them how many times, are matched in the generating graph.

- A step further can be found in [50], where the probabilities are directly assigned to the productions of context-free hyperedge replacement grammars. That is, each production has a fixed probability to be chosen during the generation. If we assign the same probability to each of them, we may consider it a uniform sampling, but in terms of the productions rather than the hypergraphs themselves. Comparing this method to the previous one, we may notice that now we can only produce graphs in context-free hyperedge replacement languages. On the other hand it becomes easy to decide which rules are available. The probabilities are arbitrarily assigned before the sampling starts according to the limits given by a generating function to control the termination of the algorithm. This time the resulting graph is guaranteed to hold the properties derived by the rules of the grammar, but there is still no guarantee whatsoever of the uniformity of the distribution.

Our approach instead, lets a user specify an arbitrary domain through a hyperedge replacement grammar. Then, starting from a simple structure, it lets the graph grow up to the size chosen by the user, according to the rules specified in the grammar. This way the properties of the generated graph, the uniformity of the sampling are guaranteed and the termination of the generating algorithm are all guaranteed.

2.1 Hyperedge Replacement

Hypergraphs[4] (Fig. 2.1) are a generalisation of the familiar concept of graphs [3], in which edges, also called hyperedges, may simultaneously connect an arbitrary sequence of vertices. In our setting, we consider hyperedges to be both labelled and typed, meaning that each of them has a symbol and the number of its connected nodes is fixed. Hypergraphs can be easily adapted to represent relations in a wide variety of fields. We may think, for example, of using a hypergraph to represent a ternary operation among some a, b, c terms or the flow of an *if,then,else* statement or even the structure of a complex molecule where the hyperedges are the atoms and the nodes their bonds. In contrast to the classic representation of the Lewis structure [46], where the atoms are placed as nodes of a graph and the edges their bonds, the advantage of this model is to provide additional information on how each pair of atoms is mutually connected. It also allows us to describe chemical reactions through means of hyperedge replacements, instead of more complex node replacements.

Replacements represent the way a hypergraph evolves. With each replacement, one or more parts of a hypergraph are substituted, transforming the original graph into a different structure. The set of rules that can be applied to transform such a hypergraph define a replacement grammar. We may imagine a network that grows adding more and more servers and clients at each step according to some specified rules.

Given a set of rules, all the hypergraphs that can be generated by such a set define a language. Although hypergraphs may potentially represent any kind of relation, it is important noticing that the range of possible languages is limited by the type of rules we choose to use. We focus our work on the type of rules describing context-free[12] languages.

We briefly highlight some of the mathematical notions used in the fundamental definitions underlying this work.

$\mathbb{N}_0$: The set $\mathbb{N} \cup \{0\}$ of all natural numbers including 0 is denoted as $\mathbb{N}_0$.

Sequence: A *sequence*, denoted as $S = (a_1, \ldots, a_n)$, is an ordered set of n elements indexed from 1 to n. A sequence can contain the same element, indexed differently, more than once. We denote an empty sequence with the symbol λ. S^* denotes the set of all possible sequences over S.

Typing function: A *typing function* $f\colon A \to \mathbb{N}_0$ is a function associating a natural number to each element in A. $f(a)$ is referred to as the *type* of a. Such a function is used to define the number of nodes a hyperedge connects to.

Free symbolwise extension: Given a map $f\colon A \to B$ its *free symbolwise extension* $f^*\colon A^* \to B^*$ is defined as $f(a_1 \ldots a_n) = f(a_1) \ldots f(a_n)$. Intuitively we may consider it a mapping from a sequence to another. In case of the empty sequence, its extension $f^*(\lambda) = \lambda$.

(Discrete) Uniform probability distribution: Given a set S with $|S| = n$, where E_i represents the event for an element $e_i \in S$, with $1 \leq i \leq n$, to be chosen, a probability distribution is defined as *uniform* if each event E_i has the same probability $P(E_i) = 1/n$ to be observed.

To understand the structure of a hypergraph and its components, let's begin with its formal definition:

Definition 2.1.1 (Hypergraph)**.** Let *type*: $C \to \mathbb{N}_0$ be a typing function for a fixed set of labels C, then a *hypergraph* over C is a tuple $H = (V_H, E_H, att_H, lab_H, ext_H)$ where:

- V_H is a finite set of *vertices*
- E_H is a finite set of *hyperedges*
- $att_H\colon E_H \to V_H^*$ is a mapping assigning a sequence of *attachment nodes* to each $e \in E_H$
- $lab_H\colon E_H \to C$ is a function that maps each hyperedge to a *label* such that $type(lab_H(e)) = |att_H(e)|$
- $ext_H \in V_H^*$ is a sequence of pairwise distinct *external nodes*.

□

If the context is clear, the subscript H may be dropped from the tuple. The external nodes are only used as interface during the application of a replacement operation as explained later in Section 2.2.

When drawing hypergraphs we adopt the following conventions:

- Internal nodes are depicted as empty circles, along with a natural number, if needed, representing their identity.
- External nodes are depicted as full circles, along with a natural number representing their identity.
- Hyperedges are depicted as squares, with their label in the centre. Their identity and mark are represented next to them when needed.
- Non-terminal hyperedges are depicted with a white background. We use capital roman letters as non-terminal labels.
- Terminal hyperedges are depicted with a grey background, they may contain any terminal symbol.
- The connections of a hyperedge to its attachment nodes are depicted with continuous lines. The index of the node in the attachment sequence is represented by a smaller font natural number close to the line.

We denote e an m-hyperedge if $\mathit{type}(\mathit{lab}(e)) = m$. Also, $\mathit{type}(e) = m$ is used instead of $\mathit{type}(\mathit{lab}_H(e))$ if the context is clear. The ith vertex in a sequence of attachment nodes is denoted as $\mathit{att}_H(e)_i$ where $1 \leq i \leq \mathit{type}(e)$. For example, in Figure 2.1, the edge e_2 has $\mathit{att}(e_2) = (2, 5, 3, 2)$, it is a 4-hyperedge having $\mathit{att}(e_2)_1 = \mathit{att}(e_2)_4 = 2$. We may think of a conventional graph as a hypergraph in which every edge e has $\mathit{type}(e) = 2$, meaning that each edge connects exactly 2 vertices.

The mapping $\mathit{lab}(e)$ assigns a label $l \in C$ to each hyperedge $e \in E_H$. One of the fundamental functions of labels is to identify the subject of the replacement during the application of a production. Please note that edges with the same label also have the same type.

For a given $\mathit{type}\colon C \to \mathbb{N}_0$, the class of all hypergraphs over C is denoted by $\mathcal{H}_C$. The set $E_H^X = \{e \in E \mid \mathit{lab}_H(e) \in X\}$ denotes the subset of E_H with labels in $X \subseteq C$. As described later in this work, we distinguish between the subset E_H^N of non-terminal hyperedges with labels in $N \subseteq C$ and the subset E_H^Σ of terminal hyperedges, with labels in $\Sigma \subseteq C$.

We use the notation $\mathit{type}(H)$ for a length $|\mathit{ext}_H|$ and we call H an n-hypergraph if $\mathit{type}(H) = n$. The ith external node of H, with $1 \leq i \leq \mathit{type}(H)$, is denoted by $\mathit{ext}_{H,i}$. As explained later, hyperedges are only allowed to be replaced by hypergraphs of the same type.

If a n-hypergraph has exactly 1 hyperededge and all its nodes are external and connected to the hyperedge, that is $E_H = \{e\}$, $|V_H| = n$ and $\mathit{ext}_H \subseteq \mathit{att}(e)$, it is called the *handle* induced by e and denoted by I_e. Moreover if $\mathit{lab}(e) = A$, $\mathit{type}(e) = n$ and $\mathit{ext}_H = \mathit{att}(e)$ such a hypergraph is called the handle induced by A and denoted by $A^\bullet$. Handles are used as starting structures for the generation of the hypergraphs in the chosen domain.

In order to implement Mairson's methods, as thoroughly explained in section 3.2.2, we need to define a measure for the hypergraphs to be generated. Among all possible choices, to ease the transition from strings to graphs, we choose to define the *size* of a hypergraph as follows:

Definition 2.1.2 (Size of a Hypergraph)**.** Let $H = (V_H, E_H, \mathit{att}_H, \mathit{lab}_H, \mathit{ext}_H)$ be a hypergraph. We define $|H| = |V_H| + |E_H|$ as the *size* of H. □

We call H a size-n-hypergraph if $|H| = n$.

Figure 2.1 shows some examples of hypergraphs:

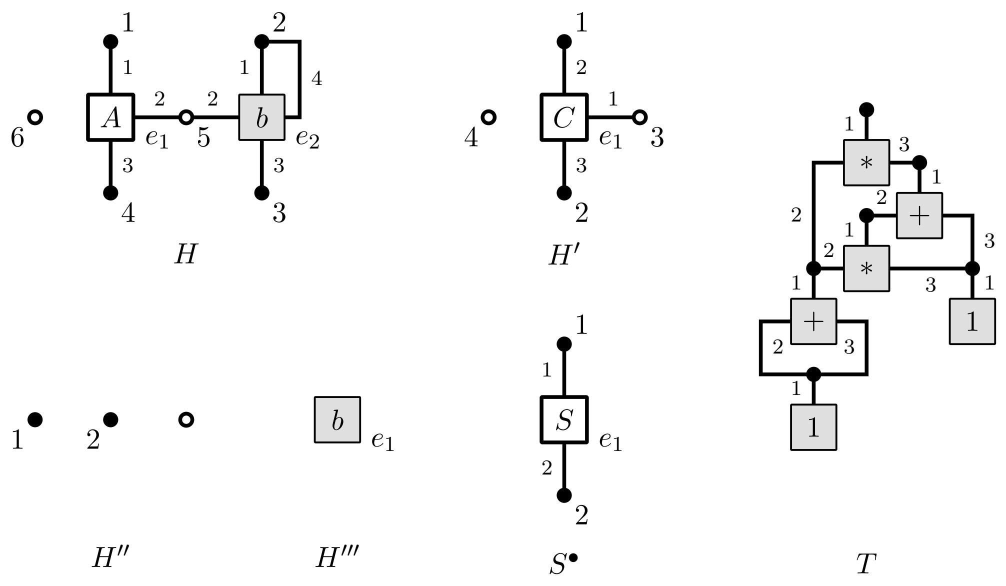


Figure 2.1: Examples of hypergraphs

- H: a size-8-hypergraph having $ext = (1, 2, 3, 4)$, $type(H) = |ext| = 4$, composed of a non-terminal hyperedge e_1, with $lab(e_1) = A$, $att(e_1) = (1, 5, 4)$, $type(e_1) = |att(e_1)| = 3$ and a terminal hyperedges e_2 with $lab(e_2) = b$, $att(e_2) = (2, 5, 3, 2)$, $type(e_2) = |att(e_2)| = 4$.
- H': a size-5-hypergraph having $ext = (1, 2)$, $type(H') = |ext| = 2$, composed of one non-terminal hyperedge e_1 with $lab(e_1) = C$, $att(e_1) = (3, 1, 2)$, $type(e_1) = |att(e_1)| = 3$.
- H'': a size-3-hypergraph having $E = \emptyset$, $ext = (1, 2)$, $type(H'') = |ext| = 2$.
- H''': a size-1-hypergraph having $V = \emptyset$, $ext = \lambda$, $type(H''') = |ext| = 0$ composed of one terminal hyperedge e_1 with $lab(e_1) = b$, $att(e_1) = \lambda$, $type(e_1) = |att(e_1)| = 0$.
- $S^\bullet$: a size-3-hypergraph having $ext = (1, 2)$, $type(S^\bullet) = |ext| = 2$. It is the handle induced by S composed of one non-terminal hyperedge e_1 with $lab(e_1) = S$, $att(e_1) = (1, 2)$, $type(e_1) = |att(e_1)| = 2$.
- T: a size-12-hypergraph composed of 6 terminal hyperedges and 6 nodes. It is an example of *Term* graph, a form of acyclic hypergraphs that represent functional expressions with possibly shared subexpressions. (See [51] for an introduction to the area of term graph rewriting.) The terminal hyperedges are labelled with the function symbols $*$, $+$ and the integer constant 1. The domain of term graphs will serve as the basis of one of the running example that will follow the description of the work.

We have chosen to show examples of planar hypergraphs only to simplify the reading of the images. We are reminded that each hyperedge in a hypergraph can potentially connect any number of nodes even more than once.

When sampling over a uniform distribution we need a way to decide if two generated elements are the same. While for strings we may simply compare their symbols from left to right, we rely on the notion of isomorphism to distinguish between equivalent hypergraphs. Such a notion takes into consideration the labels and the attachment nodes of the hyperedges, but ignores the identities of both of their edges and nodes.

Definition 2.1.3 (Isomorphism)**.** Two hypergraphs $H, H' \in \mathcal{H}_C$ are *isomorphic*, denoted $H \cong H'$, if there are bijective mappings $h_V \colon V_H \to V_{H'}$ and $h_E \colon E_H \to E_{H'}$ such that:

- $h^*_V(att_H(e)) = att_{H'}(h_E(e))$ and $lab_H(e) = lab_{H'}(h_E(e))$ for each $e \in E_H$
- $h^*_V(ext_H) = ext_{H'}$

$\square$

This definition is substantially different from the one usually adopted by classic methods in which a sampling is still considered uniform if two graphs have, for example, the same "shape", but differ by the identity of nodes. Our definition is stricter, so, when considering a uniform distribution, it is not possible to obtain two hypergraphs having the same shape, with their hyperedges having the same labels, but connected to the nodes in a different order. For example Figure 2.2 shows two isomorphic hypergraphs. Even if H and H' differ by the identity of their nodes and edges, a pair of mappings h_V, h_E can relate the components of H to the ones of H'. In this case: $h_E(e_1) = e_2$ and $h_E(e_2) = e_1$, so that $lab_H(e_1) = lab_{H'}(e_2) = A$ and $lab_H(e_2) = lab_{H'}(e_1) = b$. Moreover $h^*_H(1,2,3,4,5,6) = (3,1,6,2,4,5)$ so that $att_H(e_1) = (1,5,4)$ maps to the sequence of nodes $att_{H'}(e_2) = (3,4,2)$ and $att_H(e_2) = (2,5,3,2)$ maps to $att_{H'}(e_1) = (1,4,6,1)$. As for the external nodes: $h^*_H(1,2,3,4) = (3,1,6,2)$. The aim is to provide the generation of structurally different graphs rather than variations of identities of the same structure. If needed, once the graph is generated, node identities can be reassigned using a simple permutation algorithm. In software testing, for example, we want to provide a range of different shapes to test how a program responds towards a correctness or complexity analysis.

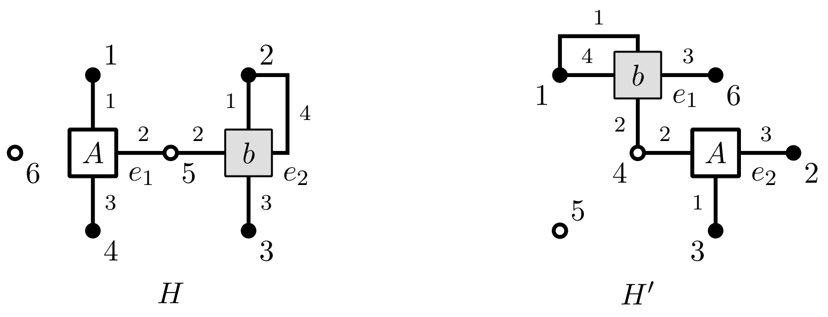


Figure 2.2: Isomorphic hypergraphs

H and H' are isomorphic and then considered as the same in the context of our generation method. We may also observe that the subgraph of H consisting of e_2 with nodes 2, 3 and 5 is isomorphic to the corresponding subgraph of H' formed by the edge equivalently labelled as b and its attachment nodes.

Accordingly, we give the following definition of subgraph:

Definition 2.1.4 (Subgraph)**.** A hypergraph H is a subgraph of H', denoted as $H \subseteq H'$, if there are bijective mappings $h_V \colon V'_H \subseteq V_H \to V_{H'}$ and $h_E \colon E'_H \subseteq E_{H'} \to E_{H'}$ such that: $h^*_V(att_H(e)) = att_{H'}(h_E(e))$ and $lab_H(e) = lab_{H'}(h_E(e))$ for each $e \in E'_H$. $\square$

Figure 2.3 shows the hypergraph H' as a subgraph of H. The mappings $h_E(e_3) = e_1$ and $h^*_V(1,2,5,6) = (1,3,2,4)$ are such that $lab_H(e_3) = lab_{H'}(e_1) = C$ and $att_H(e_3) = (2,1,5)$ maps to $att_{H'}(e_2) = (3,1,2)$.

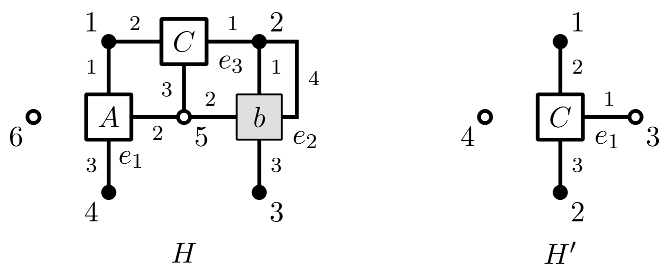


Figure 2.3: Subgraph

2.2 Replacement Grammars and Languages

In context-free string grammars the generation of a word is based on the local substitutions of non-terminals with sequences of symbols. In the context of hypergraphs instead, the equivalent operation is called *replacement*. To satisfy the dangling condition [34], a replacement can only happen between elements of the same type such as an n-hyperedge and an n-hypergraph, the rationale behind it being that all the connections of the replaced element should not be left "hanging" so that each attachment node of the edge must match the external node of its counterpart. Intuitively, considering the simple case of a n-hyperedge $e \in E_H$ and a n-hypergraph R, firstly e is removed from H, then R is added to H and finally the external nodes of R are merged with the attachment nodes of e. In the general case, a thorough definition of replacement takes into account the substitution of multiple elements at once:

Definition 2.2.1 (Replacement)**.** Let $H \in \mathcal{H}_C$ and $B \subseteq E_H$ and let *repl*: $B \to \mathcal{H}_C$ be a mapping with $\mathit{type}(\mathit{repl}(e)) = \mathit{type}(e)$ for each $e \in B$. Then the *replacement* of the hyperedges in B with respect to $\mathit{repl}(e)$ is defined by the operations:

1. Remove the subset B of hyperedges from E_H.
2. For each $e \in B$, disjointly add the vertices and the hyperedges of $\mathit{repl}(e)$.
3. For each $e \in B$ and $1 \le i \le \mathit{type}(e)$, fuse the ith external node $\mathit{ext}_{\mathit{repl}(e),i}$ with the ith attachment node $\mathit{att}_B(e)_i$.

□

We denote the resulting hypergraph by $H[e_1/R_1, \dots, e_n/R_n]$, where $B = \{e_1, \dots, e_n\}$ and $\mathit{repl}(e_i) = R_i$ for $1 \le i \le n$. If the context is clear, we may simply write $H[\mathit{repl}]$. The replacement preserves the external nodes, thus $\mathit{ext}_{H[\mathit{repl}]} = \mathit{ext}_H$.

The example in Figure 2.4 shows the replacement of the hyperedge $e_2 \in E_H$ with the hypergraph R:

1. The hyperegde $e_2 \in E_H$ is removed from H leaving behind a trace of the order of its attachment nodes.
2. The elements of R are disjointly added to H. The sequence of external nodes of R is positioned to match the order of the attachment nodes of e_2.
3. The external nodes of R are fused with the attachment nodes of e_2 generating the hypergraph $H' = H[e_2/R]$.

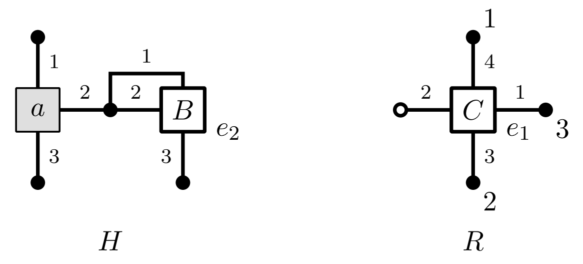
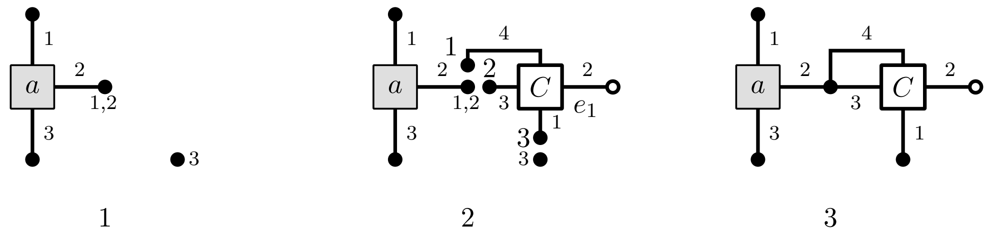



Figure 2.4: Replacement of the hyperedge e_2 with the hypergraph R

The consinstency of the result of a replacement, despite the order in which the hyperedges are substituted, is ensured by the confluence, sequentialization and parallelization properties of hyperedge replacement grammars described in [14]. It is fundamental to understand that due to these properties, two replacements that only differ in the order in which the elements are replaced will yield two isomorphic hypergraphs. Later, these properties will play a crucial role in the definition of ordered derivation trees. Since we distinguish between terminal and non-terminal hyperedges it is clear that only non-terminals can be the subject of a replacement.

The replacements applied during the generation of a hypergraph are defined in productions. Similarly to the string case, each production has the subject of the substitution, the hyperedge e indicated by the label *lab*(e), on its left hand side (*lhs*) and the object, the replacement hypergraph *repl*(e), on its right hand side (*rhs*).

Definition 2.2.2 (Production)**.** Let $N \subseteq C$ be a set of *non-terminals.* $p = (A, R)$ is a *production* over N, where *lhs*$(p) = A \in N$ is the label of the replaced hyperedge and *rhs*$(p) = R \in \mathcal{H}_C$ is a hypergraph with *type*(R) = *type*(A). $\square$

If $|ext_R| = |V_R|$ and $E_R = \emptyset$, then p is said to be *empty.* Clearly, the size of a hypergraph cannot be increased by the application of an empty production.

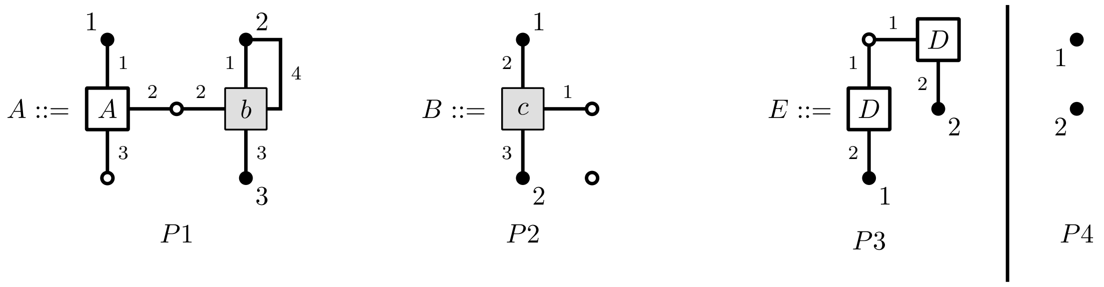


Figure 2.5: Example of productions

Figure 2.5 shows an example of productions over a set of labels $C = \{A, B, D, E, b, c\}$ with $N = \{A, B, D, E\}$:

$P1$: Represents a replacement of the hyperedge labelled as A with the hypergraph on its *rhs.* As we can clearly see, the type of the hypergraph matches the type of the hyperedge. Since the hypergraph contains a non-terminal hyperedge a further replacement is possible.

$P2$: Represents a replacement of the hyperedge labelled as B with a the hypergraph on its *rhs*. From the number of external nodes we can deduce that the hyperedges labelled as B are of *type* $= 2$. Since the hypergraph contains only a terminal hyperedge, no further replacements are possible.

$P3$: Is one of the possible replacement of the hypereges labelled as E. It is clearly a non-terminal production.

$P4$: Is the other possible replacement of the hypereges labelled as E. Since the *rhs* is composed by only external nodes, this is an empty production.

The application of the replacement specified in a production to a hypergraph, represents a direct derivation:

Definition 2.2.3 (Direct Derivation)**.** Let $H \in \mathcal{H}_C$ and let $p = (lab(e), R)$, with $e \in E_H$, then a *direct derivation* $H \Rightarrow_p H'$ is obtained by the replacement $H' = H[e/R]$. □

A sequence d of direct derivations $H_0 \Rightarrow_{p_1} \cdots \Rightarrow_{p_k} H_k$ of length k with $(p_1, \ldots, p_k) \in P$ is denoted as $H \Rightarrow^k H_k$ or $H \Rightarrow^*_P H_k$ if the length is not relevant. We denote it as $H \Rightarrow^* H_k$ if the sequence is clear from the context. A derivation $H \Rightarrow^* H'$ of length 0 is given if $H \cong H'$.

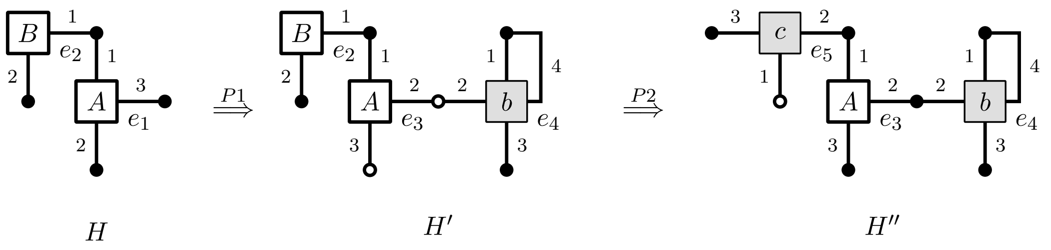


Figure 2.6: Example of derivation

Considering that each production represents a replacement, we may directly extend the confluence, sequentialization and parallelization properties to productions. As a consequence, distinct derivations, in which the same productions are applied in different order, yield isomorphic hypergraphs. Figure 2.6 shows a two steps derivation $H \Rightarrow_{P1} H' \Rightarrow_{P2} H''$ involving the production $P1$ and $P2$ from Figure 2.5. Given $P1 = (A, R_1)$ and $P2 = (B, R_2)$, we may as well express the transformation from H to H'' with the following equivalent replacements:

- $H'' = H[e_1/R_1, e_2/R_2]$, simultaneously replacing both hyperedges $e_1, e_2 \in H$.
- $H'' = H[e_1/R_1][e_2/R_2]$, considering the replacement of $e_1 \in H$ followed by the replacement of $e_2 \in H'$, since $H' = H[e_1/R_1]$. This is the replacement depicted in Figure 2.6.
- $H'' = H[e_2/R_2][e_1/R_1]$, as the previous case, but replacing e_2 before e_1.

So, to show that different derivations cannot produce isomorphic hypergraphs when considering a uniform distribution, we implement an ordering function to mark each of the hyperedges in the right hand side of a production. In [18] Engelfriet gives an induction based definition of derivation trees where the children of each node are arranged according to an arbitrary order. In our case, we exploit the order of hyperedges given by the marking to achieve a similar result. Such a marking in the *rhs* of each production, as shown in Figure 2.7, represents the order in which the replacements are carried out $(\alpha_1, \alpha_2, \dots, \alpha_{n-1}, \alpha_n)$. It is important to notice that the choice of the order does not affect the uniqueness of the derivation as long as it is kept consistent throughout the derivation. Given an ordered set $\{\alpha_1, \dots, \alpha_n\}$ where $a_i < a_j$ if $i < j \in \mathbb{N}$ we define a hyperedge replacement grammar as follows:

Definition 2.2.4 (Hyperedge Replacement Grammar). A *hyperedge replacement grammar*, or *HRG*, is a tuple $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ where:

- $N \subseteq C$ is a finite set of non-terminal labels
- $\Sigma \subseteq C$ is a finite set of terminal labels with $N \cap \Sigma = \emptyset$
- P is a finite set of productions
- $S \in N$ is the starting symbol
- $(mark_p)_{p \in P}$ is a family of functions $mark_p : E_R \to \{\alpha_1, \dots, \alpha_n\}$ assigning a mark to each hyperedge in the right-hand side of a production p. For each pair $e_i, e_j \in E_R$ with $i \neq j$, $mark(e_i) \neq mark(e_j)$

□

We denote as $P^A \subseteq P$ the subset of productions where $lhs(p) = A$. We call a production $p = (A, R) \in P_N$ *non-terminal* if $E_R^N \neq \emptyset$ or *terminal* if $p = (A, R) \in P_\Sigma$, where, accordingly, $P_N, P_\Sigma \subseteq P$, with $P_N \cap P_\Sigma = \emptyset$, are the disjoint subsets of non-terminal and terminal productions.

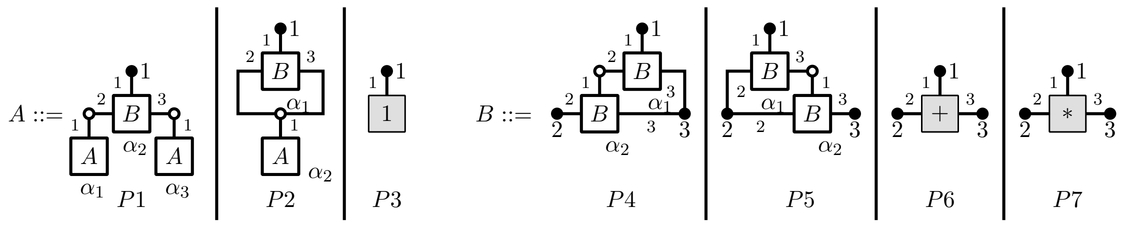


Figure 2.7: An ambiguous hyperedge replacement grammar for term graphs

The set of hypergraphs generated by a hyperedge replacement grammar G defines a *hyperedge replacement language* $L(G)$. As for the string case, different equivalent grammars may generate the same language. In drawings, we omit the marking on the terminal hyperedges, since they do not take part in further replacements.

Definition 2.2.5 (Hyperedge Replacement Language). The *hyperedge replacement language* (*HRL*) generated by a *HRG* $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ is the set $L(G) = \{H \in \mathcal{H}_\Sigma \mid S^\bullet \Rightarrow_P^* H\}$. We define for each $A \in N$, $L^A(G) = \{H \in \mathcal{H}_\Sigma \mid A^\bullet \Rightarrow_P^* H\}$. We also define for $n \in \mathbb{N}$, $L_n^A(G) = \{H \in \mathcal{H}_\Sigma \mid A^\bullet \Rightarrow_P^* H \wedge |H| = n\}$. Clearly $L_n^A(G) \subseteq L^A(G)$. □

Since we are interested in sampling from the slice of a language, that is, the subset of all members of a language having a specific size, it is important to notice that even if the set of hypergraphs generated by G is infinite, we only consider the subset $L_n(G)$ of all the size-n-hypergraphs in L. Such a subset, even if potentially very large is still finite. We denote as $|L_n^A|$ the size of the set of all size-n-hypergraphs in L that can be derived from $A^\bullet$.

Figure 2.8 shows 3 of the 3920 different size-12-hypergraphs of the slice of the language of term graphs L_{12}.

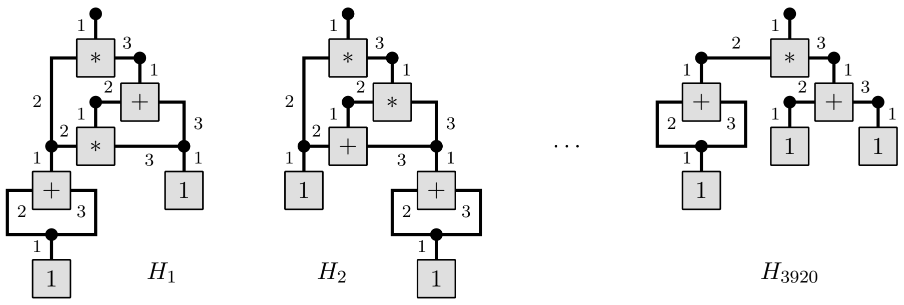


Figure 2.8: Some members of L_{12} term graphs

2.3 Ambiguity

In the setting of strings, a grammar G is considered ambiguous when there exists a word in $L(G)$ for which there are two different derivation trees or, equivalently, two different lefmost derivations. To translate both concepts to the settings of hypergraphs we rely on the ordering of the hyperedges presented in the previous section. Thus, given a set of productions P, we denote by T_P the set of all ordered trees over P which is inductively defined as follows:

1. for each $p \in P$, $p \in T_P$
2. for $t_1, \dots, t_n \in T_P$ and $p \in P_N$, $p(t_1, \dots, t_n) \in T_P$, for $(e_1, \dots, e_n)$ non-terminals in *rhs*(p).

Definition 2.3.1 (Ordered Derivation Tree)**.** Given a *HRG* $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$, an *ordered derivation tree* t for e such that $lab(e) = X \in N$, is a tree $p(t_{\alpha_1}, \dots, t_{\alpha_n})$ in T_P, such that $p = (X, R)$ is a production in P, and $t_{\alpha_1}, \dots, t_{\alpha_n}$ are derivation trees for $e_1 \dots e_n$, such that $X_1 \dots X_n$ are the labels of the non-terminal hyperedges in R marked with $\alpha_1 \dots \alpha_n$, respectively. □

The adjective *ordered* has been added to highlight the fact that its shape depends on the order of the marking chosen in the production of the grammar, since it is not possible to assign to a graph an implicit order as in the case of strings.

We define the *yield* of t, denoted with *yield*(t), as the hypergraph resulting from the sequence of replacements: $yield(p(t_{\alpha_1}, \dots, t_{\alpha_n})) = rhs(p)[e_1/yield(t_{\alpha_1}), \dots, e_n/yield(t_{\alpha_n})]$.

We need to ensure that different derivations yielding isomorphic hypergraphs are produced by the ambiguity of the grammar and not by an arbitrary order of the application of the productions. Otherwise, the simple assignment of a different marking during the derivation may produce different derivation trees. To obtain the sequence ω of pairs (e, p), where e is the hyperedge replaced in production p, from the pre-order traversal [53] of t, denoted as *trav*(t), we use the algorithm **Trav** (Alg. 1).

The reverse of ω corresponds to a recursive replacement sequence given in [18], but also identifies a unique ordering for the productions in the derivation.

There are $\prod_{p_N \in P} |E_R^N|!$ possible different ways of marking the productions of a grammar. Sampling a hypergraph using two identical grammars G, G' that only differ from their marking function generates two different ordered sequences ω, ω', but also yields two isomorphic results $H \cong H'$. For this reason, the marking of the *rhs* of the productions is defined as part of the grammar and is kept consistent throughout the derivation.

Intuitively, the ordering of the edges in the productions is somehow inherited by the elements of the generated hypergraph. To what extent it may help improving the problem of parsing, as presented in [49], may be a subject for further investigations. Nevertheless

```
Algorithm 1: Trav - Pre-order Traversal
  Given: a grammar $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ and an ordered derivation tree
    $t \in T_P$, where each node represents the application of a production $p \in P$ to a
    hyperedge $e$ in a sentential form in the derivation corresponding to $t$
  Input: an ordered derivation tree $t \in T_P$
  Output: $\omega$, a sequence of pairs $(e, p)$ corresponding to the pre-order traversal of $t$

  Let $\omega$ be an empty sequence. Recursively traverse the derivation tree $t$ and add the
    traversed nodes to $\omega$ starting from the root with $(\omega, getRoot(t))$
  function traverseTree (sequence $\omega$, node $n$):
    Append it to the sequence
    $appendNode(\omega, n)$;
    Continue the recursion for all the childs of $n$
    while $c \leftarrow getNextChild(n)$ do
      $traverseTree(\omega, c)$;
    end
```

we may already give a definition of leftmost derivation, equivalent to the conventional one used for strings. We remind that a derivation for string is called "leftmost" when, for each production of the derivation steps, the replaced non-terminal is the first to appear in the sentential form, ordering the symbols from left to right. In the context of hyperedge replacement grammars, we define as leftmost a derivation corresponding to the pre-order traversal of t:

Definition 2.3.2 (Leftmost Derivation)**.** Let $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ be a hyperedge replacement grammar and let t be an ordered derivation tree for a hypergraph H obtained from a derivation $d = S^\bullet \Rightarrow_P^* H$ and $trav(t)$ its pre-ordered visit. Then d is said to be a *leftmost derivation*, denoted as $lmd(H)$, if and only if the order of the applied productions of d corresponds to $trav(t)$. □

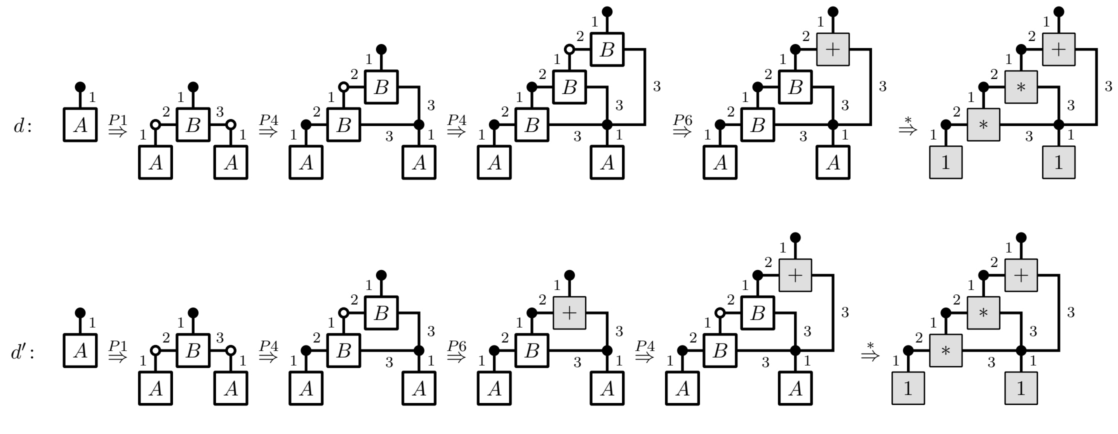


Figure 2.9: Pair of leftmost derivations showing the ambiguity of the term graphs grammar in Figure 2.7

The derivations d and d' in Figure 2.9 show the applications of the productions from the term graph grammar in Figure 2.7 yielding two size-10-hypergraphs H and H'. From $rhs(P4)$ in d we choose to apply again $P4$ to the edge marked as α_1 and then replace the result with terminal hyperedges, while in d' we first use $P6$ on the edge marked as α_1 and then $P4$ on α_2. Since the replacements has been made according to the ordering of the hyperedges, both derivations are considered as leftmost. The final replacements for the terminal hyperedges have been skipped for clarity. Figure 2.10 shows instead the ordered derivation trees t and t_1 with $yield(t) = H$ and $yield(t') = H'$. Moreover, considering their traversal, $trav(t) = lmd(H) = d$ and $trav(t') = lmd(H') = d'$. Since $H \cong H'$, it means that the same hypergraph can be generated applying either sequences of productions.

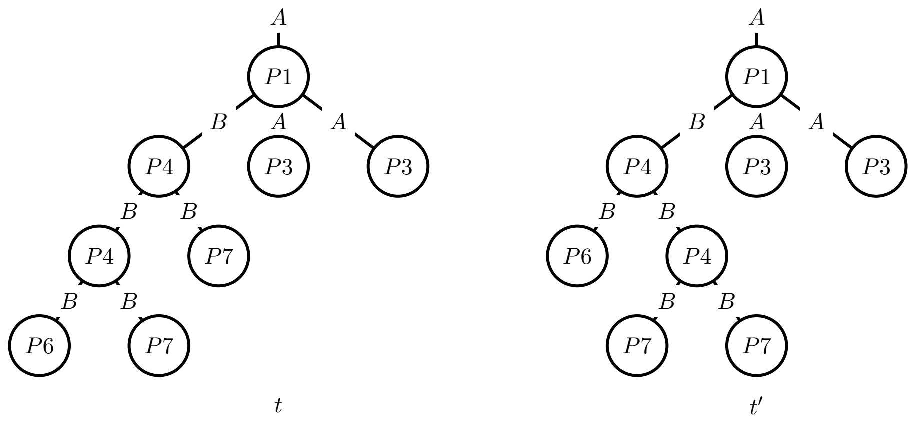


Figure 2.10: Pair of ordered trees showing the ambiguity of the therm graph grammar in Figure 2.7

Finally, using the notions of isomorphism, ordered derivation tree and leftmost derivation we give the following definition of an ambiguous grammar:

Definition 2.3.3 (Ambiguous Grammar)**.** A *HRG* $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ is *ambiguous* if there are ordered derivation trees $t_1, t_2 \in T_P$, such that $t_1 \neq t_2$ and $yield(t_1) \cong yield(t_2)$. If $yield(t_1), yield(t_2) \in L_n(G)$ we say that G is *n-ambiguous.* □

Corollary 2.3.1. *A hyperedge replacement grammar* $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ *is ambiguous if and only if there exist* $H, H' \in L(G)$ *such that* $H \cong H'$ *and* $lmd(H) \neq lmd(H')$*.*

Proof. Let $t \neq t'$ be ordered derivation trees with isomorphic yieldings $yield(t) \cong yield(t')$. Let $lmd(yield(t)) = trav(t)$ and $lmd(yield(t')) = trav(t')$ be leftmost derivations obtained by their pre-order traversals. Then, $lmd(yield(t)) \neq lmd(yield(t'))$ are distinct derivations yielding isomorphic results. □

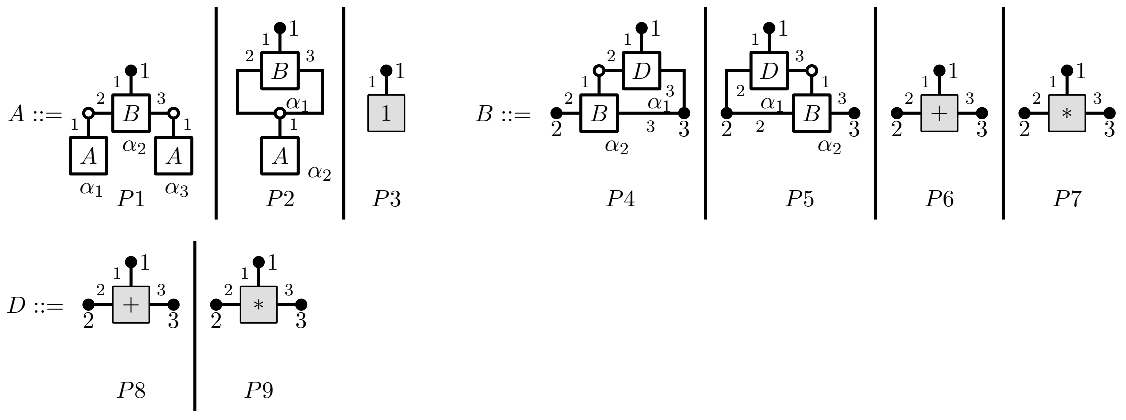


Figure 2.11: A non-ambiguous version of the hyperedge replacement grammar for term graphs

Figure 2.11 shows a non-ambiguous version of the term graph grammar previously presented in Figure 2.7. The ambiguity derived by the productions $P4$ and $P5$, featuring two B labelled hyperedges, has been resolved introducing a new non-terminal D and two new productions $P8$ and $P9$.

Moreover we can observe that a derivation $S^\bullet \Rightarrow_P^* H \Rightarrow_P^* H'$ is a unique leftmost derivation if and only if both $S^\bullet \Rightarrow_P^* H$ and $H \Rightarrow_P^* H'$ are unique leftmost derivations.

Since we are generating size-n-hypergraphs in $L_n(G)$, a particular attention should be given to the fact that even if a grammar is ambiguous, its restriction to $L_n(G)$ can still be non-ambiguous. that is, considering the size of the hypergraphs, the sampling can still be achieved uniformly at random.

On the other hand, since an n-ambiguous grammar is also generally ambiguous, our method may also be used to explore and detect the ambiguity of grammars.

Knowing that the ambiguity problem is undecidable for strings, we may wonder if the same applies to graphs. The problem can be formulated as follows: Is there an algorithm that is able to decide, in general, if a context-free *HRG* is ambiguous?

Theorem 2.3.1. *The ambiguity problem for hyperedge replacement grammars is undecidable in general.*

Proof. Let *HA* be the ambiguity problem for hyperedge replacement grammars defined as follows:

- **Input:** A context-free hyperedge replacement grammar G.
- **Question:** Is G ambiguous?

Let *SA* be the corresponding ambiguity problem for context-free string grammars. We have the following facts:

- It has been proven in [32] that *SA* is undecidable in general.
- For every context-free string grammar G_s there is an *HRG* G_H such that $L(G_H)$ consists of all string graphs representing strings in $L(G_s)$, see Section 5.2.

Then Theorem 2.3.1 is proven by contradiction based on the following reduction from *SA* to *HA*:

Let E_{TM} be an encoder from a string grammar S into an equivalent hyperedge replacement grammar H as shown in Proposition 5.1.1. Let HA_{TM} be a decider for a hyperedge replacement grammar that on input H returns *true* if H is ambiguous or *false* otherwise. We then build a Turing machine SA_{TM} that on input S, encodes S into H, runs HA_{TM} on H and finally outputs *true* if the output of HA_{TM} is *true* or *false* otherwise. SA_{TM} would then be a decider for *SA*, but, since *SA* is undecidable, then HA_{TM} cannot exists and consequently *HA* must also be undecidable. □

One more feature we have to take into account when considering the ambiguity of an *HRG* is the structure of the graphs involved in the generation. While strings are an ordered sequence of symbols for which the ambiguity of a grammar is based on the arrangement of such symbols on a line, graphs allows for a wide variety of different shapes to base the arrangements on. It means that the ambiguity of an *HRG* may be directly caused by the shape of the graphs [22] involved in the derivation. For example, the regular string language $L = \{w \in \{a^*\}\}$, composed by linear sequences of a single symbol, can be represented by the simple non-ambiguous context-free grammar $S \longrightarrow aS|\lambda$. Instead a graph language representing trees where every edge has the same symbol is proven to be inherently ambiguous. Identifying the basic structures that lead to the ambiguity of grammar is important to help solving the ambiguity problem for more complex languages.

So, along with well-known methods already used for strings, we introduce the concept of *structural symmetries*.

First of all we define the set of hypergraphs that can be obtained by the replacement applied to a particular hyperedge while keeping the rest of the structure intact. These hypergraphs are not necessarily terminal. Such a set is defined as follows:

Definition 2.3.4 (Local Generating Power)**.** Let $G = (N, \Sigma, P, S, (mark_p)_{p\in P})$ and let H be a hypergraph such that $e \in E_H$, then the *local generating power* of e is the set Π^e_H defined as:

for each R so that $I_e \Longrightarrow^*_P R$ then $H[e/R] \in \Pi^e_H$. □

That is, the set of hypergraphs based on H and having subgraphs obtained by all the possible derivations from I_e. As usual the set Π^e_H can be finite or infinite according to the productions in P. Also Note that if $lab(e) = A$, the set of subgraphs derived from I_e corresponds to L^A if and only if $I_e = A^\bullet$.

Considering two distinct hyperedges e, e' we compare their generating powers according to the following definition:

Definition 2.3.5 (Hyperedge Symmetry)**.** Two hyperedges $e, e' \in H$ are defined as symmetric, denoted as $e \approx e'$, if and only if $\Pi^e_H = \Pi^{e'}_H$. We define them as partially symmetric, denoted as $e \simeq e'$, if $\Pi^e_H \cap \Pi^{e'}_H \neq \emptyset$. □

We call a symmetry *non-finite* if Π^e_H and $\Pi^{e'}_H$ are infinite, finite otherwise. We also refer to a hyperedge symmetry in the structure of a hypergraph as a structural symmetry. Moreover, if the presence of a symmetry causes the ambiguity of a grammar, we refer to it as a structural ambiguity.

Symmetries can be found in productions, sentential forms or the members of a language themselves. Figure 2.12 shows some examples of hypergraphs potentially leading to a symmetry during a derivation. We must be careful with extending this concept directly to the *rhs* of productions, since they may be applied to a specific part of a hypegraph preventing the creation of a symmetry. Using the word *potentially* we want to stress that it is important to remember that, all the hyperedges in a hypergraph must be considered when searching for a symmetry. Also, differently from automorphism, we consider a comparison on the whole sets of graphs that can be generated by the involved hyperedges.

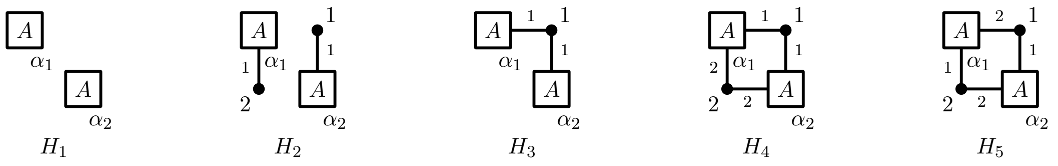


Figure 2.12: Examples of structures, potentially generating a symmetry.

The hypergraphs in Figure 2.13 do not directly represent symmetries. Even if both hyperedges labelled with A have an equal generative power, the hyperedge labelled with B breaks the symmetry. For example there could be a production for S_1 replacing the hyperedge marked as α_3 with a hypergraph featuring a B labelled hyperedge in the opposite direction or in S_2 the hyperedge marked as α_1 may generate another 1-hyperedge labelled as B and attached to the same node. This is outside the scope of our definition, since the ambiguity of a grammar can indeed manifest itself through different sequences of productions. This means that while the lack of symmetries is not sufficient to claim that a grammar is non-ambiguous, it still stands that the presence of symmetries is sufficient to claim that a grammar is ambiguous.

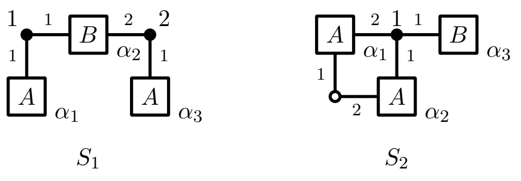


Figure 2.13: Example of structures that do not directly generate a symmetry.

We then strictly consider the study of the symmetries limited primarily to the comparison of the set of graphs generated by the symmetric hyperedges e and e', leaving the extension of the cases involving the neighbor hyperedges to further studies.

Lemma 2.3.1. *Given a grammar $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$. If there is a derivation $S^\bullet \Rightarrow_P^* H \Rightarrow_P^* H'$ and $e, e' \in E_H$ such that $e \simeq e'$, then G is ambiguous.*

Proof. Let's consider a sentential form F in a derivation $S^\bullet \Rightarrow_P^* F \Rightarrow_P^* H$ with $H \in L(G)$ and two distinct hyperedges $e, e' \in E_F$. Consequently there exist two different derivations containing the steps $F \Rightarrow_P^* F' \Rightarrow_P^* H$ and $F \Rightarrow_P^* F'' \Rightarrow_P^* H'$ with $F' \cong F''$. Since the application of the productions is strictly ordered the productions applied to e and e' belong to two different paths of a derivation tree. Suppose $e \simeq e'$, then there exist two different derivation trees t and t' with *yield*$(t) \cong$ *yield*(t') obtained by switching the paths of e and e'. By definition 2.3.3, such grammar is ambiguous. $\square$

By extending the concept of symmetries directly to the components of the members of a language, we conjecture that in particular cases it is possible to infer the inherent ambiguity of the language directly from the structure of its members. Examples of such languages include:

- The language of unbounded points stars.
- The language of multiply labelled cycles.
- The language of singly labelled binary trees.
- The language of series-parallel graphs.

Theorem 2.3.2. *A HRL having non-finite structural symmetries in its members is inherently ambiguous.*

Please refer to Chapter 4 for a thorough analysis of these cases and the proof of the theorem.

2.4 Chomsky Normal Form

The original Mairson's methods work on string grammars in Chomsky normal form. The productions of this particular form can only be in one of the following formats:

- $A \longrightarrow BC$ where B and C are non-terminal symbols.
- $A \longrightarrow a$ where a is a terminal symbol.
- $S \longrightarrow \epsilon$ where S is the starting symbol and ϵ is the empty string, but in this case, S must not appear in the *rhs* of any other production.

In Section 3.2 of [18] Engelfriet introduces a method to convert hyperedge replacement grammars in a normal form that has at most two hyperedges in the *rhs* of each of their productions. Also he shows a method to remove empty productions. In order to work with Mairosn's methods we need a slightly different form closer to the classic one for strings, that restricts non-terminal productions to have exactly two non-terminal hyperedges and non-empty terminal productions to have exactly one terminal hyperedge or just internal nodes in their right hand side.

Definition 2.4.1 (Chomsky Normal Form)**.** We define a *Chomsky normal form* (*CNF*) for hyperedge replacement grammars as a tuple $G_{CNF} = (N, \Sigma, P, S, (mark_p)_{p\in P})$ where:

- $N \subseteq C$ is a finite set of non-terminal labels
- $\Sigma \subseteq C$ is a finite set of terminal labels with $N \cap \Sigma = \emptyset$
- P is a finite set of productions
- $S \in N$ is the starting symbol
- $(mark_p)_{p\in P}$ is a family of functions $mark_p : E_R \to \{\alpha, \beta\}$ assigning a mark to each hyperedge in the right-hand side of a production p

Each production $p = (A, R) \in P$ satisfies one of the following constraints:

- $E_R = \{e_1, e_2\}$ where $lab(e_1), lab(e_2) \in N$ and $mark(e_1) \neq mark(e_2)$, in which case the replacement is firstly carried out on the hyperedge marked with α, then on the one marked with β
- $E_R = \{e_1\}$ where $lab(e_1) \in \Sigma$ and $mark(e_1) = \alpha$
- $E_R = \emptyset$, $|V_R| > |ext_R|$
- $A = S$, p is the empty production and for each $q \in P$, for each $e \in rhs(q)$, $lab(e) \neq S$

□

Note that in the first two cases, *rhs*(p) contains either exactly two non-terminal hyperedges or a single terminal hyperedge and may also contain isolated nodes. Productions according to the third case are considered as *terminal* productions. The last case specifies that the empty production is only allowed if there is no other production having the starting symbol in its right-hand side.

Lemma 2.4.1. *There exist an algorithm that for every hyperedge replacement grammar G produces a grammar G' in Chomsky normal form such that $L(G) = L(G')$.*

Proof. We present a set of rules to transform any grammar G, into an equivalent grammar G' such that, for each direct derivation $H \Rightarrow_p H'$ with $p \in P_G$, it exists an equivalent derivation $H \Rightarrow^*_Q H'$ with $Q \subseteq P_{G'}$. The proof is provided with a running example showing the application of the rules. The grammar in Figure 2.14 contains productions that are not in Chomsky normal form: $P1$ has more than 2 hyperedges; $P2$ has a single non-terminal hyperedge; $P3$ is an empty production, but its *lhs* is not S; $P4$ has 2 hyperedges one of which is terminal.

For a production $p = (A, R) \in P$, that is not already in *CNF*, we consider the following set of rules, applied in this order, to obtain a corresponding equivalent set of productions P' in *CNF*:

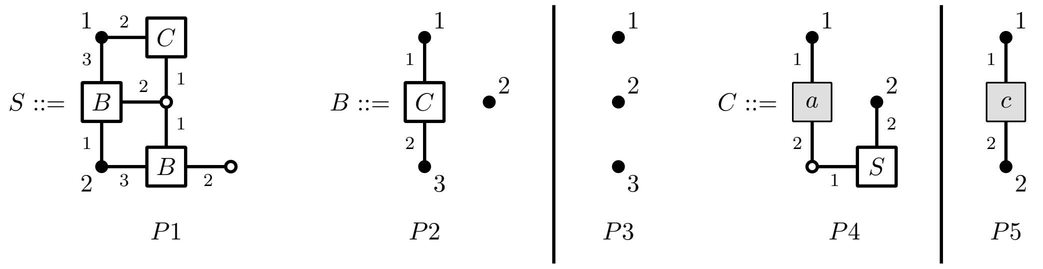


Figure 2.14: Example grammar for the proof of *CNF* equivalence

1. If p is the empty production, for each production $q = (B, X) \in P$ having $e \in E_X$ with $lab(e) = A$ in its *rhs*, for each production $q' = (A, Y) \in P$ having A in its *lhs* we apply the substitution $R' = X[e, Y]$ and add the productions $p = (B, R')$. We then remove the productions that are no longer needed. The proof of equivalence of the derivations $H \Rightarrow_q H' \Rightarrow_{q'} H''$ and $H \Rightarrow_{p'} H''$ is the following: if e' with $lab(e') = B$ is the hyperedge involved in the derivation $H \Rightarrow_q H'$ then $H'' = H[e'/X[e/Y]] = H[e'/R']$ since $R' = X[e, Y]$.

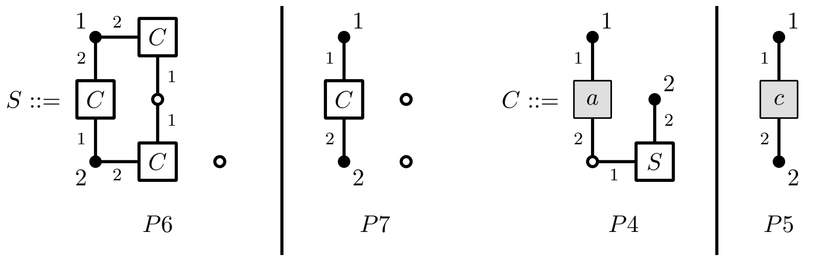


Figure 2.15: Removal of the empty production $P3$

In order to remove the empty production $P3$ (Fig. 2.15) we apply the replacements of all the productions having B as their *lhs* to all the productions having a hyperedge labelled as B in their *rhs*. We remove $P1$ and introduce the productions $P6$ and $P7$. We then remove $P2$ and $P3$ since they are not longer needed.

2. If $E_R = \{e'\}$ with $lab(e') \in N$ for each production $q = (lab(e'), X) \in P$ we add the production $p' = (lab(e), R')$ with $R' = R[e'/X]$. If $E_{R'} = \{e''\}$ with $lab(e'') \in N$ this step is iterated and terminates when $|E_{R'}| > 1$ or $E_{R'} = \{e_t\}$ with $lab(e_t) \in \Sigma$ or $|E_{R'}| = 0$ and $|V_{R'}| > ext_{R'}$. The proof of equivalence of the derivations $H \Rightarrow_p H' \Rightarrow_q H''$ and $H \Rightarrow_{p'} H''$ is the following: if e' is the hyperedge involved in the derivation $H' \Rightarrow_q H''$ then $H'' = H'[e/R[e'/X]] = H[e/R']$ since $R' = R[e'/X]$.

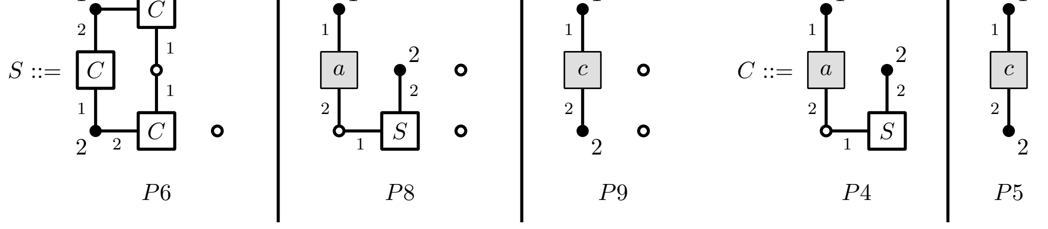


Figure 2.16: Removal of production $P7$

Since $P7$ has a single non-terminal hyperedge C (Fig. 2.16), we apply a replacement for each production that has C on its *lhs*. In our case, using the replacements of $P4$ and $P5$, we obtain $P8$ and $P9$. The production $P7$ is removed from the grammar.

3. If $|E_R| = k > 2$ we consider the subgraph X of R composed by the subset $E_X \subset E_R$ of hyperedges $e_2, \ldots, e_k$ and their attachment nodes. We introduce a new label T so that $N' = N \cup \{T\}$ and a new handle $T^\bullet$ of e_T with $ext(e_T) = \bigcup\limits_{2 \leq i \leq k} att(e_i)$ such that $type(e_T) = type(X)$. We then consider the hypergraph R' composed by $R \backslash X$ and $T^\bullet$ where $V_{R'} = V_{R \backslash X} \cup V_{T^\bullet}$ and $E_{R'} = E_{R \backslash X} \cup E_{T^\bullet}$. Finally we add the productions $p' = (A, R'), p'' = (T, X)$ to P'. If $|E_X| > 2$ this step is iterated. The proof of equivalence of the derivations $H \Rightarrow_p H'$ and $H \Rightarrow^*_{P'} H'$ is the following: if e_a is the handle of the *lhs* of p we consider the following equivalence of the replacements then $H' = H[e_a/R] = H[e_a/R'[e_T/X]]$ since $R = R'[e_T/X]$.

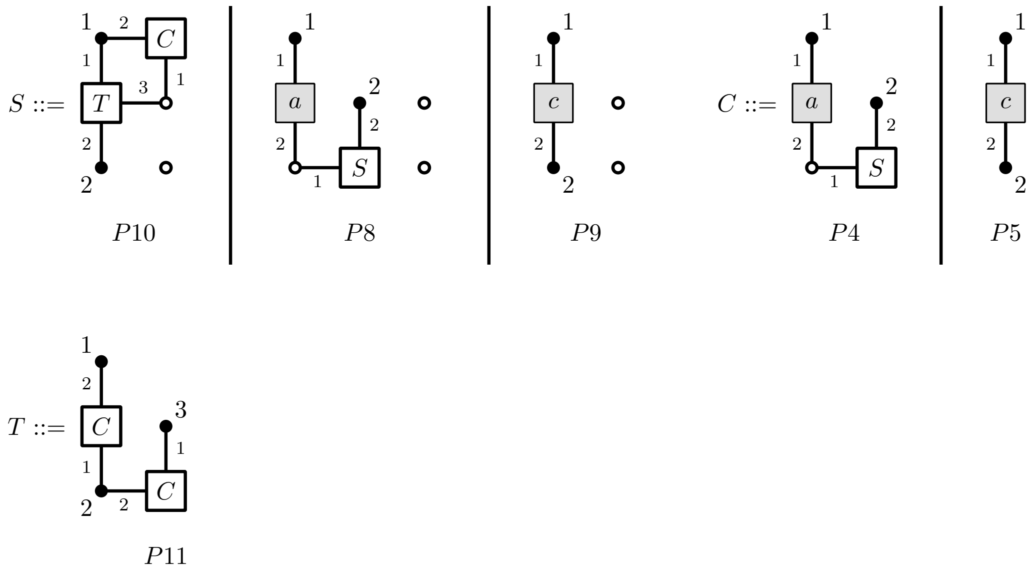


Figure 2.17: Removal of production $P6$

Since production $P6$ has three non-terminal hyperedges (Fig. 2.17), we create a new label T, a new handle $T^\bullet$ and the production $P11$. Then we add the production $P10$ so that the replacement of the hyperedge labelled as T by the *rhs* of $P11$ results in the *rhs* of $P6$. The production $P6$ is then removed from the grammar.

4. If $|E_R| > 1$ and exists $e' \in E_R$ such that $lab(e') \in \Sigma$ a new label T is introduced so that $N' = N \cup \{T\}$. We add 2 new productions $p' = (A, R')$ to P' where $R' = R$ with $lab(e') = T$ and $p'' = (T, e'^\bullet)$. This step is repeated for each $e' \in E_R$ with $lab(e') \in \Sigma$. Due to the confluency property of *HRG*s the order in which the terminal hyperedges are chosen is irrelevant. The proof of equivalence of the derivations $H \Rightarrow_p H''$ and $H \Rightarrow_{p'} H' \Rightarrow_{p''} H''$ is the following: if $e' \in E_R$ with $lab(e') \in \Sigma$ is the hyperedge involved in the derivation $H \Rightarrow_p H''$ then $H' = H[e/R] = H[e/R'[e'/e'^\bullet]]$ since $R = R'[e'/e'^\bullet]$.

Both *rhs* of productions $P4$ and $P8$ are composed by a terminal and a non-terminal hyperedge. We introduce a new label A and a its handle $A^\bullet$ along with the production $P14$ (Fig. 2.18). We then add the productions $P12$ and $P13$ resulting from the substitution of the terminal hyperedges labelled with a by the non-terminal hyperedges labelled with A. Productions $P4$ and $P8$ are then removed from the grammar.

$\square$

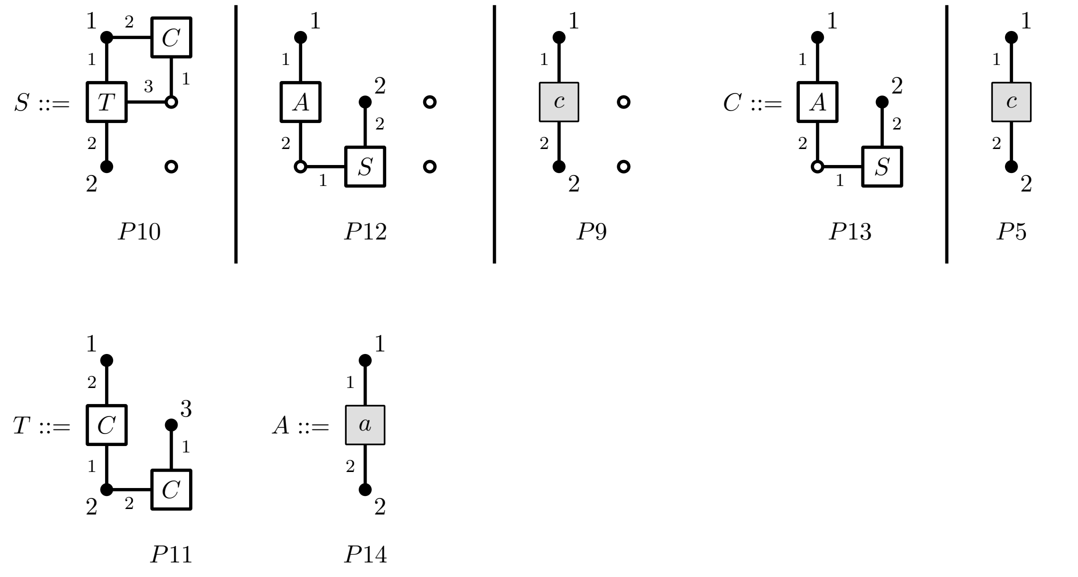


Figure 2.18: Removal of production $P8$ and $P4$

From this point on, if not explicitly specified, we always refer to an *HRG* as an *HRG* in *CNF*. We stress that the input of the method must be already provided in this form, that is, the time required for the transformation is not taken into account during the evaluation of the time complexity.

The grammar in Figure 2.19 is the Chomsky normal form of the term graph grammar in Figure 2.11.

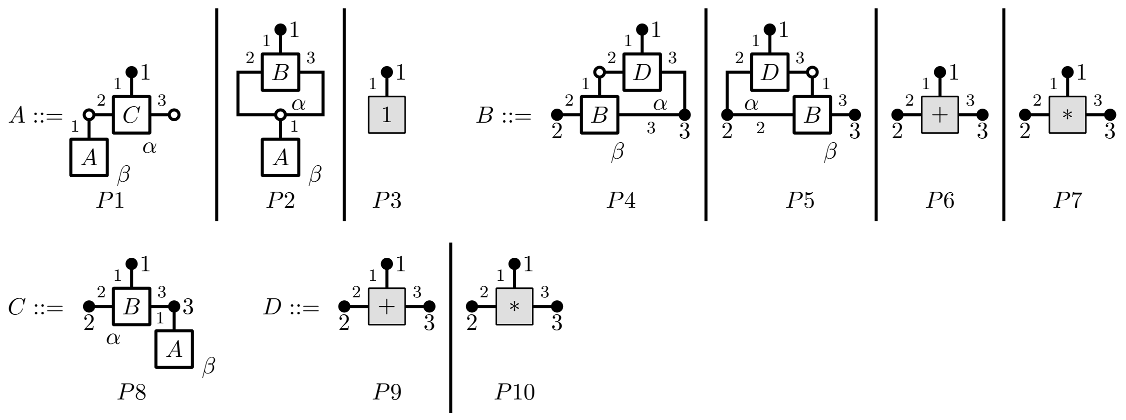


Figure 2.19: G_{tg}, Chomsky normal form of the term graphs grammar in Figure 2.11

In order to complete the adaptation of the grammar we propose a more suitable shorthand representation of the productions that only extracts the necessary information. For calculating the probabilities of the productions to be chosen, we can safely discard the topological data of the replacements. We are only interested in which hyperedge is replaced, specified by its label, the labels of the hyperedges that are generated and the number of additional nodes that are created, if any. We can safely discard any other information and keep just a pointer to the original production, to use during the replacement process. For each $p = (A, R) \in P$ and $i \in \mathbb{N}_0$ we use the following notations:

- $A \xrightarrow{p} BC, i$ for a *non-terminal* production where $B, C \in N$ are the labels of the marked hyperedges $e_\alpha, e_\beta \in R$ with $mark(e_\alpha) = \alpha$, $mark(e_\beta) = \beta$ and $i = |V_R \backslash ext_R|$.

- $A \xrightarrow{p} a, i$ for a *terminal* production where $a \in \Sigma$ is the label of the marked hyperedge $e_\alpha \in R$ and $i = |V_R \backslash ext_R|$.
- $A \xrightarrow{p} \lambda, i$ for a *terminal* production where $E_R = \emptyset$ and $i = |V_R \backslash ext_R|$.

It is important to notice that if the original grammar is non-ambiguous this method runs in polynomial time and results in a polynomial space expansion of the grammar. In case of an ambiguous grammar, let's consider the one in Figure 2.20. The first step of the derivation from S creates the hypergraph on the *rhs* of $P1$. From that point any combination of any number of $P2$ and $P3$ result in one of the possible combination of the ordering of the hyperedges a, b and c, terminated by the application of $P4$.

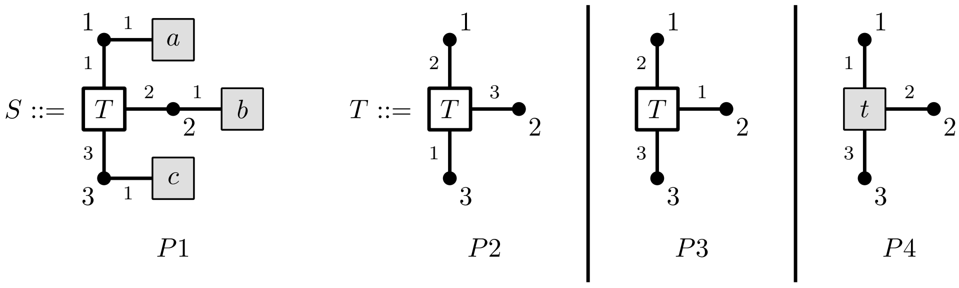


Figure 2.20: Shuffler grammar

Applying the rules for the transformation in Chomsky normal form, we have to take into account all the possible permutations given by $P2$ and $P3$. Since *type*$(T) = 3$ the new productions generated are *type*$(T)! = 6$. We should also notice that if a grammar presents such kind of productions, it is also ambiguous. There are indeed infinite ways of combining $P2$ and $P3$ to obtain the same hypergraph.

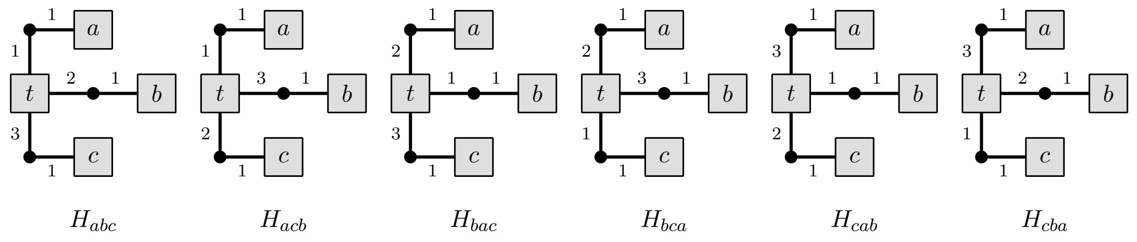


Figure 2.21: The 6 different combinations deriving from the grammar in Figure 2.20

Finally, we extend the concept of non-contracting, or monotonic, to the settings of hyperedge replacement grammars. This concept has been introduced by Chomsky in [9] to describe the behavior of productions in context sensitive grammars, also defined as *Type 1*. It states that for a derivation $\alpha N \beta \implies \alpha w \beta$ the length of a sentential form is always greater or equal than the length of the previous one. Moreover, a string grammar is defined as *essentially* non-contracting if there is a production from the starting symbol to the empty string $S ::= \epsilon$.

For example, if we want to generate a word of length 4 using the following string grammar:

$S ::= AB \mid \epsilon$

$A ::= a$

$B ::= BA \mid b$

we may notice that the length of the sentential forms does not decrease throughout the derivation:

$$S \implies AB \implies aB \implies aBA \implies aBAA \implies abAA \implies abaA \implies abaa$$

Equivalently, we define a hyperedge replacement grammar as non-contracting if the size of the sentential form does not decrease during the derivation. The only difference with the strings counterpart is that there isn't an upper bound for the growth of the generated hypergraph during each step. Each production may indeed increase the size of the subsequent sentential form by more than 1. As in the string case, we consider a hyperedge replacement grammar to be essentially non-contracting if there is a production from the starting symbol to the empty graph. Formally:

Definition 2.4.2 (Non-contracting *HRG*)**.** A *HRG* $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ is defined as *non-contracting* if for each derivation $H \Rightarrow_p H'$, $|H| \leq |H'|$. Also, it is defined as *essentially non-contracting* if there exists $p = (S, R) \in P$ such that p is the empty production. $\square$

Chapter 3

Sampling in Hyperedge Replacement Languages

A naive way to sample a graph is to enumerate a set of graphs in advance and then pick one at random. Even if it may seem, trivially, the best way to solve the problem achieving a uniform distribution, the very difficult task of building such a set still remains. It may indeed be a very large set, completely beyond our computational power as shown for the series-parallel graphs in [38] and we should also take care of carefully enumerating all the graphs to avoid counting the same element twice. We may think about the difference between adding a graph to the set and adding all the correct ones as solving a single instance of a problem and counting all the possible solutions for such an instance.

In 1979 L. G. Valiant showed that, despite the decision problem for the perfect matching can be solved in polynomial time, calculating all the possible matching for the same instance of the problem belongs to a different class, which he presented as *#P-complete* [55]. Comparing it to our settings, it means that the aforementioned naive solution would require an initial calculation of all possible graphs in the set, before starting the sampling process. Formally, given a function f, $\#P$ is the class of function problems of the form "compute $f(x)$", f equals the number of accepting paths of a polynomial-time nondeterministic Turing machine. A function problem F is classified as *#P-hard* if there exists a polynomial reduction for every problem in $\#P$ to F. So, for all the problems F in the *#P-complete* class it holds that:

- F is in $\#P$.
- F is *#P-hard*, that is, there exists a polynomial reduction to every other problem in *#P-complete.*

The canonical problem for the *#P-complete* class is $\#SAT$. Its $f(x)$ consists in finding all the satisfying assignments of a boolean expression x and is the counterpart of the common SAT problem for the *NP-complete* class.

So, for our naive solution, the definition of *#P-completeness* suggests that this approach, although being statistically correct, is infeasible even for relatively small instances of the problem [56]. Nevertheless, it has been proved in [41] by Jerrum et al. that there exists a complexity gap between counting and sampling. There are problems for which counting all the admissible solutions and generating a single possible solution uniformly at random fall into two different complexity classes. It means that there exist cases in which we do not need to generate and enumerate all the members of a set in order to produce a uniform sampling.

We may think about a graph generation process as a set of rules for constructing a structure composed of vertices and edges. It is important to notice that the properties of the generated graph may or may not be directly dependent on the procedure used to assemble such a structure. Usually, if one of more of these rules are part of a stochastic process, we say that a graph is generated at random.

For example, the following is a classic procedure for the generation of a graph with n vertices uniformly at random:

1. Start from a complete graph $G = (V, E)$ with $|V| = n$
2. For each pair $v_i, v_j \in V$ with $i > j$ remove the edge $e_{i,j} \in E$ with probability $1/2$

We should notice that in this particular case we take into consideration the identities of the nodes. That is, two graphs with the same shape are still considered distinct towards the uniformity of the sampling if their nodes have different identities. Although it may seem a good starting point, among the limitations of this method there is a particularly noticeable one: is there a way to generate a graph where a certain property holds? Of course we may think of another ad-hoc method to sample a graph with our sought property, but this would require to study a new algorithm, prove its complexity and distribution and test its limits. There exist several methods already to generate graphs with peculiar properties, but the aim of this work is to create one where only the properties should be chosen, if available, without changing the method itself. There could be properties harder to test, requiring extensive resources, thus the testing could not always be considered a feasible solution, for example consider the use of this classic method to sample k-clique graphs with $k \leq n$. Testing the existence of a complete subgraph of k nodes, also called a k-clique, is in general a hard problem [44]. The stochastic process of removing edges according to an arbitrary probability does not ensure that the resulting graph contains such a subgraph. Of course, lowering the probability of deletion increases the probability of finding larger groups of nodes connected to each other, but only a complete graph ensures the presence of a k-clique. That is, even if the sampling method is efficient, testing if the sought property holds is not.

On the other hand, the question "does a graph G have an Eulerian cycle[23]?" or equivalently "is a sampled graph G Eulerian?" can be answer by running any known algorithm that attempts to solve this problem. Carl Hierholzer in [37] not only provided an efficient algorithm to solve this problem, but proved that in order to be Eulerian, a graph must be connected and have only vertices of even degree. That is, considering the language of connected graphs L_C and the language of graphs with even degree vertices L_2, the language of Eulerian graphs is defined as $L_E = L_C \cap L_2$. In our case, if the property can be expressed by a context-free hyperedge replacement language, it is directly included into a representative grammar.

What about term graphs? Is there an efficient way to test if the generated graph is a valid term graph? Instead of trying to answer this question, differently from the classic methods, we start from a language and sample a graph directly in that specific domain. The only requirement is for the language to be represented by a context-free hyperedge replacement grammar and, if seeking a uniform probability distribution, for such a grammar to be non-ambiguous.

3.1 Mairson's Generation Methods for Strings

In 1994 H. G. Mairson proposed a pair of efficient methods [47] based on the studies of T. Hickey and J. Cohen [36] for the sampling of strings from context-free grammars. His approach required, as input, a grammar G in Chomsky normal form and the length of the word to be sampled. Conceptually the method used a progressive construction of the word by recursively choosing a production from the grammar at each step and ensuring that the length of the sentential forms didn't decrease. He also proved that, if the grammar is non-ambiguous, such a word is generated uniformly at random.

Let $G_{(CNF)} = (V, T, P, S)$ be a string grammar in Chomsky normal form where:

- V is a set of non-terminals.
- T is a set of terminals, with $V \cap T = \emptyset$.
- P is a set of productions.
- $S \in V$ is the starting symbol.

Each production $p \in P$ is in one of the following forms:

- $A \longrightarrow BC$
- $A \longrightarrow a$
- $S \longrightarrow \epsilon$

with $A, B, C \in V$, $a \in T$ and ϵ being the empty string.

Let $P^A \subseteq P$ be the subset of productions where $lhs(p) = A \in N$, $P_V \subseteq P$ the subset of non-terminal productions and $P_T \subseteq P$ the subsets of terminal productions.

Let L_ℓ be the ℓth slice of the language, that is, the subset of L consisting of all the words in L of length $|w| = \ell$.

For each non-terminal $A \in V$ let L^A be the set of all words w, such that there exists a derivation $A \Rightarrow^* w$.

Then, for each non-terminal $A \in V$ let $||A||_\ell$ define the number of all possible leftmost derivations from A to words of length ℓ. If the grammar is non-ambiguous, such words are pairwise distinct, that is $||A||_\ell = |L^A_\ell|$. Equally, each word in the language has a distinct leftmost derivation.

For each non-terminal production $A \longrightarrow BC \in P$ we also define $||A \longrightarrow BC||_\ell$ as the number of leftmost derivations yielding a word of length ℓ beginning with the application of $A \longrightarrow BC$.

The method is presented as a pair of algorithms **Pre** and **Gen**, the former being a pre-processing phase where the grammar is analyzed and the structures needed for the generation are built and the latter being the actual generation phase, where the word is recursively constructed.

Let $n \in \mathbb{N}$ be the length of a word $w \in L_n$ we would like to generate. The pre-processing phase consists of the computation of $||A||_\ell$ for each $A \in V$, for each ℓ in $1 \leq \ell \leq n$ and the computation of $||A \longrightarrow BC||_\ell$ for each $A \longrightarrow BC \in P$, for each ℓ in $2 \leq \ell \leq n$. This, according to [47], can be achieved by linear programming in $O(n)$ time and $O(n)$ space.

The generation phase begins by choosing any non-terminal A in V and the length ℓ of the word to be generated with $2 \leq \ell \leq n$. If $||A||_\ell = 0$, meaning that there is no word that can be generated from A in the ℓth slice of the language, the algorithm fails, otherwise it proceeds recursively according to the following cases:

1. If $\ell = 1$ then $p \in P^A_T$ is chosen uniformly at random with probability $1/||A||_\ell$ and $rhs(p)$ is returned.

2. If $\ell > 1$ then $p \in P_V^A$ is chosen with probability $||A \longrightarrow BC||_\ell / ||A||_\ell$. Since we konw that the grammar is in Chomsky normal form, it means that a non-terminal production has exactly 2 non-terminals on its *rhs*. We can then decide the lengths of the substrings of the words that are consequently generated by the first and second non-terminal. This is called a "split". Such a split $0 < k < \ell$ is chosen with probability $||B||_k \cdot ||C||_{\ell-k} / ||A \longrightarrow BC||_\ell$. Finally, the result of the concatenation of the words obtained by recursively applying the algorithm on B for a word of length k and C for a word of length $\ell - k$ is returned.

If the grammar is non-ambiguous, the word w, derived from A, is generated uniformly at random among all the possible words in L_ℓ^A. This is a direct consequence of having a distinct leftmost derivation for each word.

The time complexity of this algorithm is proven to be $O(n^2)$. The quadratic behavior is due to the cost of the choice of the split in **Gen**. This behavior is improved by the second method, which can generate a word in time $O(n)$ and space $O(n^2)$, requiring the computation of an equivalent grammar and binary trees to store the additional data to avoid the choice of the split.

3.2 First Method for Hypergraphs

For the settings of hyperedge replacement we begin with the adaptation of the first method that allows the random generation of a size-n-hypergraph in a user specified domain represented by a *HRG* in Chomsky normal form. Such a hypergraph is also generated uniformly at random in $L_n(G)$ if G is n-unambiguous. We present both proofs of quadratic time complexity and termination for the generation phase along with the proof of uniformity of distribution. As for the pre-processing phase, since we use the short-hand form of the productions presented in section 2.4, the time and space complexity are equivalent to the string method. Since the generation phase uses the structures derived from the previous phase, we only take into account the space needed to construct the hypergraph. It may be intended both as the sequence of replacements that needs to be applied to the initial handle in order to transform it into the final graph, or the concrete structure of the final graph itself. Clearly it cannot exceed the size of the generated graph.

To better understand the application of the method we use two relevant examples: the domain of term graphs and a simple ad-hoc grammar highlighting all the details of the procedure. To make the transition from strings easier we keep the name **Pre** and **Gen** for the algorithms. From now on, unless specified, all the reference to the methods, algorithms and functions will always be to the hyperedge replacement versions.

3.2.1 Pre-processing phase

The pre-processing phase is used to analyse the grammar and construct the structures needed in the generation phase. Since we do not need the full structure of the *rhs* of the applied productions, we can use the short-hand version of the grammar, presented in Section 2.4 as input. For each type of production of the Chomsky normal form, we need to keep track of:

- the id of the production
- the label and type of the replaced hyperedge in the *lhs*
- the labels of the non-terminals and the number of additional nodes in the *rhs*

In this phase, we may discard all the topological information of the *rhs* of the productions, such as the attachment nodes of the hyperedges or how these are arranged.

We also need as input an integer n representing the size of the hypergraph we want to generate. Since we know that the empty graph can only be generated if the grammar contains an empty production from the starting graph, we avoid this trivial case and suppose that the input size is greater than 0. So we build our tables for a size from 1 to n. We use two different tables to better highlight the difference between the number of graphs that can be generated during the process when the choice of a non-terminal or the choice of a production is required. We also include the terminal productions even if they can produce only one size of graph. We have chosen this format for a better clarity.

Formally, let $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ be an *HRG* in *CNF* and let $n \in \mathbb{N}$ be the size of the hypergraph $H \in L_n(G)$ to be generated, then **Pre** (Alg. 2) produces the pair of matrices M_1, M_2 required for the generation phase.

We begin initializing the entries of two matrices $M_1 = (N \times \mathbb{N})$ and $M_2 = (P \times \mathbb{N})$ to 0 (Tab. 3.1). Each entry (A, ℓ) of M_1, also denoted as $A[\ell]$, represents the number of derivations yielding a hypergraph of size $\ell + \mathit{type}(A)$, from a non-terminal $A \in N$. Each entry (p, ℓ) of M_2, also denoted as $p[\ell]$ represents the number of derivations yielding a hypergraph of size $\ell + |ext_R|$, from a production $p \in P$. According to the type of production they are also denoted as:

- $A \xrightarrow{p} BC, i[\ell]$ for a non-terminal production
- $A \xrightarrow{p} a, i[\ell]$ for terminal productions
- $A \xrightarrow{p} \lambda, i[\ell]$ for terminal productions containing only nodes

Considering each terminal production $p \in P_T$, either yielding a single terminal hyperedge $A \xrightarrow{p} a, i$ or at least a single isolated node $A \xrightarrow{p} \lambda, i$, the corresponding M_2 entry $p[i+1]$ in the former case, or $p[i]$ in the latter, is set to 1.

Then, for each $\ell \in \mathbb{N}$ in $1 \leq \ell \leq n$, for each non-terminal $A \in N$, $A[\ell] = \sum_{p \in P^A} p[\ell]$ and for each production $p \in P_N$, $p[\ell] = \sum_{0<k<\ell} B, [k] \cdot C[\ell - k]$.

The matrices can be used to generate hypergraphs in $L^A(G)$ of size $\ell + \mathit{type}(A)$, with $1 \leq \ell \leq n$ from any non-terminal $A \in N$. If the non-terminal A is chosen before the pre-processing phase we can reduce the size of the tables to $n - \mathit{type}(A)$. The gaps present in these matrices, represented by zero entries, are due to the possibility of a production to increase the size of the resulting hypergraph by more than 1. In the strings case instead, the application of each production contributes to the final length of the word by exactly 1 additional symbol, producing a matrix without zero entries. Even if in the context of hypergraphs a sparse matrix results in some of the splits being unavailable due to the impossibility to continue the derivation using a particular size from a non-terminal, the complexity of their choice remains the same. As in the string case, the entry of the tables can be used to deduce the ambiguity of a grammar if the size of the slice of the language is already known [43, 59].

Table 3.2 shows the result of algorithm **Pre** using as input the term graphs grammar G_{tg} in Figure 2.19 and 12 as the size of the sampled hypergraph. We should remember that since the handle induced by A is of $\mathit{type}(1)$, one node is already added at the beginning of the derivation, so the total number of size-12-hypergraphs in $L_{12}(G_{tg})$ are 3920, as shown in the entry $A[11]$ of M_1.

Algorithm 2: Pre - Pre-Process

```
Given: the short-hand version of a CNF grammar $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$
Input: $(G, n)$, where $n \in \mathbb{N}$, $n \geq 1$ is the size of the hypergraph to be sampled;
  Such a size also represents the maximum size of a hypergraph that can be
  generated from any non-terminal in $G$ using the resulting tables
Output: Two tables $\langle M_1, M_2 \rangle$ respectively representing the number of
  hypergraphs of size up to $n$ that can be generated either from each non-terminal
  $A \in N$ or a non-terminal production $p \in P_N$

Initialize tables $M_1$ and $M_2$ for each size $\ell$ from 1 to $n$
for $1 \leq \ell \leq n$ do
    For each non-terminal $A \in N$
    foreach $A \in N$ do
        Initialize the corresponding entry of $M_1$ for $A$ and a size $\ell$ to 0
        $A[\ell] := 0$;
    end
    For each non-terminal production $p \in P_N$
    foreach $p \in P$ do
        Initialize the corresponding entry of $M_2$ for $p$ and a size $\ell$ to 0
        $p[\ell] := 0$;
    end
end
Set the initial entries of $M_1$ and $M_2$ for each terminal production $p \in P_\Sigma$ having a
  terminal hyperedge $a$ on its rhs
foreach $A \xrightarrow{p} a, i \in P_\Sigma$ do
    Set the corresponding entry of $M_2$ for $p$ and a size $i + 1$ to 1
    $A \xrightarrow{p} a, i[i + 1] := 1$;
end
For each terminal production $p \in P_{\Sigma|}$ having only internal nodes in its rhs
foreach $A \xrightarrow{p} \lambda, i \in P_\Sigma$ do
    Set the corresponding entry of $M_2$ for $p$ and a size $i$ to 1
    $A \xrightarrow{p} \lambda, i[i] := 1$;
end
Fill the remaining entries of $M_1$ and $M_2$ for each size $\ell$ from 1 to $n$
for $1 \leq \ell \leq n$ do
    For each non-terminal $A$
    foreach $A \in N$ do
        For each production having $A$ on its lhs
        foreach $p \in P^A$ do
            Add the entry of $M_2$ for the production $p$ and size $\ell$ to the
              corresponding entry of $M_1$ for $A$ and the same size $\ell$
            $A[\ell] := A[\ell] + p[\ell]$;
        end
    end
    For each non-terminal production $p$
    foreach $A \xrightarrow{p} BC, i \in P_N$ do
        For each split represented by a size $k$ from 1 to $\ell - 1$
        for $1 \leq k < \ell$ do
            Add the entry of $M_1$ for the first non-terminal $B$ and size $k$ and the
              second non-terminal $C$ and size $\ell - k$ to the corresponding entry of $M_2$
              for $p$ and size $\ell + i$
            $A \xrightarrow{p} BC, i[\ell + i] := A \xrightarrow{p} BC, i[\ell + i] + B[k] \cdot C[\ell - k]$;
        end
    end
end
```

Table 3.1: Example of initialization of matrices M_1 and M_2.

M_1

N	1	2	$\cdots$	n-1	n
S	0	0	$\cdots$	0	0
A	0	0	$\cdots$	0	0
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$

M_2

P	1	2	$\cdots$	n-1	n
$S \xrightarrow{P1} AB, i$	0	0	$\cdots$	0	0
$A \xrightarrow{P2} BC, i$	0	0	$\cdots$	0	0
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$A \xrightarrow{Pu} a, 2$	0	1	$\cdots$	0	0
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$

Table 3.2: Matrices M_1 and M_2 resulting from $\textbf{Pre}(G_{tg}, 12)$

M_1

N	1	2	3	4	5	6	7	8	9	10	11	12
A	1	0	2	0	14	0	92	0	616	0	3920	0
B	2	0	8	0	32	0	128	0	256	0	512	0
C	0	0	2	0	32	0	76	0	488	0	2928	0
D	2	0	0	0	0	0	0	0	0	0	0	0

M_2

P	1	2	3	4	5	6	7	8	9	10	11	12
$A \xrightarrow{P1} CA, 1$	0	0	0	0	2	0	16	0	128	0	992	0
$A \xrightarrow{P2} BA, 1$	0	0	2	0	12	0	76	0	488	0	2928	0
$B \xrightarrow{P4} DB, 1$	0	0	4	0	16	0	64	0	128	0	256	0
$B \xrightarrow{P5} DB, 1$	0	0	4	0	16	0	64	0	128	0	256	0
$C \xrightarrow{P8} BA, 1$	0	0	2	0	12	0	76	0	488	0	2928	0
$A \xrightarrow{P3} 1, 0$	1	0	0	0	0	0	0	0	0	0	0	0
$B \xrightarrow{P6} +, 0$	1	0	0	0	0	0	0	0	0	0	0	0
$B \xrightarrow{P7} *, 0$	1	0	0	0	0	0	0	0	0	0	0	0
$D \xrightarrow{P9} +, 0$	1	0	0	0	0	0	0	0	0	0	0	0
$D \xrightarrow{P10} *, 0$	1	0	0	0	0	0	0	0	0	0	0	0

3.2.2 Generation phase

The generation phase uses the structures produced by the pre-processing phase to construct a hypergraph of a chosen size. Even if the matrices allow for the process to begin from any non-terminal for any of the pre-calculated sizes, for the examples we use the starting symbol of the input grammar and as size the same one chosen to run the pre-processing algorithm.

Formally, a non-terminal $\bar{A} \in N$ is chosen and a size-$\bar{n}$-hypergraph H, with $1 \leq \bar{n} \leq n + \mathit{type}(A)$, is generated using the data collected in the matrices M_1, M_2 and a pseudo-random number generator *RNG*. **Gen** (Alg. 3) describes this process.

On input $\mathbf{Gen}(G, \langle M_1, M_2 \rangle, \bar{A}, \bar{n} - \mathit{type}(A))$, if $\bar{A}[\bar{n} - \mathit{type}(A)] = 0$ the generating algorithm fails, otherwise, having $\bar{A}^{\bullet}$ as a basis, the algorithm recursively calls the function **derH** proceeding through the following steps:

1. The *RNG* is used to choose a production $p \in P^A$ with probability $p[\ell]/A[\ell]$.
2. If $p \in P^A_\Sigma$, the replacement of e, the *handle* of A, with the hypergraph R in $\mathit{rhs}(p)$ is returned.
3. If $p \in P^A_T$ the Random Number Generator is used again to choose a "split" $0 < k < \ell'$ with $\ell' = \ell - i$ and probability $B[k] \cdot C[\ell' - k]/A \xrightarrow{p} BC, i[\ell]$. The hypergraph $\mathit{rhs}(p)[e_\alpha/\mathbf{derH}(B,k), e_\beta/\mathbf{derH}(C, \ell' - k)]$ produced by the replacement of e_α with the result on the recursive function on input $\mathbf{derH}(B,k)$ and the replacement of e_β with the result of the recursive function on input $\mathbf{derH}(C, \ell' - k)$ is computed. Then, the replacement of the hyperedge e, the *handle* of A, with the aforementioned hypergraph is returned. We use the notation $B_k C_{\ell' - k}$ to indicate such a split.

The derivation $d = A^{\bullet} \Rightarrow^*_P H$ in Figure 3.1 corresponds to the sequence of replacements computed by the recursive function **derH** to generate the size-12-hypergraph T in Figure 2.1, using as input the grammar G_{tg}, the matrices in Table 3.2, the non-terminal A and 12 as size.

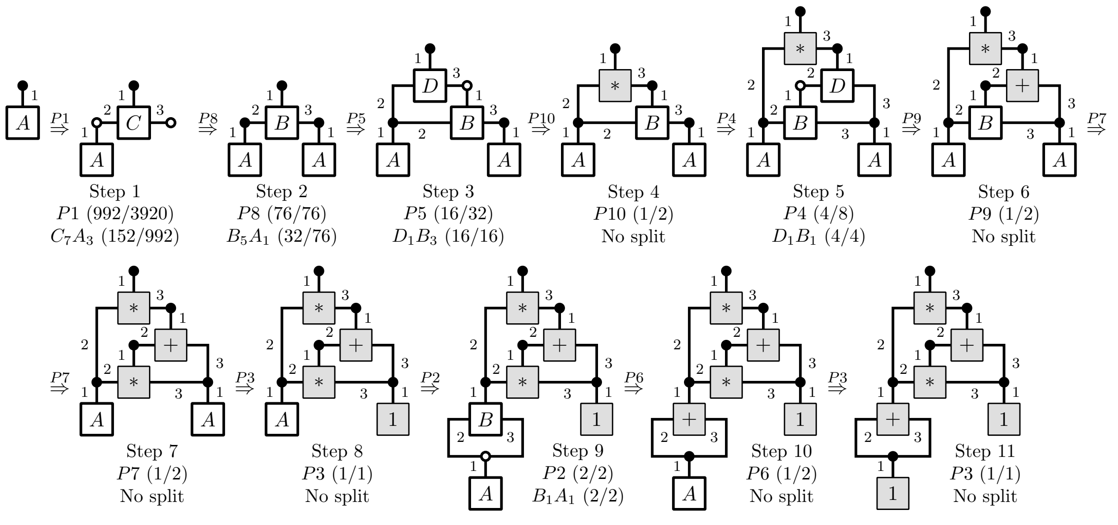


Figure 3.1: A derivation $d = A^{\bullet} \Rightarrow^*_P H$ using the grammar of Figure 2.19

For each step we show the probability of the production p to be chosen and the choice of the split along with its probability if $p \in P^N$. Since G_{tg} is non-ambiguous, the first step shows that $|L_{12}(G_{tg})| = 3920$, that is, there are 3920 unique size-12-hypergraphs to choose from, each having a different ordered derivation tree. Figure 3.2 shows the tree t for which $\mathit{yield}(t) = H$, so that $\mathit{trav}(t)$, or equivalently $\mathit{lmd}(H)$, corresponds to the unique sequence

Algorithm 3: Gen - Generate

Given: A grammar $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ in Chomsky normal form
Input: $(\langle M_1, M_2 \rangle, \bar{A}, \bar{n})$, where $\langle M_1, M_2 \rangle$ are the tables resulting from running **Pre** (Alg. 2) on input (G, n), $\bar{A} \in N$ is the chosen non-terminal to begin the generation from and $\bar{n} \in \mathbb{N}$, $1 \leq \bar{n} \leq n + \mathit{type}(\bar{A})$ is the size of the sought hypergraph
Output: A size-$\bar{A}$-hypergraph $H \in L_{\bar{n}}^{\bar{A}}(G)$

```
Set $\ell$ as the size of the elements that needs to be generated, that is, the input size
  minus the number of vertices of the handle induced by the non-terminal hyperedge
  induced by $\bar{A}$
$\ell = \bar{n} - \mathit{type}(\bar{A})$
If the slice of the language $L_{\bar{n}}^{\bar{A}}(G)$ is empty
if $\bar{A}[\ell] = 0$ then
    Terminate the algorithm
    return $\bot$;
end
Recursively generate the sought hypergraph $H$ using $(\bar{A}, \ell)$ as first input as follows:
function derH $(A, \ell)$:
    Use the $RNG$ for generating an integer in the interval $[1, A[\ell]]$ to select the
      corresponding production $p$ among all the productions having $A$ on their $lhs$
      with probability $p[\ell]/A[\ell]$
    $p \leftarrow RNG$ with $p \in P^A$ and probability $p[\ell]/A[\ell]$;
    If it is a terminal production
    if $p \in P_\Sigma$ then
        Return the replacement of the hyperedge $e$ labelled as $A$ by the terminal
          hypergraph $R$
        return $A^\bullet[e/R]$;
    else
        Subtract the number of internal nodes of $p$ to the remaining size of the
          generating hypergraph
        $\ell' = \ell - i$;
        Use the $RNG$ for generating an integer in the interval $[1, (A \xrightarrow{p} BC, i)[\ell]]$ to
          select the corresponding split among all the possible splits $B[k], C[\ell' - k]$,
          that is, the sizes of the subgraphs that will be generated from the first and
          second hyperedges in $rhs(p)$
        $k \leftarrow RNG$ with $0 < k < \ell'$ and probability $B[k] \cdot C[\ell' - k]/(A \xrightarrow{p} BC, i)[\ell]$;
        Continue the recursion replacing the first non-terminal $B$ with the
          hypergraph resulting from the function derH for a size $k$ and the second
          non-terminal $C$ with the hypergraph resulting from the function derH for
          a size $\ell' - k$
        return $A^\bullet[e/R[e_\alpha/\mathbf{derH}(B, k), e_\beta/\mathbf{derH}(C, \ell' - k)]]$;
    end
end function
```

of productions applied by the generation algorithm to produce H. In the figure are also indicated the starting symbol A and the replaced hyperedges e_α and e_β, respectively on the edges connecting the left and right child of each node.

To stress the difference with other algorithms, suppose we use the grammar in Figure 2.19 in a stochastic process where, at each step, each production is chosen uniformly at random among all the available ones. Since the grammar is context-free, we may assign these probabilities directly to the productions before the sampling, since the only requirement for their availability is the presence of a hyperedge with the label corresponding to their *lhs*. So, $P1$, $P2$ and $P3$ have $1/3$ of probability to be chosen, $P4$, $P5$, $P6$ and $P7$ have $1/4$, $P9$ and $P10$ have $1/2$. Since $P8$ is the only production for C it is always chosen. First of all we may notice that the algorithm is not ensured to terminate. For example, both $P1$ and $P2$ generate another hyperedge labelled as A. It means that there is a probability of $2/3$ that for the next step another choice needs to be made among the same productions, even if this probability gradually tends to 0. We may also notice that there is no way to ensure that the resulting graph has the chosen size.

Suppose now that the process generates the same hypergraph of Figure 3.1. The probability to obtain such a graph is $1/82944$ according to the probability of each production to be selected for replacing the correct labelled hyperedge and considering each choice to be independent from the previous one. This result is clearly not correct.

For a process similar to Gillespie algorithm instead, we may discard the part describing the time since we are interested in terminating the algorithm when a terminal graph is generated. As before we may notice again that there is no guaranteed termination. Considering all productions to have the same reaction rate of 1, the graph in Figure 3.1 is generated with an overall probability of $1/653184000$. In this case, we haven't considered any order of the replacement, because restricting the order would result in a process equivalent to the previous one. We may also notice that nothing prevents us from generating an isomorphic graph by following another order of replacements.

Lemma 3.2.1. *Given a grammar* $G = (N, \Sigma, P, S, (mark_p)_{p\in P})$*, Algorithm* ***Gen*** *terminates on any input* $(\langle M_1, M_2\rangle, \bar{A}, \bar{n})$*.*

The proof of termination of the generation algorithm is based on the assumption that the input grammar is non-contractive:

Proof. Let's consider a measure equivalent to the size of a hypergraph $|H|$. To each application of the recursive function **derH** in each step of **Gen** (Alg. 3), corresponds a direct derivation between two sentential forms $F \Rightarrow F'$ such that $|F| \leq |F'|$. Since the grammar is in *CNF*, at each step there are two possible cases:

1. **derH** chooses a non-terminal production. In this case a single hyperedge $e \in F$ is replaced with a hypergraph $R \subseteq F'$ containing 2 hyperedges and 0 or more internal nodes. Clearly $|F| < |F'|$, meaning that the size of the sentential forms gets progressively close to n.

2. **derH** chooses a terminal production. A hyperedge is replaced by a terminal hyperedge or a single node and 0 or more additional internal nodes. In this case $|F| \leq |F'|$. Even if the size is not incremented, being a terminal production, the recursion does not progress any further.

If it is not possible to generate a size-n-hypergraph using the input grammar G the algorithm trivially ends in one step. □

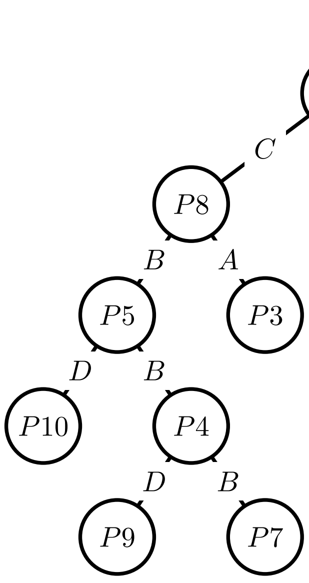


Figure 3.2: Ordered tree t for the derivation d in Fig. 3.1

3.2.3 First Method Running Example

This example is built on a very simple ad-hoc grammar presenting all the characteristic traits described so far providing a better understanding of the aspects of the generation process.

Let's consider the grammar $G_r = (N, \Sigma, P, S, (mark_p)_{p \in P})$, where the set of non-terminals $N = \{S, C, D\}$, the set of terminals $\Sigma = \{c, d\}$, S is the starting symbol and P is the set of productions shown in Figure 3.3. Since every production has either two non-terminals or one terminal on its *rhs*, we know that this grammar is already in Chomsky normal form. thus it is ready to be used for our generation process.

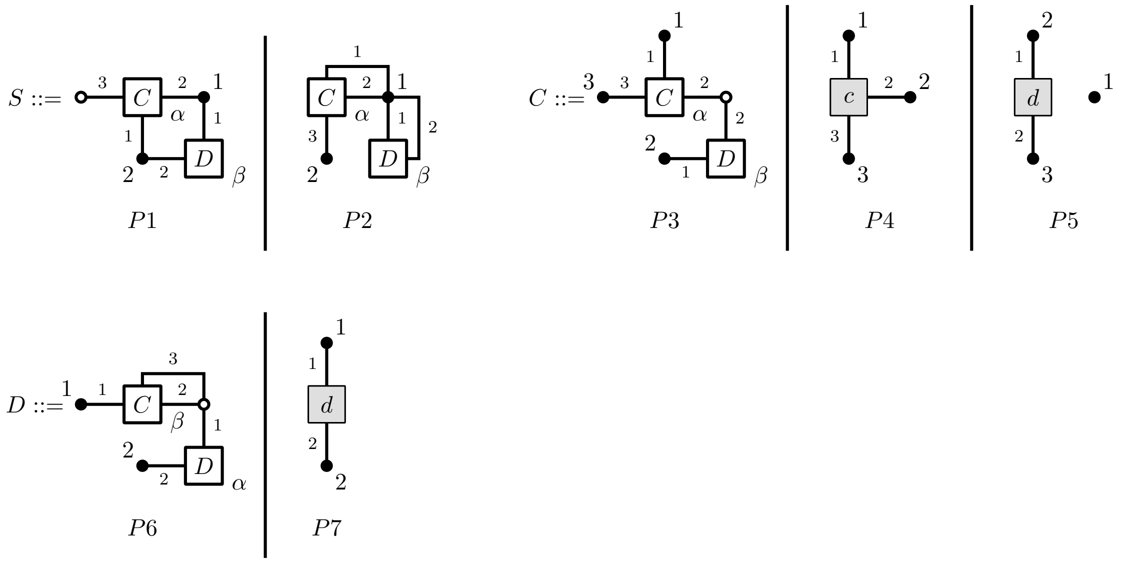


Figure 3.3: Running example grammar *RE* for Mairson's generation methods.

In order to achieve a sampling over a uniform distribution we also need to know if *RE* is non-ambiguous. Please note that this part of the example has only been shown for proving the uniform result of the sampling, but it is not required for the generation itself. Figure 3.4 shows the compact visualization of two ordered derivation trees for two size-6-

hypergraphs in $L_6(RE)$. The root represents the production applied to the starting symbol, while the other nodes represent the production applied to the non-terminal marked with α for the left child and with β for the right child. Figure 3.5 instead shows two leftmost derivations obtained by the pre-ordered visit of those trees. Since the yielded hypergraphs are isomorphic, while both the leftmost derivations and ordered trees are different, the grammar is ambiguous.

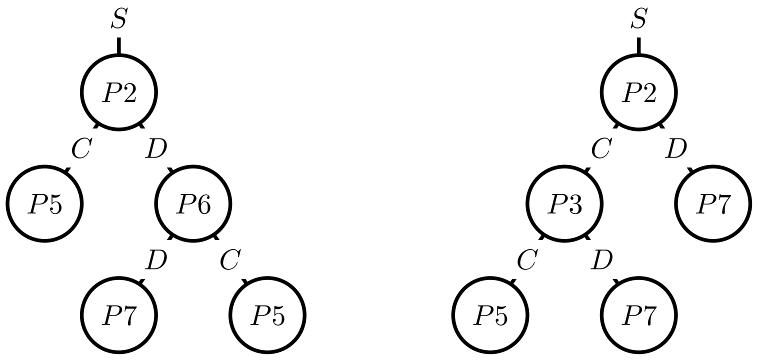


Figure 3.4: Ordered trees proving the ambiguity of the grammar *RE*.

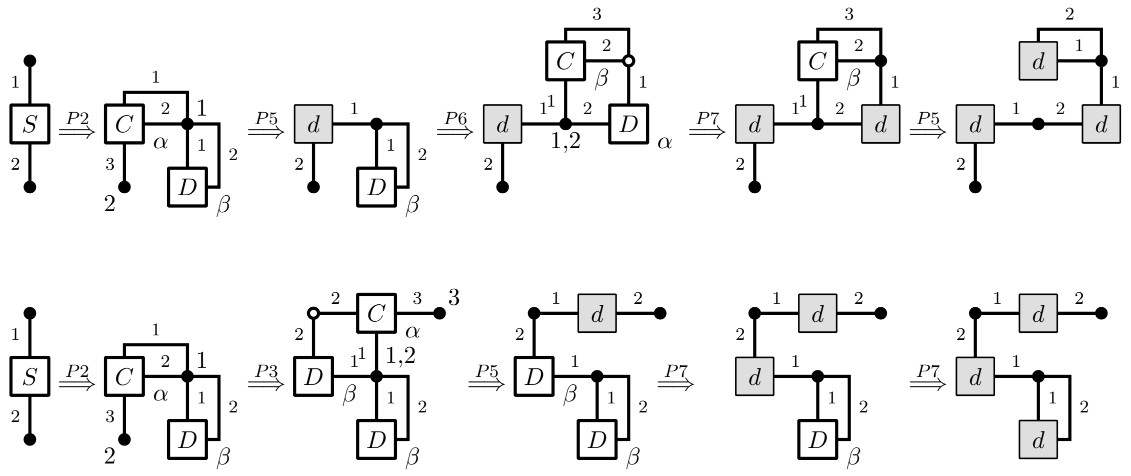


Figure 3.5: Derivations proving the ambiguity of the grammar *RE*.

Nevertheless if we consider the slice $L_7(RE)$ we notice that *RE* is also 7-unambiguous. To prove it, we can build the total language tree of the slice $L_7^S(RE)$. Figure 3.6 shows each possible derivation from the handle $S^\bullet$ to each of the 6 different size-7-hypergraph in the slice. The edges also show the applied production for each step of the derivation.

To begin the generation, first we extract all the information we need from the grammar and present it as their short version. Table 3.3 shows the short version of the 7 productions of *RE*. We may notice that each entry contains the labels, identifier and additional nodes of the original production.

For example $C \xrightarrow{P3} CD, 1$ shows that the original production is $P3$, C is the label of the 3-hyperedge on its *lhs*, C and D are the hyperedges marked as α and β and the hypergraph on its *rhs* also contains an additional node.

We use the pre-processing algorithm on input $(RE, 7)$ yielding the matrices $\langle M_1, M_2\rangle$ in Table 3.4. Let's remember that we need to take into account the *type* of the handle of the starting symbol when calculating the number of derivable graphs. So, for example, since *type*$(S) = 2$, the entry $S[5] = 6$, indicates that there are 6 different ways to generate a $|7|$-

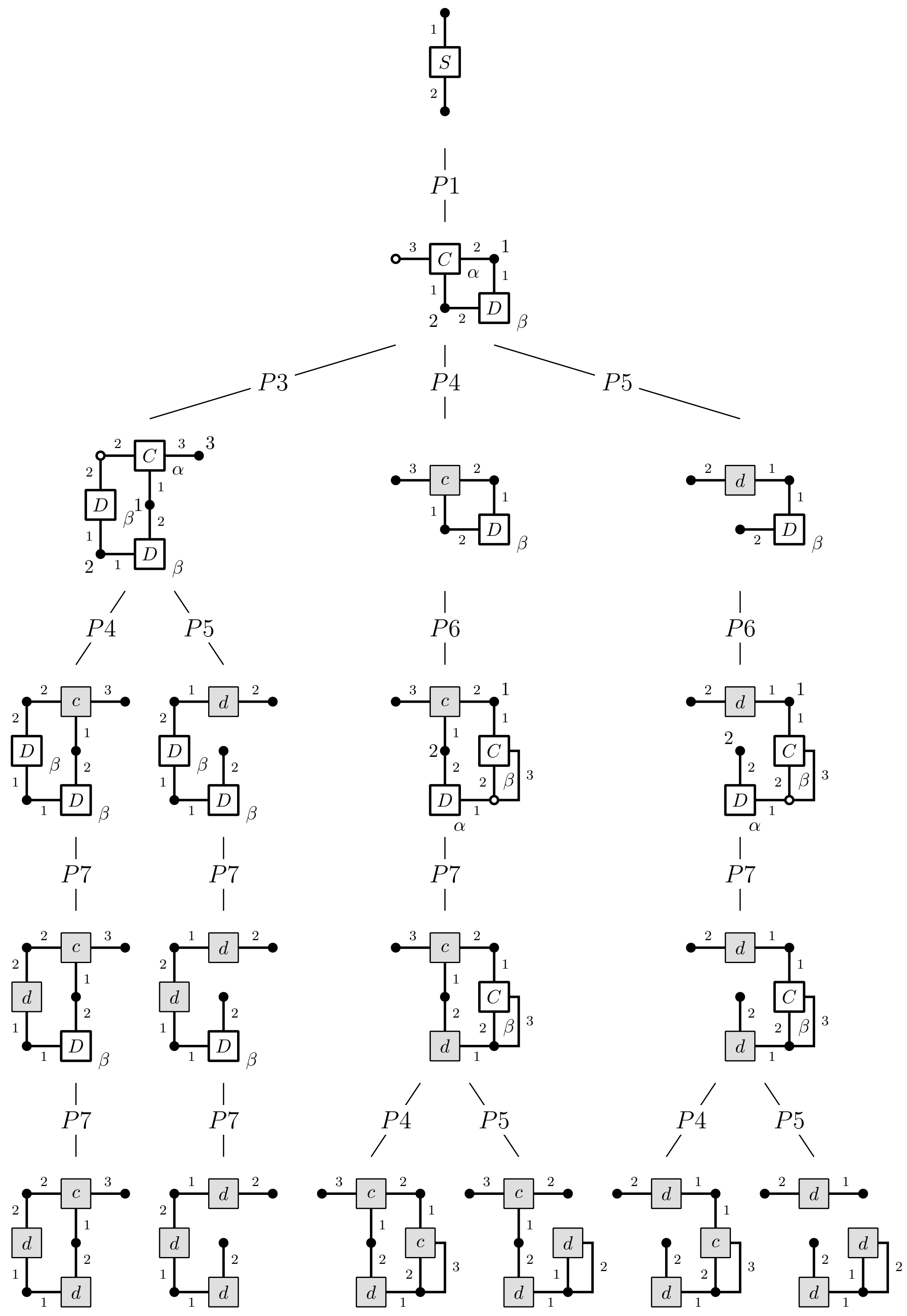


Figure 3.6: Total language tree for the slice of the language $L_7^S(RE)$

Table 3.3: Short version of the productions for the grammar RE

$S \xrightarrow{P1} CD, 1$	$C \xrightarrow{P3} CD, 1$	$D \xrightarrow{P6} DC, 1$
$S \xrightarrow{P2} CD, 0$	$C \xrightarrow{P4} c, 0$	$D \xrightarrow{P7} d, 0$
	$C \xrightarrow{P5} d, 0$	

hypergraph from the starting symbol S, while $S[7] = 22$ actually refers to the hypergraphs in L_9. Equivalently for the productions, $S \xrightarrow{P2} CD, 0[4] = 6$, indicates that there are 6 different ways to generate a $|6|$-hypergraph applying $P2$ as first production.

Table 3.4: Matrices M_1 and M_2 resulting from running $\mathbf{Pre}(RE, 7)$

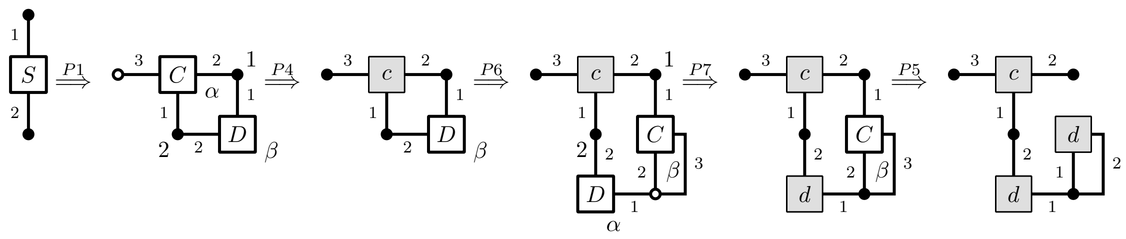
M_1

N	1	2	3	4	5	6	7
S	0	2	2	6	6	22	22
C	2	0	2	0	6	0	22
D	1	0	2	0	6	0	22

M_2

P	1	2	3	4	5	6	7
$S \xrightarrow{P1} CD, 1$	0	0	2	0	6	0	22
$S \xrightarrow{P2} CD, 0$	0	2	0	6	0	22	0
$C \xrightarrow{P3} CD, 1$	0	0	2	0	6	0	22
$D \xrightarrow{P6} DC, 1$	0	0	2	0	6	0	22
$C \xrightarrow{P4} c, 0$	1	0	0	0	0	0	0
$C \xrightarrow{P5} d, 0$	1	0	0	0	0	0	0
$D \xrightarrow{P7} d, 0$	1	0	0	0	0	0	0

We then use the generation algorithm on input $\mathbf{Gen}(RE, \langle M_1, M_2 \rangle, S, 7)$, to construct uniformly at random the derivation shown in Figure 3.7. Each step shows the production that has been applied and, in parentheses, the probability it has been chosen with. For non-terminal productions the chosen splits and their probabilities are also indicated.



Step 1 Prod.: $P1$ (6/6) Split: C_1D_3 (4/6)

Step 2 Prod.: $P4$ (1/2) No split

Step 3 Prod.: $P6$ (2/2) Split: D_1C_1 (2/2)

Step 4 Prod.: $P7$ (1/1) No split

Step 5 Prod.: $P5$ (1/2) No split

Figure 3.7: Leftmost derivation corresponding to the output of generation phase

Figure 3.8 shows the compact visualization of the ordered derivation tree corresponding to the derivation in Figure 3.7. Each node represents the production applied to the non-terminal on the *rhs* of its parent node. The non-terminal marked with α is shown on the left edge while the one marked with β on the right. Next to each node the probability of the choice of the production is reported, along with the one of the chosen split in case of it being non-terminal.

The generating process is described in details by the following steps:

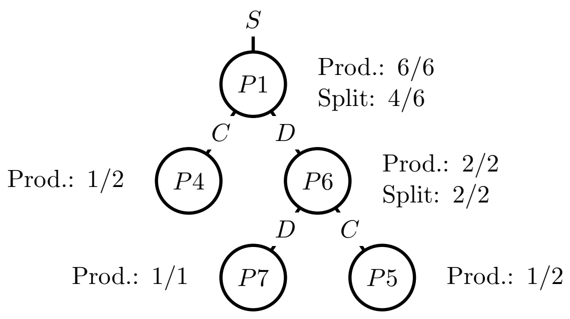


Figure 3.8: Ordered derivation tree corresponding to the output of the generation phase

Step 1 On input $\mathbf{Gen}(S, G, \langle M_1, M_2\rangle, 7)$, since $type(S) = 2$ then $\ell = 5$. $S[5] = 6$ in M_1, so there are 6 different derivations yielding a |7|-hypergraph from S. The algorithm proceeds calling the recursive function on input $\mathbf{derH}(S, 5)$. There is only one production $S \xrightarrow{P1} CD, 1[5] = 6$ in M_2 yielding a |7|-hypergraph having S on its *lhs*, so $P1$ is chosen with probability 6/6. Since the *rhs* of $P1$ includes 1 internal node then $\ell' = \ell - 1 = 4$. There are two possible split to chose from: C_1D_3 with probability $C[1] \cdot D[3]/S \xrightarrow{P1} CD, 1[5] = 2 \cdot 2/6 = 4/6$ and C_3D_1 with probability $C[3] \cdot D[1]/S \xrightarrow{P1} CD, 1[5] = 2 \cdot 1/6 = 2/6$. We suppose that the *RNG* chooses C_1D_3. The algorithm then recursively calls the function on input $\mathbf{derH}(C, 1)$ and $\mathbf{derH}(D, 3)$ and returns the replacement corresponding to $P1$.

Step 2 On input $\mathbf{derH}(C, 1)$, the function finds 2 possible derivations for $C[1]$. There are also 2 possible productions, $P4$ and $P5$, having the same probability to be chosen: $C \xrightarrow{P4} c, 0[1]/C[1] = 1/2$ for the former and $C \xrightarrow{P5} d, 0[1]/C[1] = 1/2$ for the latter. We suppose that the *RNG* chooses $P4$, since it is a terminal production the result of the hyperedge replacement corresponding to $P4$ is returned.

Step 3 On input $\mathbf{derH}(D, 3)$, the function finds 2 possible derivations for $D[3]$. There is only one production to choose with probability 2/2, that is $P6$. Since $i = 1$ then $\ell' = 2$. The only possible split is D_1C_1 with probability 2/2. The algorithm then recursively calls the function on input $\mathbf{derH}(D, 1)$ and $\mathbf{derH}(C, 1)$ and returns the replacement corresponding to $P6$.

Step 4 On input $\mathbf{derH}(D, 1)$, the function finds only 1 possible derivation for $D[1]$ and only one production with probability 1/1, that is $P7$. Since it is a terminal production the result of the hyperedge replacement corresponding to $P4$ is returned.

Step 5 On input $\mathbf{derH}(C, 1)$, as we have seen in Step 2, the function finds 2 possible derivations for $C[1]$. We suppose that this time the *RNG* chooses $P5$ with probability 1/2. Since it is a terminal production the result of the hyperedge replacement corresponding to $P5$ is returned.

Let H be the |7|-hypergraph generated by the algorithm, the reconstruction of the recursive steps taken by **derH** in Equation (3.1) proves the correctness of our method.

$$H = S^\bullet[e_\alpha/rhs(P1)[e_\alpha/rhs(P4), e_\beta/rhs(P6)[e_\alpha/rhs(P7), e_\beta/rhs(P5)]]] \tag{3.1}$$

The probability to generate this hypergraph is:

$$\frac{6}{6}\frac{4}{6}\frac{1}{2}\frac{1}{1}\frac{2}{2}\frac{2}{2}\frac{1}{2} = \frac{1}{6} \tag{3.2}$$

As expected, this result corresponds to the entry $S[5] = 6$ of Table 3.4, as proven in Section 3.2.4.

3.2.4 Uniform Probability Distribution

We now state our first main result, the uniform generation guarantee for Algorithm **Gen**.

Theorem 3.2.1. *Given a grammar $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$, Algorithm **Gen** generates from every non-terminal $A \in N$ a size-n-hypergraph $H \in L_n^A(G)$, provided that $L_n^A(G) \neq \emptyset$. If G is n-unambiguous and RNG is a uniform random number generator, the hypergraph is chosen uniformly at random.*

Proof. Let G be a n-unambiguous grammar in *CNF*, the recursive function **derH** derives a hypergraph $H \in L_{\bar{n}}^{\bar{A}}(G)$ simulating *trav*(t) where *yield*$(t) = H$ and let $P(c_j)$ denote the probability of the jth choice c made using the *RNG* at each step of the recursion, for a production or a split, according to *lmd*(H).

Let's remind that for the parallelization, confluence and associativity properties of context-free hyperedge replacement grammars [12], the sequence of replacements associated to a derivation preserves the result of the derivation, despite of the order in which the replacements are applied. Thus, we are able to discuss each of its steps independently.

By definition, since the grammar is n-unambiguous, for any non-terminal $A \in N$ we know that the set of hypergraphs that can be generated using different productions $p \in P^A$ are pairwise distinct. Otherwise, there would exist *trav*$(t') \neq$ *trav*(t'') for which *yield*$(t') \cong$ *yield*(t'').

From **Pre** (Alg. 2) we know that $\sum_{p \in P^A} p[\ell] = A[\ell]$ and so the probability of the choice c_j of each production in *lmd*(H) can be expressed by $P(c_j) = p[\ell]/A[\ell]$. Also, if $p \in P_N$, since the grammar is n-unambiguous the subsets of hypergraphs that can be derived by choosing different splits are also pairwise distinct. For a production $p \in P_N$ then $\sum_{0<k<\ell} B[k] \cdot C[\ell' - k] = A \xrightarrow{p} BC, i[\ell]$, thus a split can be chosen with probability $P(c_j) = B[k] \cdot C[\ell' - k]/p[\ell]$.

Knowing that for an *lmd*, if the grammar is n-unambiguous, both the choices of productions and splits are made from independent sets, considering the corresponding derivation tree t, the probabilities associated to the choice of a node $P(c)$ and the ones associated to its children $P(c')$ and $P(c'')$ are of the form $\frac{m}{q}$, $\frac{m'}{q'}$ and $\frac{m''}{q''}$ with $m, m', m'', q, q', q'' \in \mathbb{N}$ and $q'q'' = m$. Moreover, the probabilities of two consecutive choices $P(c)$ and $P(c')$ are bound to the law of compound probabilities [45], that is, the choice of a node given the choice of its parent is of the form $P(c'|c) = P(c' \cap c)/P(c)$. Then, considering their independence, $P(c'|c) = (P(c')P(c))/P(c) = P(c')$. The same applies for $P(c'')$. The overall probability of the choice of a node and its children is then $P(c)P(c')P(c'') = \frac{m}{q}\frac{m'}{q'}\frac{m''}{q''} = \frac{m'm''}{q}$.

Finally, considering the chain of probabilities described by an *lmd*, since for $\bar{A}$ $q = |L_{\bar{n}}(G)|$ and for each terminal production $p \in P_\Sigma$ $m = 1$, then for each $H \in L_{\bar{n}}(G)$ we can define its probability $P(H)$ to be generated as the productory of independent choices:

$$P(H) = \prod_{j=1}^{k} P(c_j) = \frac{m_1}{|L_{\bar{n}}(G)|} \cdot \frac{m_2}{q_2} \cdot \frac{m_{k-1}}{q_{k-1}} \cdots \frac{1}{q_k} = \frac{1}{|L_{\bar{n}}(G)|}$$

Each hypergraph $H \in L_{\bar{n}}(G)$ is generated over a uniform distribution given the uniformity of the sampling of the underlying *RNG*.

□

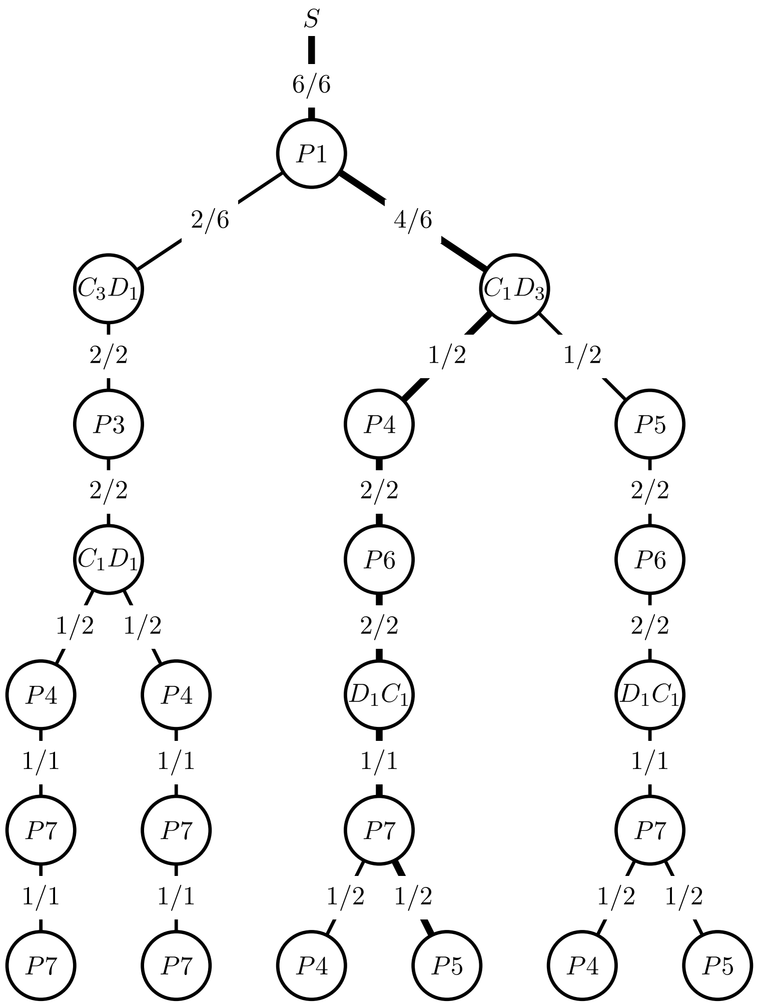


Figure 3.9: Graphical representation of all the possible choices of **Gen** (Alg. 3)

Figure 3.9 is a graphical visualization of a tree representing all possible outcomes of algorithm **Gen** with input $(G_r, \langle M_1, M_2 \rangle, S, 7)$. Each node represents the choice of a production and the subsequent choice of a split. The value on the edges represents the probability of the production or a split to be chosen, while the path taken by the running example is highlighted in bold. We may also notice the similarity with the total language tree for $L_7^S(G_r)$ in Figure 3.6. Merging the nodes representing the choices of splits into their parent node, representing the choice of a production, results indeed into an isomorphic tree.

3.2.5 Time and Space Complexity

For the complexity analysis we consider that the input grammar is already provided in a correct Chomsky normal form and the query to the random number generator as well as the replacement operations are performed in unit time.

Let's first consider the complexity analysis of the pre-processing Algorithm **Pre**. Since we have used a short-hand version of the productions (Tab. 3.3) we may notice that the only difference with the string counterpart is the addition of the number of internal nodes i. We may also notice that the fewer internal nodes we have in the *rhs* of a production, the more it takes to reach the sought size n.

Lemma 3.2.2. *Algorithm **Pre** produces the tables M_1 and M_2 needed for the generation of a size-n-hypergraph in time $O(n)$.*

Proof. Considering the worst case being a production that doesn't generate any internal nodes, this corresponds exactly to the string case. That is, the time required to populate the pre-processing tables is the same of the one presented in [47], which is already proven to be linear in n by means of dynamic programming. □

As for the space required to store the tables, we may notice that the gaps present in the tables (for example Tab. 3.4), that are not encountered in string method, are due to the possibility of a production to increase the size of the sentential form by more than 1 in a single step. Again, in the absence of internal nodes, we have a complexity equivalent to the string case.

Lemma 3.2.3. *The space required by Algorithm **Pre** to store the tables M_1 and M_2 needed for the generation of a size-n-hypergraph is in $O(n)$.*

Proof. For the short-hand version of a Context-free *HRG* in Chomsky normal form presented in Chapter 2, we know that no additional space is required to describe the productions besides the integer indicating the number of internal nodes. Considering the size of the grammar as a constant, we may easily understand that the tables only grow linearly in one direction according to the input size n, resulting in a linear asymptotic behavior $O(n)$. □

In this work, to ease the understanding of the generation process we try to show, when possible, the full tables, but it is easy to see how they could be contracted in case of 0-filled columns or terminal productions.

For the generation phase we consider the space required as the total size of data that is necessary to describe the generated hypergraph, which, according to our process, it is considered as the structure built by the recursion function **DerH**. Since such a hypergraph can be concretely described by either its nodes and edges or, equivalently, by the sequence of replacements described by the applied productions from the original handle to its terminal form, and, considering that each step increases the size of the sentential form of at least *one*, we state that the space required by the generation algorithm is linear in n.

The time complexity of the generation phase is expressed in the following theorem:

Theorem 3.2.2. *With the assumptions of Theorem 3.2.1, the size-n-hypergraph H is generated by* **Gen** *(Alg. 3) in time $O(n^2)$.*

Proof. The proof of Theorem 3.2.2 is based on the analysis of the following recurrence relation for the function **derH**: $T(n) \leq cn + \underset{1 \leq k < (n-i)}{max} [T(k) + T(n-k-i)]$, where $T(k)$ and $T(n-k-i)$ are the computational steps required to process the result of the split and i is the number of internal nodes of the current production. In the worst case, we consider that $i = 0$ and that $k = 1$. A simple example is the discrete hypergraph language in which every iteration may generate a terminal hyperedge from e_α and the rest of the resulting hypergraph from e_β without adding any node. Since the choice of the production is constant, while the choice of a split is linear in n, choosing a split n times leads to a quadratic behavior.

Since $i \ll n$, we may rewrite the recursion as:

$$T(n) \leq cn + \underset{1 \leq k < n}{max} [T(k) + T(n-k)]$$

Then, considering the worst case $k = 1$, for the next step of the recursion we obtain:

$$T(n-1) \leq c(n-1) + \underset{1 \leq k < (n-1)}{max} [T(k) + T(n-k-1)]$$

That is, at each step the choice of a split happens on an input of size $n-1$. Since this choice requires linear time and it is taken n times, the relation has solution $O(n^2)$. □

Lemma 3.2.4. *The space required by Algorithm* **Gen** *for the generation of a size-n-hypergraph is in $O(n)$.*

Proof. The analysis of the space complexity of **Gen** (Alg. 3) takes into account all the data that is dynamically constructed during the production of the hypergraph. That excludes the tables needed for holding the number of derivable graphs, which space has already being accounted for the pre-processing phase. Considering the data needed by each replacement described by the recursive function **DerH** for a size-n-hypergraph as a constant, in the worst case, the number of steps required is equivalent to the nodes of the derivation tree and such data is held until the hypergraph is fully constructed. Since it is a binary tree, we know that there are exactly $2n - 1$ nodes, that is, the space needed is linear in n. Additionally considering the space required to hold the data of the hypergraph to be trivially n, the overall space required by the algorithm is clearly $O(n)$. □

3.3 Second Method for Hypergraphs

We now move to the adaptation of the second method presented by Mairson allowing for the generation of hypergraphs in linear time using quadratic space. As we see in Section 3.2, the bottleneck of the first method is the choice of the split for a non-terminal production. Once a number is generated by the *RNG*, it required linear time to go through the available splits and pick one of them. The improvement presented in the second method is a faster selection of the split, at the cost of the space required to hold additional structures. To achieve this result we make use of the encoding proposed by Huffman in [39]. As for the pre-processing tables, once these structures have been computed there is no need to generate them again and will be available to use throughout the whole generation process. For this reason, we consider the time required to calculate them as well as the space to hold them as part of the pre-processing phase.

3.3.1 Huffman codes

Originally designed to work on the compression of messages for a more efficient transmission, its importance rapidly spread across all fields of computer science. The encoding is based on the concept that symbols with a high frequency, that is, more often to be found in a message, should be encoded using shorter codes, while less frequent symbols may be represented by longer codes. Knowing the frequency of each symbol is the only requirement for both the encoding and decoding of a message. Without loss of generality, let the frequency be indicated by a positive integer weight $w_n \in \mathbb{N}$ for a node n, then the underlying structure is a weighted ordered full binary tree called Huffman tree. In details:

- Full Binary: the tree is rooted and every internal node has exactly two children.
- Ordered: the children of each node are ordered. In the original Huffman coding algorithm, the left child is indexed by 0, while the right one by 1. We keep this ordering to represent the sequence corresponding to the path leading to the chosen split. In our case we are more interested in the length of such a sequence rather than the values it contains.
- Weighted: every node has an additional attribute. Originally, it represents the frequency of the symbols in a string. In our case it is the number of hypergraphs that can be derived by a split.

It is important for us to consider how the size and height of a full binary tree vary according to the number of leaves: larger trees require more space for being stored while higher trees require longer to be traversed.

Proposition 3.3.1 (Size of a Full Binary Tree)**.** *If t is a full binary tree and n is the number of its leaves, the tree has exactly $2n - 1$ nodes.*

Proof. Let $|t|$ be the size of a full binary tree t represented by the number of its nodes and n the number of its leaves, then, inductively:

1. If t is composed by a single node, its only leaf, then trivially $|t| = 2n - 1 = 1$.
2. Adding a pair of leaves as children of an existing leaf of t increases the number of nodes by 2, but it also turns the parent leaf into an internal node, increasing the number of leaves only by 1. Thus, if t' is a full binary tree obtained by growing t by a pair of leaves then $n' = n + 2 - 1 = n + 1$ while its size $|t'| = 2n - 1 + 2 = 2(n + 1) - 1 = 2n' - 1$.

   Notably, adding one leaf only, results in the need of a new internal node to preserve the binary and fullness constraints.

□

Proposition 3.3.2 (Height of a Full Binary Tree)**.** *If t is a full binary tree, n is the number of its leaves and h is its height, then $\lceil log_2 n \rceil + 1 \leq h \leq n$.*

Proof. Let h be the height of a full binary tree h and n the number of its leaves, then each leaf can potentially be the parent of a new pair. That is, in a complete binary tree, there are $2^{(h-1)}$ leaves.

1. If t is composed by a single node, its only leaf, then $h = \lceil log_2 1 \rceil + 1 = 1$.
2. Adding a pair of leaves as children of each existing leaf of t results in doubling their number. Thus, if t' is a full binary tree obtained by adding 2 leaves to each leaf of t then $n' = 2n$ and $h' = \lceil log_2 2n \rceil + 1 = log_2 2 + \lceil log_2 n \rceil + 1 = 1 + \lceil log_2 n \rceil + 1 = h + 1$.

That is, at each step the height of the complete binary tree is increased by 1 when adding $2n$ leaves to all the preexisting ones.

On the other hand, let now h be the height of an unbalanced full binary tree in which for every pair of child nodes at most one has further children, then $h = n$.

1. If t is composed by a single node, its only leaf, then $h = n = 1$.
2. Adding a new pair of leaves as children of an existing leaf of t increases both the number of leaves and height by 1, since we cannot add a new pair to a sibling that already has children. Thus, if t' is a new full binary tree obtained by adding 2 new leaves to each leaf of t then $n' = n + 1$ and $h' = n' = n + 1 = h + 1$.

□

We can now define a Huffman tree as follows:

Definition 3.3.1 (Huffman Tree)**.** Let w_v represent the frequency for a node v, then a Huffman tree h is inductively defined as follows:

1. A tree h composed by a single node v having weight w_v is a Huffman tree.
2. If h_0 and h_1 are Huffman trees with roots v_0 and v_1 having weights $w_{v_0} \leq w_{v_1}$, then a tree h composed by a node v and h_0 and h_1 as its left and right subtrees, so that the roots v_0 and v_1 are respectively the left and right child of v and $w_v = w_{v_0} + w_{v_1}$, is also a Huffman tree.

The property of the Huffman trees we are most interested in, allowing us to improve our generation method, is the ability to produce an *optimal encoding.* Since we are not working exactly with codes and frequencies, but rather with paths and splits, we construct our proof starting from the one given in [11]. We want to prove that if a split can derive a greater number of hypergraphs, it can also be chosen following a shorter path in the tree and also that such a tree gives an optimal ratio among the overall numbers of hypergraphs of each split and the corresponding paths.

3.3.2 Pre-processing Phase

In the second method the pre-processing phase is extended for accommodating the additional data structure needed to speed up the generation of the hypergraph. In order to simplify this approach, we begin with the construction of an equivalent grammar to take into account all the possible splits for each production. Since the choice of the split happens after the choice of the production and only affects the next recursive steps of the generation for the non-terminal hyperedges in the *rhs* of such a production, we can safely state that all different splits are based on the same replacement. Without a topological change in the application of a production with different splits, we can then directly work on the short-hand version of the productions presented in Section 2.4.

Given the short-hand version of a grammar $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ and tables $\langle M_1, M_2 \rangle$ resulting from running $\mathbf{Pre}(G, n)$, an equivalent short-hand version of grammar $G' = (N', \Sigma, P', S', (mark_p)_{p \in P'})$ such that $L_n(G) = L_n(G')$ can be computed by the following steps:

1. for each $A \in N$, for each $0 < j \leq n$, if $A[j] \neq 0$ define $A_j \in N'$.
2. for each $A \xrightarrow{p} BC, i \in P_N$, for each $2 \leq j \leq n$, if $A \xrightarrow{p} BC, i[j] \neq 0$, for each $0 < j - i \leq n$, if $B[k] \neq 0$ and $C[j-i-k] \neq 0$ define $A_j \xrightarrow{p} B_k C_{\ell'-k}, i \in P'_{N'}$, with $0 < k < j - i$.
3. for each $A \xrightarrow{p} a, i \in P_\Sigma$, if $A[j] \neq 0$ define $A_j \xrightarrow{p} a, i \in P'_\Sigma$.

The original grammar G is still referenced for the application of the replacements during the generation phase, while G' is only used in the pre-processing algorithm, where, this time, each entry indicates the number of hypergraphs that can be obtained not only by each production but also by each split. For each production $p \in P_N$, a Huffman tree denoted as $h_\ell[p]$, with $2 \leq \ell \leq n$, is constructed using **Enc** (Alg. 4). Each leaf represents a possible choice of a split $B_k C_{\ell'-k}$ while its weight is the number of hypergraphs that can be derived choosing such a split $B[k] \cdot C[\ell' - k]$.

3.3.3 Generation Phase

The generation phase presents a significant change, comparing to the first method, in the way the split is selected during the following step:

$$k \leftarrow \mathit{RNG\ with}\ 0 < k < \ell'\ \mathit{and\ probability}\ B[k] \cdot C[\ell' - k]/(A \xrightarrow{p} BC, i)[\ell];$$

In this modified algorithm, **Gen'**, instead of going through all the available splits, we use the Huffman trees produced in the pre-processing phase, to quickly select the split corresponding to the interval that includes the integer generated by the *RNG*. Once a production is chosen, we navigate the tree from the root to a leaf, if the generated number is lesser or equal than the weight of the left child we follow the path to the left, otherwise the one to the right. We repeat this operation until a leaf is reached, its label representing the chosen split. **Sel** (Alg. 5) recursively performs such a selection.

3.3.4 Second Method Running Example

Running **Pre** with input $(G', 7)$ yields the matrices M_3 and M_4 in Table 3.6. We then proceed, as described in [47], to build the binary trees $\tau_\ell[A \xrightarrow{p} BC, i]$, also denoted as $\tau_\ell[p]$, for each production $p \in P_N$ and $2 \leq \ell \leq n$ used to optimize the selection of the splits. Each leaf of the tree represents a possible split $B_k C_{\ell'-k}$ and has a weight $w = B[k] \cdot C[\ell' - k]$. Each internal node's weight is equal to the sum of the weight of its left and right children $w = w' + w''$. To build such tree we can use the same approach proposed by Huffman in [39].

For example, to produce the tree $\tau_6[P2]$, for selecting a split $B_k C_{\ell-k}$, with $\ell = 6 - 0 = 6$ and $1 \leq k < \ell$, we proceed according to the following steps described in Figure 3.10.

Step 1 The weight corresponding to the possible splits $C_1 D_5$, $C_3 D_3$ and $C_5 D_1$ are arranged in a sequence of single node trees according to their weight in ascending order.

Step 2 The trees having the roots with the lowest weight, 4 and 6, are removed from the sequence and added as the left and right child of a new subtree having $4 + 6 = 10$ as root.

Step 3 The trees having root 10 and the 12, are joined into a single tree having $10 + 12 = 22$ as root. Since the original forest has been reduced to a single binary tree, our structure is now complete.

Algorithm 4: Enc - Splits Encoding

Given: the short-hand version of a grammar $G' = (N', \Sigma', P', S', (mark_p)_{p \in P'})$; the output $\langle M_1, M_2 \rangle$ resulting from **Pre**(G', n); a set of splits represented by the *rhs* of a set of productions $A_x \xrightarrow{p} B_j C_k, i \in P'_{N'}$ referring to the same production p and having the same $lhs = A_x$.

Input: a labelled weighted discrete graph D where each node v represents a distinct split, with $B_j C_k$ being its label and $w = A_x \xrightarrow{p} B_j C_k, i[x]$, the value of the corresponding entry of M_2, its weight.

Output: a weighted ordered binary tree constructed from D having its nodes as leaves and each internal node having its weight equal to the sum of the weights of its children.

```
Let $\omega$ be an empty sequence of pointers to nodes of the graph $D$ and $|\omega|$ its length.
Insert a pointer to each node of $D$ into $\omega$, in descendant order according to their
  weights:
foreach node $v$ in $D$ do
    $insertSort(\omega, v)$;
end
Until there are less than 2 pointers in the sequence $\omega$:
while $|\omega| > 1$ do
    Create a new node $\hat{v}$ in $D$:
    $\hat{v} \leftarrow addNode(D)$;
    Set the weight of $\hat{v}$ to the weight of the node pointed by the last element of the
      sequence:
    $\hat{v}.w = getLast(\omega).w$;
    Append the node pointed by the last element of the sequence as the left child of
      $\hat{v}$:
    $appendLeftChild(D, \hat{v}, getLast(\omega))$;
    Remove the last element of the sequence, so that the sequence has a new last
      pointer:
    $removeLast(\omega)$;
    Add to the weight of $\hat{v}$ the weight of the node pointed by the last element of the
      sequence:
    $\hat{v}.w = \hat{v}.w + getLast(\omega).w$;
    Append the node pointed by the last element of the sequence as the right child
      of $\hat{v}$:
    $appendRightChild(D, \hat{v}, getLast(\omega))$;
    Remove the last element of the sequence:
    $removeLast(\omega)$;
    Insert a pointer to the new node $\hat{v}$ into $\omega$:
    $insertSort(\omega, \hat{v})$;
end
```

Algorithm 5: Sel - Select Split

```
Given: the short-hand version of a grammar $G' = (N', \Sigma', P', S', (mark_p)_{p \in P'})$; the
  output $\langle M_1, M_2 \rangle$ resulting from $\mathbf{Pre}(G', n)$; a set of splits represented by the rhs
  of a set of productions $A_x \xrightarrow{p} B_j C_k, i \in P'_{N'}$ referring to the same production $p$
  and having the same $lhs = A_x$
Input: an ordered weighted binary tree $h$ representing the possible splits, where
  each leaf represents a distinct split for $p$, with $l = B_j C_k$ being its label and
  $w = A_x \xrightarrow{p} B_j C_k, i[x]$, the value of the corresponding entry of $M_2$, its weight; a
  uniformly generated at random number $r \longleftarrow RNG$ in the range $0 < r \leq \ell'$
Output: the label of a leaf representing the chosen split

Recursively navigate $h$ and return the label of the first encountered leaf, starting
  from the values $(getRoot(h), r)$
function selectSplit $(v, n)$:
    If $v$ has no children
    if $isLeaf(v)$ then
        return $v.l$;
    else
        If the weight of the left child is less or equal than $n$
        if $v.w \leq n$ then
            Continue the recursion on the left child of $v$ with $n$
            return selectSplit$(getLeftChild(v), n)$;
        else
            Continue the recursion on the right child of $v$ with subtracting the
              weight of the left child of $v$ from $n$
            return selectSplit$(getRightChild(v), n - getLeftChild(v).w)$;
        end
    end
end function
```

Table 3.5: Short version of the productions of the equivalent grammar G' for language $L_7(G')$

$S_3 \xrightarrow{P1} C_1D_1, 1$	$C_3 \xrightarrow{P3} C_1D_1, 1$	$D_3 \xrightarrow{P6} D_1C_1, 1$
$S_5 \xrightarrow{P1} C_1D_3, 1$	$C_5 \xrightarrow{P3} C_1D_3, 1$	$D_5 \xrightarrow{P6} D_1C_3, 1$
$S_5 \xrightarrow{P1} C_3D_1, 1$	$C_5 \xrightarrow{P3} C_3D_1, 1$	$D_5 \xrightarrow{P6} D_3C_1, 1$
$S_7 \xrightarrow{P1} C_1D_5, 1$	$C_7 \xrightarrow{P3} C_1D_5, 1$	$D_7 \xrightarrow{P6} D_1C_5, 1$
$S_7 \xrightarrow{P1} C_3D_3, 1$	$C_7 \xrightarrow{P3} C_3D_3, 1$	$D_7 \xrightarrow{P6} D_3C_3, 1$
$S_7 \xrightarrow{P1} C_5D_1, 1$	$C_7 \xrightarrow{P3} C_5D_1, 1$	$D_7 \xrightarrow{P6} D_5C_1, 1$
$S_2 \xrightarrow{P2} C_1D_1, 0$	$C_1 \xrightarrow{P4} c, 0$	$D_1 \xrightarrow{P7} d, 0$
$S_4 \xrightarrow{P2} C_1D_3, 0$	$C_1 \xrightarrow{P5} d, 0$	
$S_4 \xrightarrow{P2} C_3D_1, 0$		
$S_6 \xrightarrow{P2} C_1D_5, 0$		
$S_6 \xrightarrow{P2} C_3D_3, 0$		
$S_6 \xrightarrow{P2} C_5D_1, 0$		

Table 3.6: Matrix M_3 and M_4 resulting from running $\mathbf{Pre}(G', 7)$.

M_3

N	1	2	3	4	5	6	7
S_2	0	2	0	0	0	0	0
S_3	0	0	2	0	0	0	0
S_4	0	0	0	6	0	0	0
S_5	0	0	0	0	6	0	0
S_6	0	0	0	0	0	22	0
S_7	0	0	0	0	0	0	22
C_1	2	0	0	0	0	0	0
C_3	0	0	2	0	0	0	0
C_5	0	0	0	0	6	0	0
C_7	0	0	0	0	0	0	22
D_1	1	0	0	0	0	0	0
D_3	0	0	2	0	0	0	0
D_5	0	0	0	0	6	0	0
D_7	0	0	0	0	0	0	22

M_4

P	1	2	3	4	5	6	7
$S_3 \xrightarrow{P1} C_1D_1, 1$	0	0	2	0	0	0	0
$S_5 \xrightarrow{P1} C_1D_3, 1$	0	0	0	0	4	0	0
$S_5 \xrightarrow{P1} C_3D_1, 1$	0	0	0	0	2	0	0
$S_7 \xrightarrow{P1} C_1D_5, 1$	0	0	0	0	0	0	12
$S_7 \xrightarrow{P1} C_3D_3, 1$	0	0	0	0	0	0	4
$S_7 \xrightarrow{P1} C_5D_1, 1$	0	0	0	0	0	0	6
$S_2 \xrightarrow{P2} C_1D_1, 0$	0	2	0	0	0	0	0
$S_4 \xrightarrow{P2} C_1D_3, 0$	0	0	0	4	0	0	0
$S_4 \xrightarrow{P2} C_3D_1, 0$	0	0	0	2	0	0	0
$S_6 \xrightarrow{P2} C_1D_5, 0$	0	0	0	0	0	12	0
$S_6 \xrightarrow{P2} C_3D_3, 0$	0	0	0	0	0	4	0
$S_6 \xrightarrow{P2} C_5D_1, 0$	0	0	0	0	0	6	0
$C_3 \xrightarrow{P3} C_1D_1, 1$	0	0	2	0	0	0	0
$C_5 \xrightarrow{P3} C_1D_3, 1$	0	0	0	0	4	0	0
$C_5 \xrightarrow{P3} C_3D_1, 1$	0	0	0	0	2	0	0
$C_7 \xrightarrow{P3} C_1D_5, 1$	0	0	0	0	0	0	12
$C_7 \xrightarrow{P3} C_3D_3, 1$	0	0	0	0	0	0	4
$C_7 \xrightarrow{P3} C_5D_1, 1$	0	0	0	0	0	0	6
$D_3 \xrightarrow{P6} D_1C_1, 1$	0	0	2	0	0	0	0
$D_5 \xrightarrow{P6} D_1C_3, 1$	0	0	0	0	2	0	0
$D_5 \xrightarrow{P6} D_3C_1, 1$	0	0	0	0	4	0	0
$D_7 \xrightarrow{P6} D_1C_5, 1$	0	0	0	0	0	0	6
$D_7 \xrightarrow{P6} D_3C_3, 1$	0	0	0	0	0	0	4
$D_7 \xrightarrow{P6} D_5C_1, 1$	0	0	0	0	0	0	12
$C_1 \xrightarrow{P4} c, 0$	1	0	0	0	0	0	0
$C_1 \xrightarrow{P5} d, 0$	1	0	0	0	0	0	0
$D_1 \xrightarrow{P7} d, 0$	1	0	0	0	0	0	0

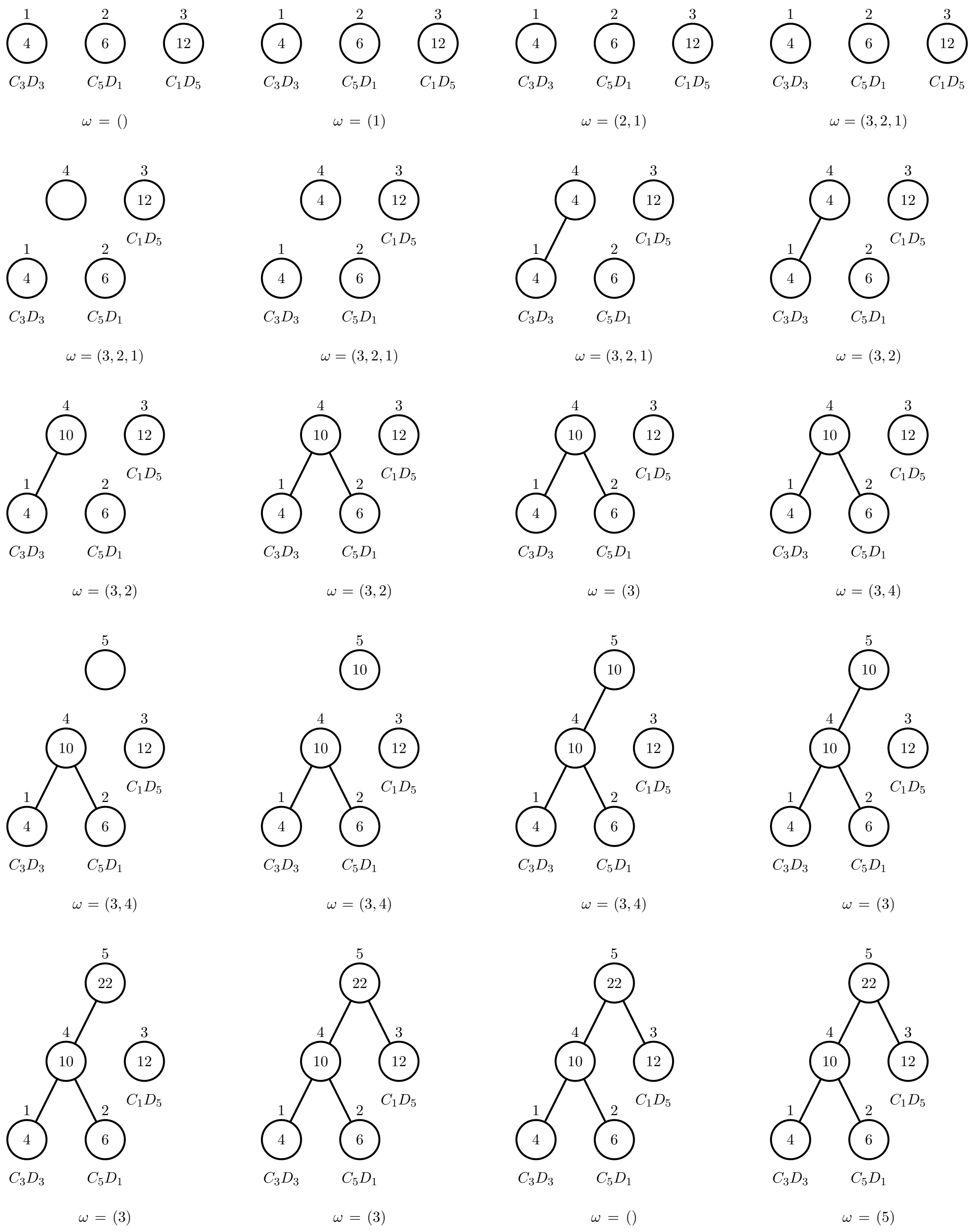


Figure 3.10: Example of Huffman encoding for the production $S \xrightarrow{P2} CD, 0$

To select the split for production $S \xrightarrow{P2} CD, 0$ we follow the path from the root of the binary tree to one of its leaves. For each internal node, having weight w, the left child, having weight w', has a probability w'/w to be chosen, while the right one, having weight w'', has a probability of $w''/w = 1 - w'/w$.

For example, starting from the root, $w = 22$, we may select its left child with probability $10/22$ and then the split C_5D_1 with probability $6/10$. As expected, the overall probability to choose this split is $10/22 \cdot 6/10 = 6/22$.

3.3.5 Uniform Probability Distribution

For the second method, we focus the analysis of the uniformity of the sampling to show that the modifications introduced for the choice of the split do not affect the probability distribution of the generation. In the first method the *RNG* is used to select a split $B_kC_{\ell'-k}$, among all the available ones for a given production p, each of them having probability $B[k] \cdot C[\ell' - k]/(A \xrightarrow{p} BC, i)[\ell]$, that is, the number of hypergraphs that can be derived using that split over the number of hypergraphs that can be derived using their relative production. We have already proven that choosing a split requires linear time, but, before getting rid of this bottleneck, it is fundamental to prove that the uniformity of the sampling is preserved when using the trees constructed with the Huffman coding.

Theorem 3.3.1. *Let **Gen'** be a modified algorithm obtained by selecting the splits in Algorithm **Gen** using Huffman trees, then given a grammar $G = (N, \Sigma, P, S, (mark_p)_{p\in P})$, **Gen'** generates from every non-terminal $A \in N$ a size-n-hypergraph $H \in L_n^A(G)$, provided that $L_n^A(G) \neq \emptyset$. If G is n-unambiguous and RNG is a uniform random number generator, the hypergraph is chosen uniformly at random.*

Proof. Given a grammar $G = (N, \Sigma, P, S, (mark_p)_{p\in P})$, let $p \in P_N$ be a non-terminal production and $B_kC_{\ell'-k}$ a split for p.

Let $N_{s_j} = B[k] \cdot C[\ell' - k]$ be the number of hypergraphs that can be derived from the jth split and $N_p = (A \xrightarrow{p} BC, i)[\ell]$ be the number of hypergraphs that can be derived from p so that the probability of a split to be chosen is $P(s_j) = N_{s_j}/N_p$, with $\sum_j N_{s_j} = N_p$. In the first method, for choosing the split, a random integer r in the interval $[1, p[\ell]]$ is generated and all the splits are evaluated until the jth split for which $\sum_{1\leq i\leq j} N_{s_i} \geq r$ is reached.

Let h be a Huffman tree for p and a size ℓ where each leaf has its weight $N_{s_j} = B[k] \cdot C[\ell' - k]$, that is, the number of hypergraphs that can be derived choosing the split. Let v_{s_j} be the leaf corresponding to the jth split having weight N_{s_j} and let v_p be the root of h having weight N_p. Each node has a probability to be chosen equal to $N_v/N_{parent(v)}$.

From **Sel** (Alg. 5) we know that the probability of traversing the tree from the root v_p to v_{s_j} is given by the productory of all the probabilities of the nodes in the path $v_p \to v_{s_j}$ having length m to be chosen. Let $v_0 = v_p$ and $v_m = v_{s_j}$, then the overall probability of the path to be traversed is:

$$\prod_{1\leq i\leq m} \frac{N_{v_i}}{N_{v_{i-1}}} = \frac{N_{v_j}}{N_{v_{m-1}}}\frac{N_{v_{m-1}}}{N_{v_{m-2}}} \cdots \frac{N_{v_1}}{N_p} = \frac{N_{v_j}}{N_p} = \frac{B[k] \cdot C[\ell' - k]}{(A \xrightarrow{p} BC, i)[\ell]} \tag{3.3}$$

That corresponds to the choice of the split $B[k] \cdot C[\ell' - k]$.

□

A simple and practical example of this proof is the following: let's consider the first and the last graphs of Figure 3.10. Since in a uniform distribution the order of the elements doesn't affect their probability to be sampled, let's arrange them as they are found in the pre-order visit of the Huffman encoded tree. Thus let's consider the result of the *RNG* to be 8. In the first case we have a sequence of intervals, that is, the frequencies of the elements, of 4, 6 and 12. Going through them we skip the first element because $4 < 8$, so it lies outside the first interval and then we know that 8 falls into the second one for $5 \leq 8 \leq 10$, that is, the sampled element is the split C_5D_1. Now let's consider the same elements arranged in the tree. Navigating from the root, the first choice we encounter is between the intervals $[1, 10]$ and $[11, 22]$. 8 falls into the first interval so we follow the path on the left. The next choice is between the intervals $[1, 4]$ and $[5.10]$. Clearly, as in the previous case, it falls into the second interval, resulting in the choice of the right path, that is, the split C_5D_1.

As we can see, Huffman encoded trees are merely a way to enhance the time for the choice of a split, but they do not alter the probability distribution of the generation.

3.3.6 Time and Space Complexity

The analysis of the complexity of the second method is mostly based on the shifting of part of the computational weight from the generation to the pre-processing phase. Even if at a first glance it may appear that the number of additional operation outweighs its advantages, or may even lean towards longer generation times, we should remember that we suppose that the size of the generated graphs to be sufficiently larger than the grammar describing their language and that the size of the grammar itself is consider as a constant.

Lemma 3.3.1. *The structures needed for the generation of a size-n-hypergraph in the second method can be produced by Algorithms* ***Pre*** *and* ***Enc*** *in $O(n^2)$ time.*

Proof. Lemma 3.2.2 proves that the pre-processing tables for a grammar G and a size n can be produced in linear time $O(n)$. The data needed for the Huffman encoding, that is, the splits and the number of hypergraphs that can be derived from each one of them, can be also retrieved while running the same algorithm. Indeed, during the computation of the number of hypergraphs that can be derived from a non-terminal A, we have to go through all the available splits for a production having A on its *lhs*. Alternatively, we can first produce an equivalent grammar G' input (G', n), with $L(G) = L(G')$, for all the possible splits an then run **Pre** (Alg. 2) to produce them. The table can then be easily condensed again into the standard one for an input (G, n) calculating the sum of all the entries generated for the same referenced production. As for the time complexity of **Enc** (Alg. 4), we consider that for an entry $p[\ell]$ a production has at most $\ell - 1$ splits. Without loss of generality let's consider the case of $n - 1$ splits. To produce the encoded tree the splits are arranged into a graph and a sequence is constructed to order them in their ascending order. This can be done in $O(n \; log \; n)$ time by a simple binary sort algorithm. Adding a new node to the graph and its weight to the sequence can be done in constant time. Joining the first 2 nodes with the least weight as children of the new node and removing their weight from the sequence can also be done in constant time. All the actions involving the resize of both the tree and the sequence are performed exactly $n - 1$ times. Reordering the sequence for a single new value can be done in $O(log \; n)$ time. Since in the worst case we need to produce $n - 1$ trees for each production, considering the size of the grammar as a constant the time complexity of the whole process is bound to be $O(n^2)$. □

Lemma 3.3.2. *The space needed by Algorithm* ***Pre*** *and Algorithm* ***Enc*** *to store the data structures used by the modified generation algorithm for producing a size-n-hypergraph is bounded by $O(n^2)$.*

Proof. Let $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$ be an *HRG* and n the size of the hypergraph to be generated. Lemma 3.2.3 proves that tables M_1 and M_2 resulting from running **Pre** (Alg. 2) on input (G, n) need $O(n)$ space to be stored. Let's consider the worst case in which all non-terminal productions $p \in P_N$ do not produce any internal nodes and can derive all size-ℓ-hypergraphs for $2 \le \ell \le n$. Moreover, let's consider that for each size ℓ, all possible $\ell - 1$ splits are available. We recall that for Proposition 3.3.1 a Huffman tree with k leaves has exactly $2k - 1$ nodes, that is to represent all the trees for all the splits of all productions. Considering both algorithms, the total space needed is:

$$\begin{aligned} &c\sum_{\ell=2}^{n} 2(\ell-1) - 1 + cn = c\sum_{\ell=2}^{n} 2\ell - 3 + cn = 2c\sum_{\ell=2}^{n} \ell - c\sum_{\ell=2}^{n} 3 + cn = \\ &= 2c(\frac{n(n+1)}{2} - 1) - 3c(n-1) + cn = cn^2 + cn - 2c - 3cn + 3c + cn = c(n^2 - n + 1) \end{aligned} \tag{3.4}$$

The result of Equation 3.4 is in $O(n^2)$ for any positive integer $n \in \mathbb{N}$. □

To analyze the complexity of the generation algorithm used in the second method we consider its amortised time. While a thorough explanation of this subject can be found in [11], we stress that such a complexity must not be confused with the "average" time of the execution of an algorithm. The radical difference is in that while the average time is based on the probability of such an algorithm to perform better according to the input or the results of its stochastic processes, the amortized time guarantees the average time of execution by sharing the overall computational weight among all its operations. This is due to the fact that operations are guaranteed to balance each others, that is, if any of them has a longer execution time, it unconditionally results in the shortening of another. The group of operations is so considered and analyzed as a whole, rather than as a single construct.

Theorem 3.3.2. *With the assumptions of Theorem 3.3.1, the size-n-hypergraph H is generated by* ***Gen'*** *in amortized time $O(n)$.*

Proof. Given a grammar $G = (N, \Sigma, P, S, (mark_p)_{p \in P})$, let $p \in P_N$ be a non-terminal production and $B_k C_{\ell' - k}$ a split for p. Let $N_{s_j} = B[k] \cdot C[\ell' - k]$ be the number of hypergraphs that can be derived from the jth split and $N_p = (A \xrightarrow{p} BC, i)[\ell]$.

Algorithm **Gen'**, the modified version of Algorithm 3 uses the Huffman trees constructed in the pre-processing phase to select the splits. In the first method we had to go through all the possible splits of a production and then pick one according to the result of the *RNG*, this clearly required linear time in n, that is, the size of the generated hypergraph. Proposition 3.3.2 states that the height h of a full binary tree, being s the number of its leaves is $\lceil log_2 s \rceil + 1 \le h \le s$, thus, in the worst case, the height of such a tree is equal to the number of leaves. That could easily mislead to consider **Gen'** to have the same asymptotic behavior of its original counterpart. The proof consists instead in showing that even if a single choice may fall in the worst case, it will eventually improve the time needed for choosing the next ones. That is, the computational weight is shared among the choices.

Let's first consider that since a derivation tree for the generation of a size-n-hypergraph has exactly $n - 1$ internal nodes representing the choice of a production, we need $n - 1 + \log_2 A[\ell]$ computational steps to produce the whole hypergraph traversing all the necessary binary trees, taking into account the additional time for choosing the splits. As shown in Algorithm 4, at each step of the encoding the two nodes with the lowest weights are joined together as child of a new node having the weight equal to the sum of both. and then the resulting node is added and reordered in the sequence. In order to have this node picked again, increasing the height of the tree at each step, its weight must be again one of the lowest ones. Without loss of generality, let's consider a sequence of nodes $v_1 \dots v_{\ell'}$ with weights $w_1 = 2^0 \dots w_{\ell'} = 2^{\ell'-1}$. Figure 3.11 shows an example of such an encoding for $\ell' = 7$.

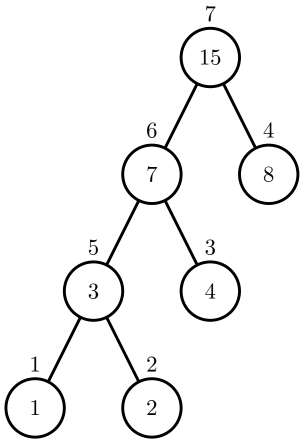


Figure 3.11: Huffman encoding for 4 nodes having weights $2^0 \dots 2^3$

Thus, the height of the Huffman tree for a production p is at most $\lfloor \log_2(N_p/N_{s_j}) \rfloor$ for any jth split. Let's for example consider the boundary case in which $N_p = 2^a$ for some $a > 0$ and $N_{s_j} = 1$. Without loss of generality let's suppose that each non-terminal has at most 1 terminal production and each production does not generate additional nodes, that is $\ell = \ell'$ at each step. Knowing that $N_{s_j} = B[k] \cdot C[\ell' - k]$ for some non-terminal B and C, we can inductively calculate the amortized time t complexity for generating a size ℓ hypergraph from any non-terminal A inductively as follows:

- If $\ell = 1$, we only have to choose a terminal production, then, since we do not need to choose any split, $t = 1 - 1 + \log_2 A[\ell] = 0$.
- Otherwise, $t \le 1 - 1 + \log_2 A[\ell] = 1 + (\lfloor log_2(N_p/N_{s_j}) \rfloor) + (k - 1 + \log_2 B[k]) + (\ell' - k - 1 + \log_2 C[\ell' - k])$.

We may notice that this is independent from the choice of k and, moreover that if the choice of a split results from the travelling of a long path in the Huffman tree, it means that in the next step the choices relative to the non-terminal B and C must be quicker, because there will be less hypergraphs to choose from. Again, considering the boundary case, if we follow the longest path to a split that can generate only 1 hypergraph, trivially the next 2 steps will immediately end.

Finally we should notice that in a more general case, removing the constraint of a single terminal production for each non-terminal, a split $k = 1$ for A, may generate at most a fixed number of graphs. $\square$

Lemma 3.3.3. *The space required by Algorithm **Gen'** for the generation of a size-n-hypergraph is in $O(n)$.*

We omit the proof of the complexity analysis for the generation phase of the second method because it is identical to the one of the first method (Lemma 3.2.4).

Chapter 4

Case Studies

This chapter is dedicated to the analysis of some relevant case studies using sets of simple but significant productions. Differently from string languages, in which the positions of symbols are always arranged in a fixed order from left to right, when dealing with graph languages, as described in chapter 3, we must consider both the labels of the hyperedges and their positions in the graph. For each case, we discuss the reasons for the inherent ambiguity of the language or we provide a non-ambiguous grammar. When discerning the use of multiple labels, without loss of generality, we limit the study to grammars presenting productions having the same structure for their hyperedges on the *rhs*, but having 1 or 2 different labels, being straightforward the extension to 3 or more. It is very important to remember that even if we generate hypergraphs in the slice of a language and such a slice is finite, the choice of the slice happens after the definition of the grammar. So, the analysis of the ambiguity of the following cases is based on non-finite languages.

Since we have already proven the equivalence of the probability distribution of the two sampling methods, we construct the examples using only the first one, being the process clearer to follow.

String graph languages represent a particular case and will be treated in the next chapter.

4.1 Discrete Hypergraph Languages

The discrete hypergraph language is the language of all non-connected graphs composed only by 0-hyperedges, that is, independent edges that are not connected to any node. For its simplest form, the singly labelled, it is possible to construct a non-ambiguous *CNF* grammar to sample from as shown in Figure 4.1.

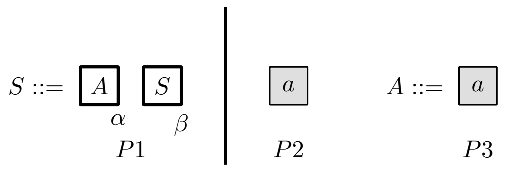


Figure 4.1: Non-ambiguous *CNF* grammar D_1 representing the single labelled discrete hypergraphs language.

From the grammar we can already evince that it doesn't present any structural nor labels related ambiguity. Indeed, the productions do not present any pair of hyperedges that could potentially generate the same subgraph, leading to derivation trees yielding isomorphic hypergraphs. We remind that, despite a shorter non-ambiguous grammar can be found, according to the Chomsky Normal Form defined in 2.4, the *rhs* of a production must have either 1 terminal or 2 non-terminal hyperedges. This is the reason $P1$ has a hyperedge labelled A that can only turn into a terminal a applying the production $P3$.

Table 4.1: Matrices M_1 and M_2 resulting from $\mathbf{Pre}(D_1, 3)$

M_1

N	1	2	3
S	1	1	1
A	1	0	0

M_2

P	1	2	3
$S \xrightarrow{P1} AS, 0$	0	1	1
$S \xrightarrow{P2} a, 0$	1	0	0
$A \xrightarrow{P3} a, 0$	1	0	0

Tables 4.1 show that, correctly, there is only one way to generate a size-3-hypergraph. Inductively we could simply extend these tables for any size $k > 3$, for $P1[k] = A[1]S[k-1]$ and $S[k] = P1[k]$.

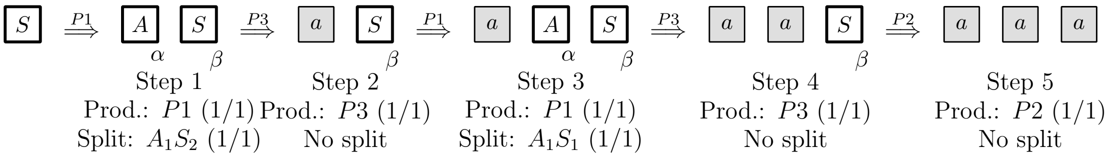


Figure 4.2: Derivation for a size-3-hypergraph using grammar D_1.

The derivation in Figure 4.2 represents the unique way to generate a single labelled size-3-hypergraph uniformly at random with probability 1/1. We may notice, as we did before for the grammar, that none of the sentential forms presents a structural ambiguity. We may conclude that:

Proposition 4.1.1. *For every single labelled discrete hypergraph language there exists a non-ambiguous grammar generating that language.*

Proof. D_1 is a non-ambiguous grammar representing the discrete hypergraph language $L(D_1)$ having $lab(e) = a$ for each hyperedge $e \in E_H$ with $H \in L(D_1)$. □

If we increase the number of labels, we can still construct a non-ambiguous *CNF* grammar represented in Figure 4.3. In this case, it is easy to avoid the ambiguity of generating either a or b from the same non-terminal hyperedge using the common strategy of producing all the as before the bs.

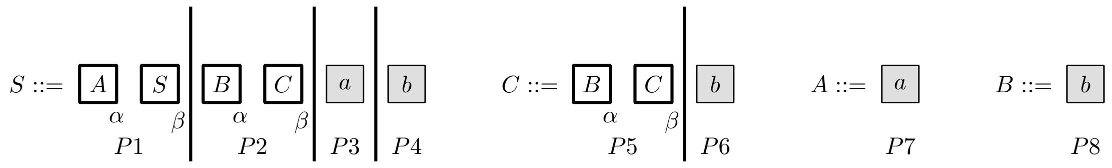


Figure 4.3: Non-ambiguous *CNF* grammar D_2 representing the doubly labelled discrete hypergraphs language.

From Tables 4.2 we may notice that the entry $S[3] = 4$ corresponds to the 4 different combinations of hyperedges labelled with a or b that can be obtained with a size-3-hypergraph. As the previous case, it is possible to see how the table extends for the generation of a size-k-hypergraph with $S[k] = k + 1$.

The derivation in Figure 4.4 shows the unique derivation for the size-3-hypergraph with labels a b b. Such a hypergraph is sampled with probability 1/4 among the 4 possible

Table 4.2: Matrices M_1 and M_2 resulting from $\mathbf{Pre}(D_2, 3)$

M_1

N	1	2	3
S	2	3	4
C	1	1	1
A	1	0	0
B	1	0	0

M_2

P	1	2	3
$S \xrightarrow{P1} AS, 0$	0	2	3
$S \xrightarrow{P2} BC, 0$	0	1	1
$C \xrightarrow{P2} BC, 0$	0	1	1
$S \xrightarrow{P3} a, 0$	1	0	0
$S \xrightarrow{P4} b, 0$	1	0	0
$C \xrightarrow{P6} b, 0$	1	0	0
$A \xrightarrow{P7} a, 0$	1	0	0
$B \xrightarrow{P8} b, 0$	1	0	0

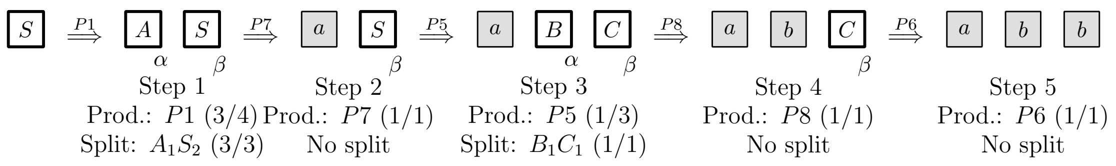


Figure 4.4: Derivation for a size-3-hypergraph using grammar D_2.

combinations of as and bs. Considering the grammar G_{dldh}, without loss of generality, we can express the following proposition for discrete graphs having an arbitrary number of labels:

Proposition 4.1.2. *For every multiple labelled discrete hypergraph language there exists a non-ambiguous grammar generating such a language.*

Proof. As we have extended the grammar D_1 into D_2 by adding the productions for B and C, we could iterate this process to generate a non-ambiguous grammar having any number n of terminal labels. So, without loss of generality, let's consider a language over just two labels a and b. Since D_2 is a non-ambiguous grammar representing the discrete hypergraph language $L(D_2)$ having $lab(e) \in \{a, b\}$ for each hyperedge $e \in E_H$ with $H \in L(D_2)$, so D_n is a non-ambiguous grammar representing the language $L(D_n)$ having $lab(e) \in \{a_1, \ldots, a_n\}$ for each hyperedge $e \in E_H$ with $H \in L(D_n)$. $\square$

We can now apply these proposition to more complex structures and present the following corollary:

Corollary 4.1.1. *Let Comp be a finite set of connected hypergraphs and L be a hypergraph language having members composed by one or more non-connected parts, each part being an element of Comp. Then, there exists a non-ambiguous grammar generating such a language.*

Proof. Let G be a grammar presenting the same structure of non-terminal productions as in D_2 for the sequential derivation of discrete non-terminals $A_1 \ldots A_k$, that avoids the interleaved production of different non-terminals. Let $\{C_1 \ldots C_k\} \subseteq Comp$ be a finite set of non-isomorphic hypergraphs. Adding the productions for each derivation $A_j^\bullet \longrightarrow C_j$ for $1 \leq j \leq k$ to G results in a non-ambiguous grammar. If we consider a sub-tree t' of the derivation tree t up to the production of the non-terminals $A_1 \ldots A_k$, we know from Proposition 4.1.2 that each t' has a different sequence of non-terminals given by its pre-order traversal. If we then extend t' to t, we know that each additional branch is attached to a production having a specific non-terminal on its *lhs*, so their order cannot be changed. We also know that the derivations obtained by following two different branches cannot generate the same graph. That is, there cannot exist two different derivation trees yielding the same non-connected hypergraph. □

As for the string case, we remind that the trivial case of a finite hyperedge replacement language has always a non-ambiguous grammar representing it.

4.2 Star Graph Languages

Corollary 4.1.1 is restricted by the finiteness of the non-connected components. If we consider the star graphs instead, we may move forward to study further implications. By "star" we identify a graph having a set of 1-hyperedges all connected to a single common node. Graph S and S' represent respectively a 4-pointed single labelled star and a 3-pointed multiply labelled star.

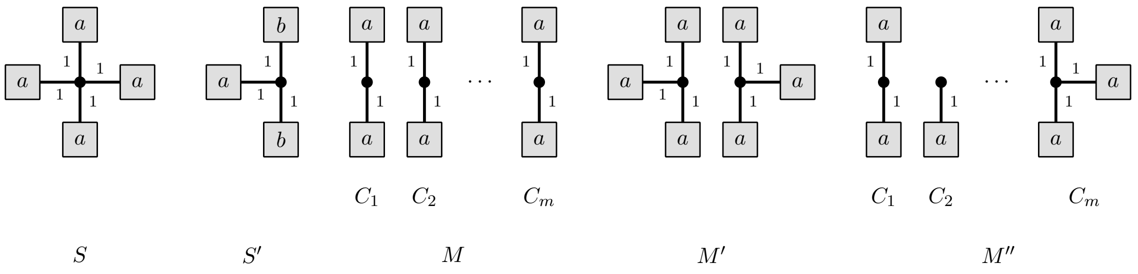


Figure 4.5: Examples of star graphs.

The language of star graphs has non-ambiguous grammars for both its singly and multiply labelled forms. Figures 4.6 and 4.7 represent respectively the former and the latter.

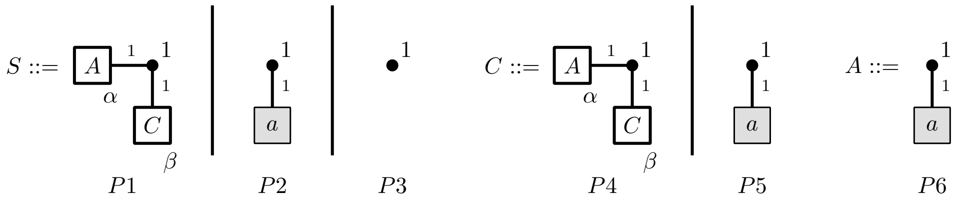


Figure 4.6: Non-ambiguous grammar for the single labelled star graph language.

Let's now consider the language of multiple star graphs, that is, each graph is composed by non-connected stars. Let e be an edge connected to the centre of a star, we use the following notation $(e^x)^y$ to indicate the structure of the hypergraphs in the language, where x is the number of points of a star and y the number of stars in the graph. We use the symbol * to indicate an unbounded quantity. We may discern 5 different cases, according to the settings of the number of starts and their points.

Case 1: $(e^n)^m$. If both the number of points n for each star and the number of stars m in the graph are arbitrary (Graph M in Fig. 4.5) we fall into a problem structurally equivalent to the string language $(a^n b)^m$. That is, an arbitrary number m of substrings all having the same length n separated by a symbol b. Similarly M shows m separated components C_1, C_2, $\ldots C_m$. Intuitively, in the context of hypergraphs, we may consider it as a grid expanding in 2 directions where each column m has exactly n rows. As proven by Engelfriet in [18], this language is not context-free.

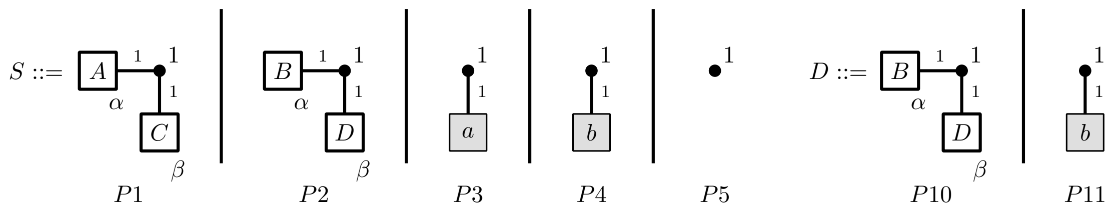
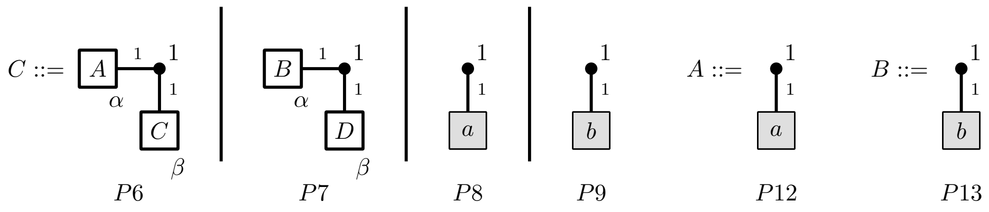



Figure 4.7: Non-ambiguous grammar for the multiple labelled star graph language.

Case 2: $(e^k)^m$. If the number of points for each star is a constant k, while the number of stars m in the graph is arbitrary, we can apply the Corollary 4.1.1, since each non-connected component of the graph is part of a finite set. Figure 4.8 shows a non-ambiguous grammar representing the language for $k = 3$. A member of this language is the graph M' in Figure 4.5. We may notice that production $P1$ allows for any number of 3-pointed stars to be generated.

Proposition 4.2.1. *For every stars graph language $(e^k)^m$, where k is a constant number of points e of each star and m is an arbitrary number of stars, there exists a non-ambiguous grammar generating such a language.*

Proof. $S_{k,m}$ is a non-ambiguous grammar representing the stars graph language $L(S_{k,m})$ having $k = 3$. □

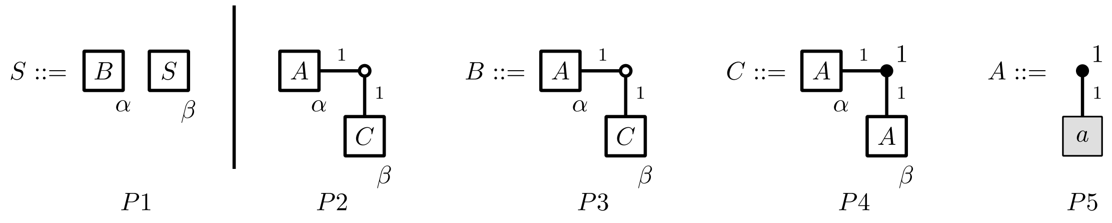


Figure 4.8: Non-ambiguous grammar $S_{k,m}$ for the $(e^k)^m$ star graph language with $k = 3$.

Case 3: $(e^n)^k$. If the number of points n for each star is arbitrary, while the number of stars is constant k, there exists a non-ambiguous grammar representing the language. We can indeed connect a k-hyperedge to the central node of each star at once and generate the same number of points. Figure 4.9 shows a non-ambiguous grammar representing the language for $k = 2$. The graph M' is a member of this language. We may notice that iterating $P3$ and $P5$ we may generate 2 stars with an arbitrary number of points.

Proposition 4.2.2. *For every stars graph language $(e^n)^k$, where n is an arbitrary number of points e of each star and k is a constant number of stars, there exists a non-ambiguous grammar generating such a language.*

Proof. $S_{n,k}$ is a non-ambiguous grammar representing the stars graph language $L(S_{n,k})$ having $k = 2$. □

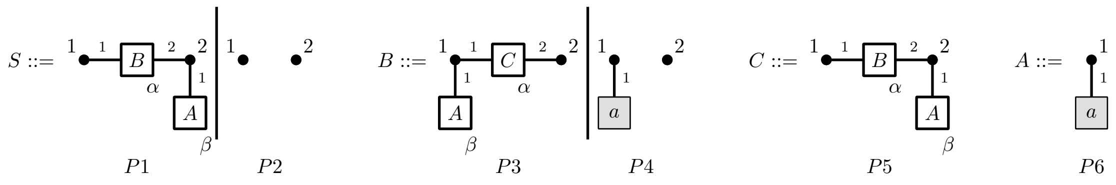


Figure 4.9: Non-ambiguous grammar $S_{n,k}$ for the $(e^n)^k$ star graph language with $k = 2$.

Case 4: $(e^n)^*$. If the number of points n is arbitrary and the number of stars is unbounded, the language is again non context-free. Intuitively, as the first case, we need to generate non-connected identical components. A grammar as in Figure 4.9 would not work because the type of the shared hyperedges adding the same amount of points to each star should be also unbounded. That is, the hyperedges should be untyped.

Case 5: $(e^*)^m$. If the number of points is unbounded, but the number of stars m in the graph is arbitrary, such a language is inherently ambiguous. This is the most interesting case as it represents a first example of structural ambiguity. Graph M'' in Figure 4.5 is a member of this language. Starting from the same approach used for the grammar in Figure 4.8 for the case $(e^k)^m$ we generate a sequence of isolated non-terminal hyperedges, then, from any of them, we generate a star with an unbounded number of points.

Proposition 4.2.3. *The stars graph language $(e^*)^m$, where $*$ is an unbounded number of points e of each star and m is an arbitrary number of stars is inherently ambiguous.*

Proof. These graphs are composed by an arbitrary number of stars each of them having an unbounded number of points. We can consider them as unconnected components of non-finite size, since each of them can have any number of edges. It is not possible to generate all components from a single connected non-terminal hyperedge, since it would require an arbitrary type m-hyperedge and by definition all hyperedges have a fixed type. Let t be the maximum type for a hyperedge, then there are exactly $\lceil m/t \rceil$ unconnected components. Thus, without loss of generality, let's consider each component $C_1, \ldots, C_m$ to be generated by a 0-hyperedge $e_1, \ldots, e_m$. Since each component has an unbounded number of edges $e_1, \ldots, e_m$ are pairwise symmetric. Let S_1, S_2 be star graphs with $|S_1| \neq |S_2|$ and let $m = 2$ be the number of stars. For any context-free grammar representing $L(S_{*,m})$ it is always possible to obtain two different derivations where $C_1 = S_1$ and $C_2 = S_2$ or $C_1 = S_2$ and $C_2 = S_1$. That is, the language $L(S_{*,m})$ is inherently ambiguous. $\square$

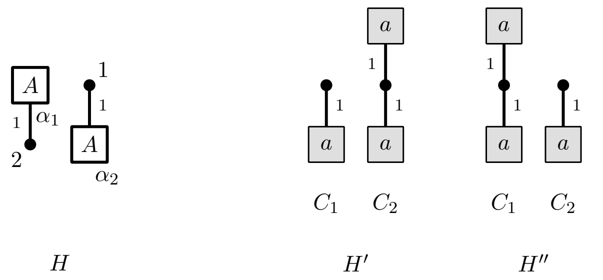


Figure 4.10: Symmetric hyperedges along with the derived isomorphic size-5-hypergraphs for the $(e^*)^m$ star graph language with $m = 2$.

Figure 4.10 shows an example of derived isomorphic graphs $H' \cong H''$ in the language $L(S_{*,m})$ for $m = 2$ where the components C_1 and C_2 are a single and a double pointed star graphs. Such an ambiguity is caused by the structural symmetry of the hypergraph H.

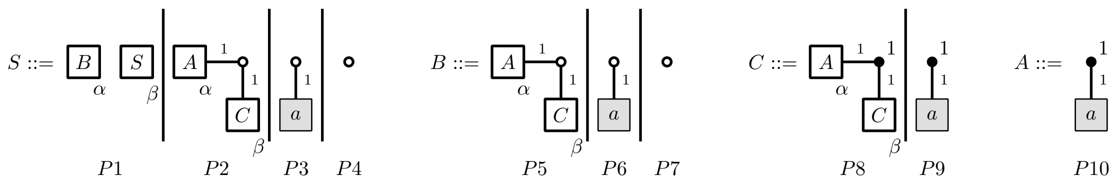


Figure 4.11: Ambiguous grammar $S_{*,m}$ for the $(e^*)^m$ star graph language.

Table 4.3 show the result of algorithm **Pre** on input $(S_{*,m}, 3)$. We may notice that the entry $S[3] = 4$. There are indeed 4 hypergraphs that can be generated from the starting symbol, but, 2 of them are isomorphic, as shown in Figure 4.12. That proves the ambiguity of the grammar.

Table 4.3: Matrices M_1 and M_2 resulting from $\mathbf{Pre}(S_{*,m}, 3)$

M_1

N	1	2	3
S	1	2	4
B	1	1	1
C	1	1	1
A	1	0	0

M_2

P	1	2	3
$S \xrightarrow{P1} BS, 0$	0	1	3
$S \xrightarrow{P2} AC, 1$	0	0	1
$B \xrightarrow{P5} AC, 1$	0	0	1
$C \xrightarrow{P8} AC, 0$	0	1	1
$S \xrightarrow{P3} a, 1$	0	1	0
$S \xrightarrow{P4} \lambda, 1$	1	0	0
$B \xrightarrow{P6} a, 1$	0	1	0
$B \xrightarrow{P7} \lambda, 1$	1	0	0
$C \xrightarrow{P9} a, 0$	1	0	0
$A \xrightarrow{P10} a, 0$	1	0	0

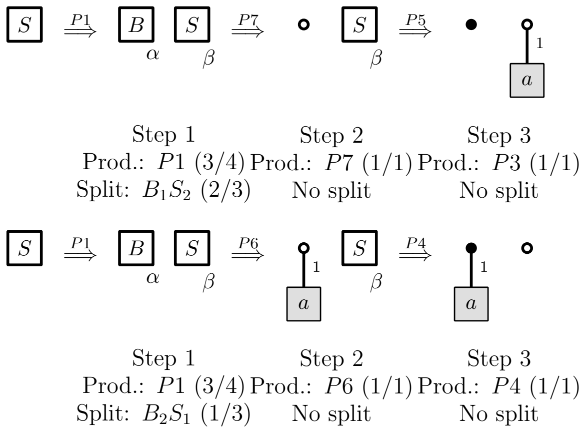


Figure 4.12: Derivations of a size-3-hypergraph showing the ambiguity of $S_{*,m}$ grammar.

Generalising these concepts to a wider class of graphs we can present the following Corollary on the inherent ambiguity of languages of which members are made of an arbitrary number of unconnected parts, each part being an element of an infinite set:

Corollary 4.2.1. *Let Comp be an infinite set of connected hypergraphs and L be a hypergraph language having members composed by one or more non-connected parts, each part being an element of Comp. Then, L is inherently ambiguous.*

Proof. The proof comes from the generalisation of Proposition 4.2.3 to components that are elements of an infinite set of connected graphs. □

4.3 Cycle Graph Languages

The Cycle Graph Languages are the languages of hypergraphs composed by closed chains of 2-hyperedges. To study the characteristics of these languages we consider all the hyperedges oriented in the same direction. That is, if a pair of them shares a node, it is the first attachment node for one and the second for the other. Formally, if $att(e)_i = att(e')_j$ then $i \neq j$.

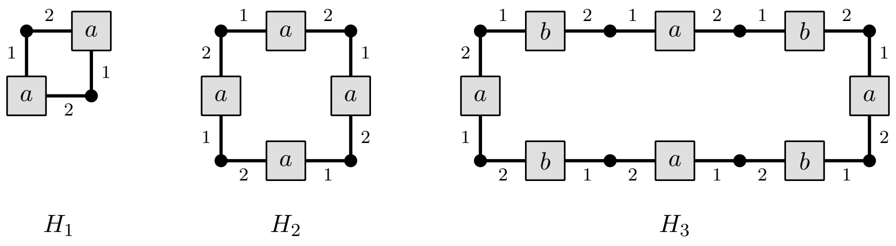


Figure 4.13: Example of cycle graphs.

Figure 4.13 shows 3 examples of cycle graphs. H_1 is a simple cycle composed by 2 hyperedges. H_2 is a singly labelled cycle while H_3 is a multiply labelled cycle. As the previous case, we limit the number of labels to 2. The structural ambiguity of the cyclic component is slightly different from the previous case. While, as a subgraph, a cycle can potentially lead to a structural ambiguity, in this particular case we are interested in the ambiguity derived from the number of labels used.

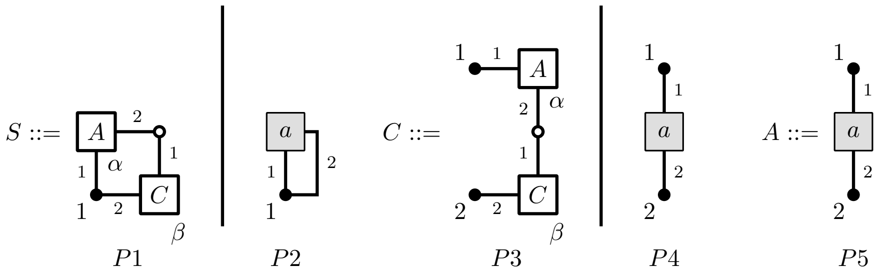


Figure 4.14: Non-ambiguous grammar C_1 for the single labelled cycle graph language.

Figure 4.14 shows a non-ambiguous grammar for the singly labelled cycle graph. This grammar is conceptually equivalent to the one representing an open chain of hyperedges, or, in the string setting, a grammar resolving the ambiguity for the regular language a^*. Indeed, the graph develops only in a single direction.

Proposition 4.3.1. *For every singly labelled cycle hypergraph language there exists a non-ambiguous grammar generating that language.*

Proof. C_1 is a non-ambiguous grammar representing the cycle hypergraph language $L(C_1)$ having $lab(e) = a$ for each hyperedge $e \in E_H$ with $H \in L(C_1)$. □

Figure 4.15 shows an ambiguous grammar for the multiply labelled cycle graph. Even if apparently the difference from its singly labelled counterpart resides only in the doubling of the productions for the b labelled terminal hyperedges, the circular structure of the graphs generates an unavoidable symmetry. Intuitively, we may think about these graphs as an unbounded number of blocks a^*b^* given in any order, since their cyclic nature.

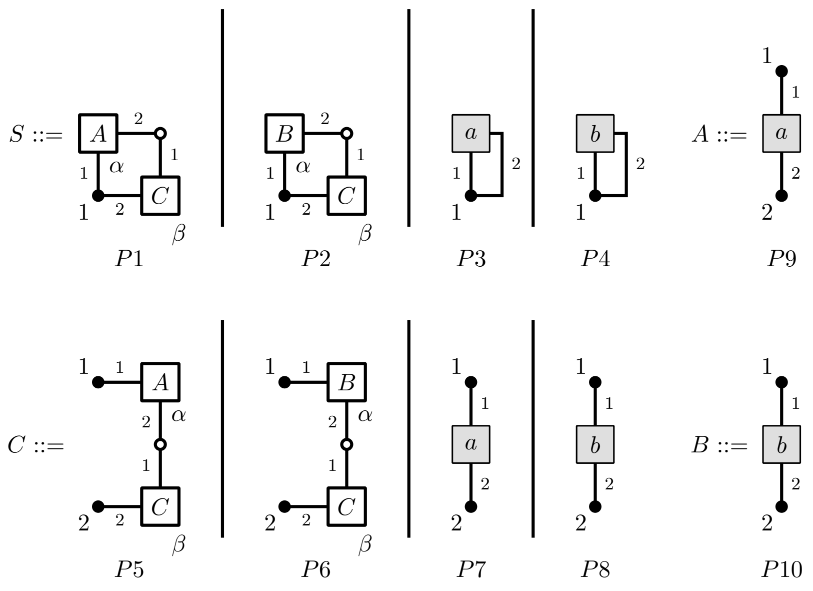


Figure 4.15: Ambiguous grammar C_2 for the multiply labelled cycle graph language.

Proposition 4.3.2. *The multiply labelled cycle graph language is inherently ambiguous.*

Proof. Let a and b be the two labels of a multiply labelled cycle graph language and let's consider the following cases:

- All non-terminal hyperedges are produced before being replaced by terminal as and bs. Suppose that they are replaced in an arbitrary order. Since the cycle has an arbitrary number of edges, keeping track of their terminal replacements would require an untyped non-terminal hyperedge. This case presents the same problem of the $(e^*)^m$ stars graph language and, equivalently, it is not solvable for a non-ambiguous grammar.

- The non-terminal hyperedges are produced and replaced without any specific order. That is, at some point of the derivation two non-terminal hyperedges may be separated by one or more terminals and still having an infinite generative power. That's structurally equivalent of having two unconnected components. For Corollary 4.2.1 such a grammar is ambiguous. The minimal structure causing the ambiguity is the graph H in Figure 4.16.

- The non-terminals are produced and replaced in a circular order as described in the grammar in Figure 4.15. Even if this grammar is able to produce unique sequences of terminals, we should remember that the underlying structure is a circle, not a line. That is, for a cycle with n hyperedges, there are at most n ways to produce the same hypergraph, the worst case being the one with all as but one or viceversa.

Since there is no arrangement in the production of labels or the relative hyperedges that can allow for a non-ambiguous grammar, the language is inherently ambiguous. □

We may notice that a non-ambiguous grammar, as the one in Figure 4.14, for the singly labelled cycle graphs is possible because in the third case producing an ordered circle of terminals has a unique sequence.

Figure 4.16 shows an example of derived isomorphic size-8-hypergraphs $H' \cong H''$ in the multiply labelled cycle graph language. Such an ambiguity is caused by the structural symmetry of the hypergraph H.

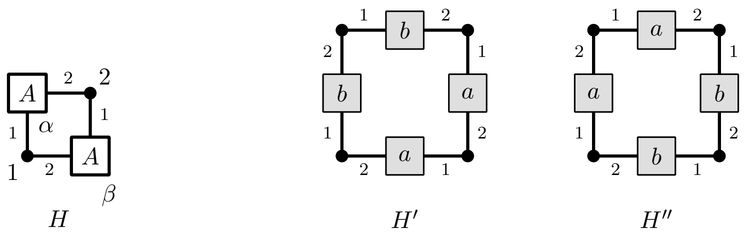


Figure 4.16: Symmetric hyperedges along two possible isomorphic size-8-hypergraphs for the multiply labelled cycle graph language with labels a and b.

Table 4.4 show the result of algorithm **Pre** on input $(C_2, 3)$. We may notice that the entry $S[3] = 4$. Since the starting handle is a 1-hypergraph we may generate the table for a size of $n - 1$. There are indeed 4 hypergraphs that can be generated from the starting symbol, but, 2 of them are isomorphic, as shown in Figure 4.17. That proves the ambiguity of the grammar.

Table 4.4: Matrices M_1 and M_2 resulting from **Pre**$(C_2, 3)$

M_1

N	1	2	3
S	2	0	4
C	2	0	4
B	1	0	0
A	1	0	0

M_2

P	1	2	3
$S \xrightarrow{P1} AC, 1$	0	0	2
$S \xrightarrow{P2} BC, 1$	0	0	2
$C \xrightarrow{P5} AC, 1$	0	0	2
$C \xrightarrow{P6} BC, 1$	0	0	2
$S \xrightarrow{P3} a, 0$	1	0	0
$S \xrightarrow{P4} b, 0$	1	0	0
$C \xrightarrow{P3} a, 0$	1	0	0
$C \xrightarrow{P4} b, 0$	1	0	0
$A \xrightarrow{P9} a, 0$	1	0	0
$B \xrightarrow{P10} b, 0$	1	0	0

In a more general case, the same approach can be used to analyze the ambiguity of a cycle language in which the hyperedges can be oriented in both ways. Even for the singly labelled case, this language is still inherently ambiguous. The ambiguity in fact derives by the circular structure itself, potentially generating an infinite number of components, rather than them being distinguished by different labels or subgraphs.

4.4 Tree Languages

There are several ways to represent trees using hypergraphs, for example using 3-hyperedges for representing branching in binary trees where the 3 attachment nodes respectively represent the parent, left child and right child. Despite being very effective, this approach hides the ambiguity problem we want to study. The advantage of hypergraphs to naturally describe the order of their connected elements would also enforce an ordering for the children of a parent node. That's the reason we prefer to use the more classic approach of using terminal 2-hyperedges to represent connections of parent nodes to their children. Again we distinguish between singly labelled trees and multiply labelled trees, but with an additional restriction: the uniqueness of the labelling is extended to the edges connecting the children of a node. It means that in the multiply labelled case a node cannot have

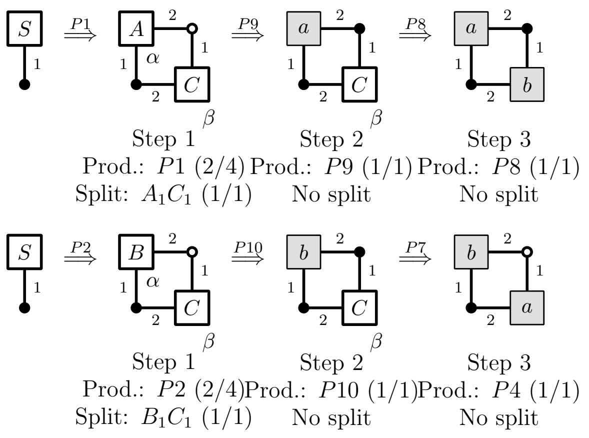


Figure 4.17: Derivations of a size-4-hypergraph showing the ambiguity of C_2 grammar.

children connected with hyperedges having the same label, otherwise, we would fall into the same singly labelled case while considering the sub-tree of that node.

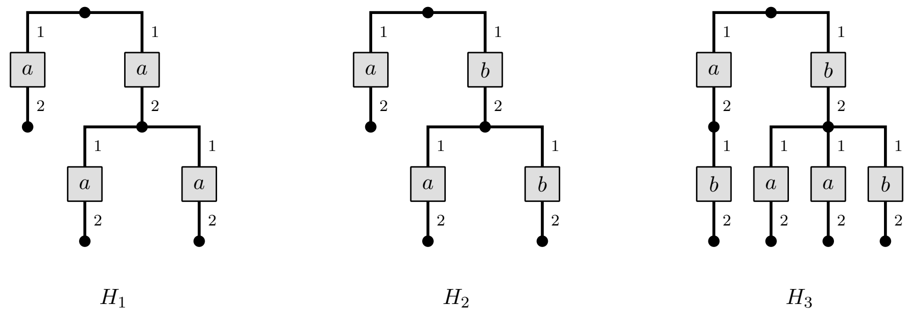


Figure 4.18: Example of tree graphs.

The graphs H_1 and H_2 in Figure 4.18 are respectively a singly labelled and a multiply labelled binary trees. Even if H_3 is also a multiply labelled binary tree we may notice that on one side there are two children both connected with a labelled edges. If we consider the sub-tree composed by those two edges, this structure presents the same problem as H_1. So, to avoid any confusion, without loss of generality, we consider multiply labelled trees as ordered full binary trees on the set of terminal labels $\{a, b\}$ where a symbolically represents the connection between the parent and the left child and b the connection to the right one. Interestingly, these languages present an opposite behavior to the one observed for cycle graphs.

Figure 4.15 shows a non-ambiguous grammar for generating ordered full binary trees. We may notice that $P8$ and $P11$ represent productions with potentially symmetric behavior. On a closer look, we may notice instead that in the sentential forms, every time there are seemingly symmetric hyperedges, they belong to two different branches of the tree as in the graph H_2 in Figure 4.18. Since such a tree is ordered there are never two identical paths from the root to a node. We have already analyzed this structure when defining the Huffman codes in 3.3.1. Alternatively we may formulate this concept saying that none of the paths leading to any of the hyperedges in the tree is a prefix of another. That is, the only way to expand the tree is to create a new path. The result is that no one of the hyperedge can be in a potentially symmetric position to any other. If there was such a configuration, there should be a parent node with 2 children with the same label, as in graph H_3 of Figure 4.18, but that's impossible, because the tree is fully ordered. So, even the hyperedges in $P8$ and $P11$ have the same generating power, the language doesn't

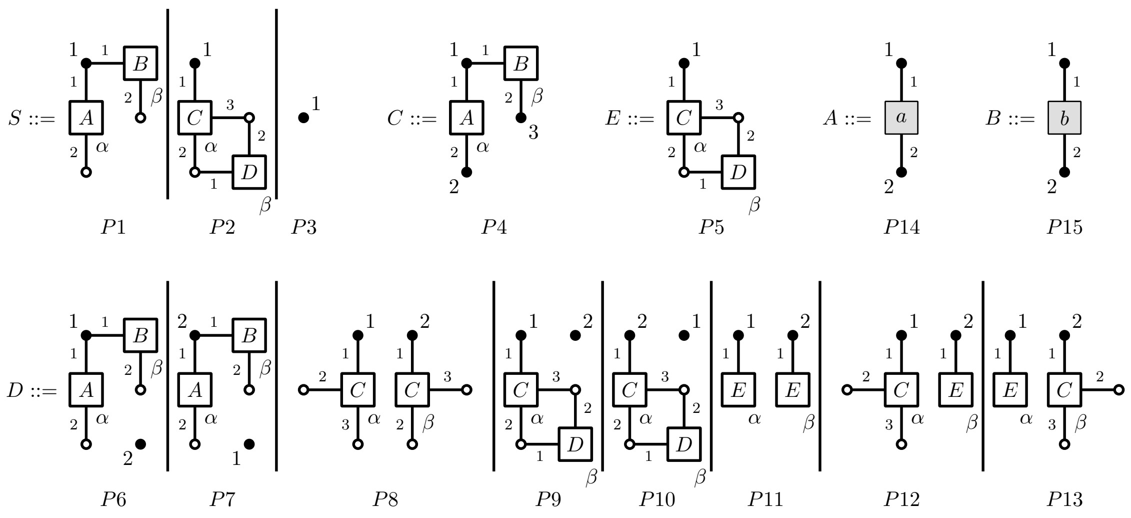


Figure 4.19: Non-ambiguous grammar T_2 for the ordered full binary tree language.

present any structural ambiguity.

Proposition 4.4.1. *For every ordered full binary tree language there exists an non-ambiguous grammar generating that language.*

Proof. T_2 is a non-ambiguous grammar representing the ordered full binary tree language $L(T_2)$ having for each pair of edges e, e' connecting a parent node to its left and right children, $lab(e) = a$ and $lab(e') = b$. $\square$

If we reduce the number of labels to 1 then the singly labelled tree language obtained is instead inherently ambiguous. Intuitively we may notice that losing the ordering, makes the children of a node indistinguishable. We may notice that productions $P8$ and $P11$ contain symmetric hyperedges.

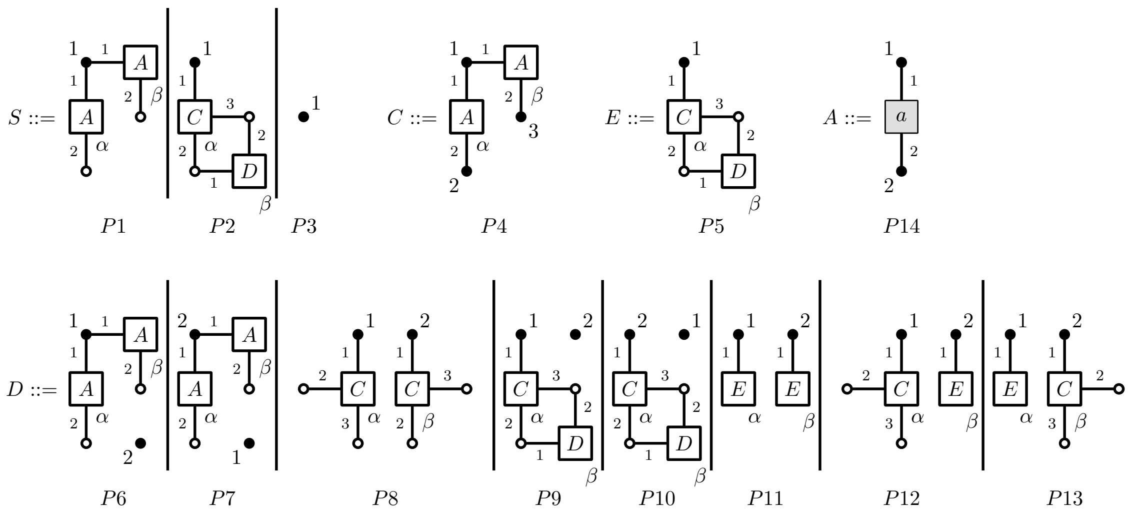


Figure 4.20: Ambiguous grammar T_1 for the singly labelled binary tree language.

Proposition 4.4.2. *The singly labelled tree language is inherently ambiguous.*

Proof. Without loss of generality, let's consider a singly labelled full binary tree T. Let a be the label. Given h as the height of T, we know from Proposition 3.3.2 that the number of its leaves is at most 2^{h-1}. So generating such a tree sequentially from the root using a single hyperedge would require an exponential increase of its type at each step, which is not feasible due to its fixed type.

Let n, n' be two leaves of T and let n_p be their parent. Let e, e' be a pair of edges, so that $lab(e) = lab(e')$ and $att(e)_1 = att(e')_1 = n_p$. Let $att(e)_2 = n$ and $att(e')_2 = n'$. Since $\Pi^e_T = \Pi^{e'}_T$, e and e' form a structural ambiguity and so the singly labelled full binary tree language is inherently ambiguous. $\square$

Two children is the minimum indeed to produce a structural ambiguity. This proof can easily be extended to trees allowing for any number of children, besides the trivial case of the language of trees having only 1 child per parent. In such a language the hyperedges are arranged in a chain and it can be easily categorized as a string graph language rather than a tree. Despite the possibility of a string graph language to still be inherently ambiguous, the ambiguity of the grammars representing it are not directly caused by the structure of their members, but rather to the limits of context-freeness of *HRG* as shown in Chapter 5.

Table 4.5: Matrices M_1 and M_2 resulting from **Pre**$(T_1, 8)$

M_1

N	1	2	3	4	5	6	7	8
S	0	0	0	1	0	0	0	2
C	0	1	0	0	0	0	0	0
E	0	0	0	0	0	0	0	2
D	0	0	0	2	0	0	0	4
A	1	0	0	0	0	0	0	0

M_2

P	1	2	3	4	5	6	7	8
$S \xrightarrow{P1} AA, 2$	0	0	0	1	0	0	0	0
$S \xrightarrow{P2} CD, 2$	0	0	0	0	0	0	0	2
$C \xrightarrow{P4} AA, 0$	0	1	0	0	0	0	0	0
$E \xrightarrow{P5} CD, 2$	0	0	0	0	0	0	0	2
$S \xrightarrow{P3} \lambda, 0$	1	0	0	0	0	0	0	0
$D \xrightarrow{P6} AA, 2$	0	0	0	1	0	0	0	0
$D \xrightarrow{P7} AA, 2$	0	0	0	1	0	0	0	0
$D \xrightarrow{P8} CC, 4$	0	0	0	0	0	0	0	0
$D \xrightarrow{P9} CD, 2$	0	0	0	0	0	0	0	2
$D \xrightarrow{P10} CD, 2$	0	0	0	0	0	0	0	2
$D \xrightarrow{P11} EE, 0$	0	0	0	0	0	0	0	0
$D \xrightarrow{P12} CE, 2$	0	0	0	0	0	0	0	0
$D \xrightarrow{P13} EC, 2$	0	0	0	0	0	0	0	0
$A \xrightarrow{P14} a, 0$	1	0	0	0	0	0	0	0

Table 4.5 shows the result of the Pre-processing phase using grammar T_1 for the singly labelled tree case. Since the starting handle is a 1-hypergraph we may generate the table for a size of $n - 1$. We may notice from entry $S[8]$ that there are 2 possible derivations, shown in Figure 4.22, yielding the isomorphic size-9-hypergraphs H' and H'' of Figure 4.20. Hypergraph H shows the minimal structure causing such an ambiguity.

4.5 Series Parallel Languages

The last case we propose is the singly labelled series-parallel graph language [20], commonly used for describing *RLC* electrical circuits. Such a language presents an interesting structure deriving from the composition of two simple languages. The series component is a chain of edges, equivalent to a string graph, for which, we have proven that there exists a non-ambiguous grammar (Chapter 5). The other component is structurally equivalent to a singly labelled star graph, with the only difference that the hyperedges share 2 nodes

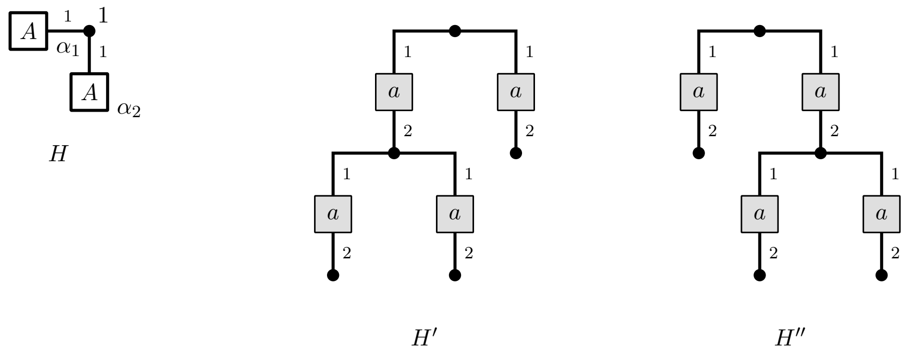


Figure 4.21: Symmetric hyperedges along with the derived isomorphic size-9-hypergraphs for the singly labelled full binary tree language

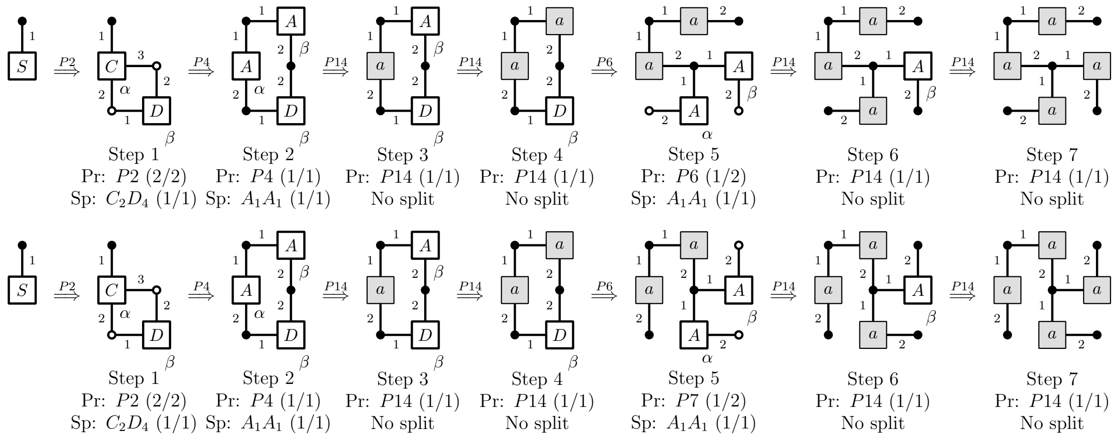


Figure 4.22: Derivations of a size-9-hypergraph showing the ambiguity of T_1 grammar

instead of 1. It has also already proven to allow a non-ambiguous grammar in Section 4.2. When we combine the two, we obtain a language where a non-terminal hyperedge can either indefinitely expand creating a new parallel circuit or adding more edges to the same line.

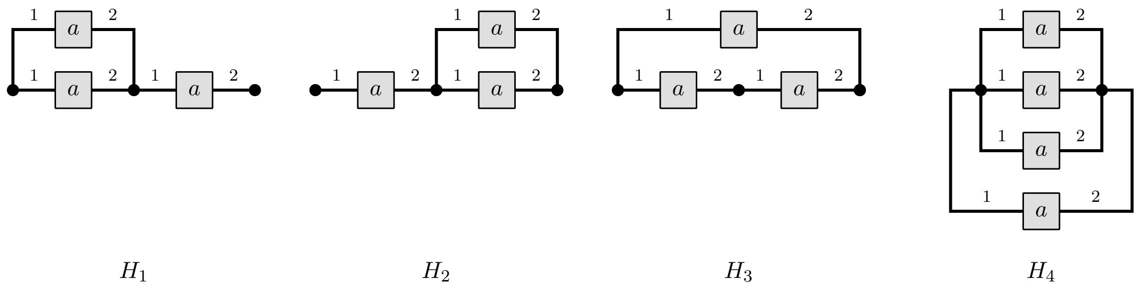


Figure 4.23: Example of series-parallel graphs

Similarly to the star graph languages we may associate a single parallel component a $(e^*)^*$ graph, being e^* the unbounded number of consecutive edges on a line for * parallel lines. Intuitively we may already understand that since the number of parallel lines is unbounded there is no typed hypergraph from which they can be generated unambiguously. Consequently, a grammar generating them non-ambiguously cannot exist.

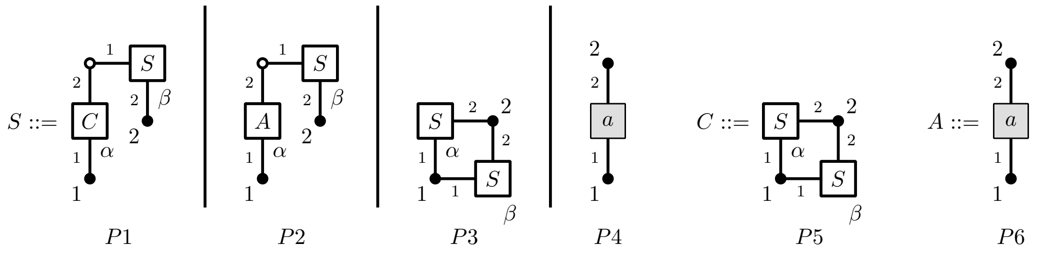


Figure 4.24: Example of an ambiguous grammar SP_1 for the series-parallel graphs

Figure 4.24 shows an ambigous grammar for the series-parallel graph language. This grammar is ambiguous because productions $P3$ and $P5$ contain symmetric hyperedges. We may notice how another potential source how ambiguity has been solved for the chain components in productions $P1$ and $P2$ with the addition of productions $P5$ and $P6$.

Proposition 4.5.1. *The series-parallel graph language is inherently ambiguous.*

Proof. Let π be a subgraph representing a parallel component of a series-parallel graph H, between nodes n_1, n_2 so that for each $e \in \pi$ $att(e)_1 = n_1$ and $att(e)_2 = n_2$. Since the number of edges in π is unbounded it is not possible to generate all of them in a single replacement with a fixed type hyperedge.

Without loss of generality let $e, e' \in \pi$ be two parallel hyperedges, so that $lab(e) = lab(e')$. Since $\Pi_T^e = \Pi_T^{e'}$ and the replacements of e and e' are in two separated contexts, the hyperedges form a structural ambiguity and so the series-parallel language is inherently ambiguous. □

Table 4.6 shows the result of the Pre-processing phase using grammar SP_1. Since the starting handle is a 2-hypergraph we may generate the table for a size of $n - 2$. We may notice from entry $S[4]$ that there are 11 ways to generate a size-6-hypergraph. That highlights the ambiguity of the grammar, since the only series-parallel size-6-hypergraph are the 4 graphs in Figure 4.23.

Table 4.6: Matrices M_1 and M_2 resulting from $\mathbf{Pre}(SP_1, 4)$

M_1

N	1	2	3	4
S	1	1	3	11
C	0	1	2	7
A	1	0	0	0

M_2

P	1	2	3	4
$S \xrightarrow{P1} CS, 1$	0	0	0	1
$S \xrightarrow{P2} AS, 1$	0	0	1	3
$S \xrightarrow{P3} SS, 0$	0	1	2	7
$C \xrightarrow{P5} SS, 0$	0	1	2	7
$S \xrightarrow{P4} a, 0$	1	0	0	0
$A \xrightarrow{P6} a, 0$	1	0	0	0

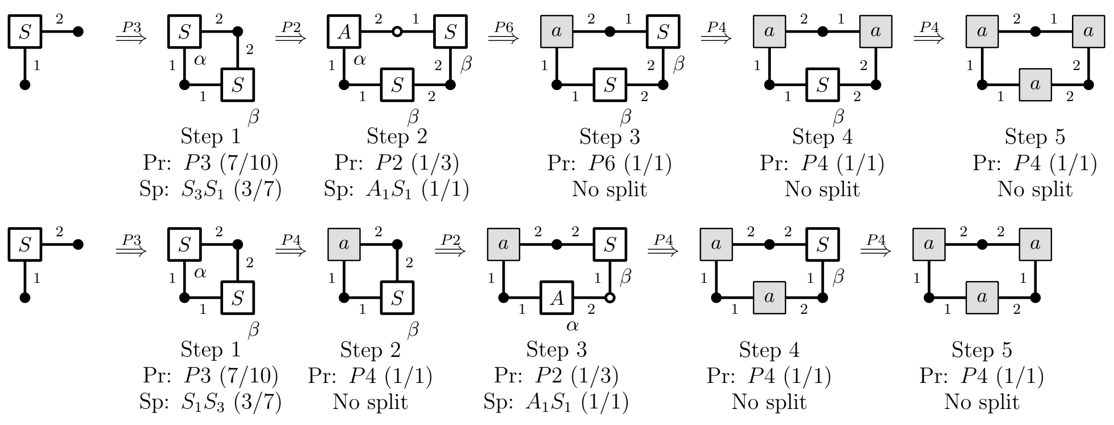


Figure 4.25: Derivations of a size-6-hypergraph showing the ambiguity of SP_1 grammar

Series-parallel graphs have a source and a target, the former being a node allowing only connections via the first attachment point of edges and the latter allowing only connections from the second attachment point. If we instead consider closed series-parallel graphs, where the source and target are merged, we can give a simple proof of inherent ambiguity of such languages by reducing them to multiply labelled cycle languages. Identifying the components of the graph is outside the scope of this work, since an algorithm has already been given in [20].

Proposition 4.5.2. *The closed series-parallel graph language is inherently ambiguous.*

Proof. Let C be a closed series-parallel graph with a single edge followed by a parallel component of 2 edges. Let $lab(e) = a$ for each $e \in E_C$. We reduce the parallel component by replacing it with a single hyperedge e' having $lab(e') = b$. Such a graph is a multiply labelled cycle graph also representing a minimal case of structural ambiguity. From Proposition 4.5.1 we know that every component can be ambiguously expanded from this structure to generate any other member of the language. That is, the language is inherently ambiguous. □

The reduction from a more complex language to one of the cases analyzed is a very practical way to identify inherently ambiguous languages. For example, this is the case of string diagrams, the language of which can now easily be proven inherently ambiguous due to their parallel component.

Chapter 5

String Graph Languages

String graphs are a particular form of hypergraphs shaped as a chain of hyperedges that can be used to represent literal strings [19] with a direct correspondence between their labels and the symbols of the represented word. Mairson's methods, described in Section 3.1, have been originally developed to work with classic context-free string grammars, but, since we have extended the algorithm to the settings of hypergraphs, it is worth to explore the advantages acquired by generating words from graph grammars. The aim of this chapter is to increase the range of string languages to sample from and, when possible, offer a uniform distribution of the sampling. We stress again the importance of a uniform distribution as a sought result or even a requirement for many applications relying on random generation. One limitation of Mairson's methods is that they indeed require a context-free grammar to work with. Moreover, to obtain a uniform distribution such a grammar must be non-ambiguous. Nevertheless there are several cases in which we may overcome these limitations representing string languages with hyperedge replacement grammars. In [19] Engelfriet and Heyker show that it is possible to represent some non-context-sensitive-free string grammars with context-free hyperedge replacement grammars. For example, the non-context-free language $L = \{ww \mid w \in \{a,b\}^*\}$ may be represented by the string graph grammar in Figure 5.3.

Considering strings as sequences of symbols, they are easily representable as chains of 2-hyperedges with a consistent order of their attachment nodes, each label corresponding to a different symbol.

Definition 5.0.1 (String Graph)**.** Let $s = \sigma_1 ... \sigma_n$ be a string and let H be a hypergraph having $V_H = \{v_0\}$ and $E_H = \emptyset$, then a *string graph* representing s, also denoted as $s\bullet$, is constructed as follows:

For each $\sigma_k \in s$ with $1 \leq k \leq n$:

1. $V_H = V_H \cup \{v_k\}$
2. $E_H = E_H \cup \{e_k\}$ with $\mathit{type}(e_k) = 2$, $\mathit{att}(e)_1 = v_{k-1}$, $\mathit{att}(e)_2 = v_k$ and $\mathit{lab}(e) = \sigma_k$

Clearly the empty string λ is represented by a hypergraph containing only a single node. If $H' \cong s\bullet$, then H' is also a graph representation of s. We write $(\sigma_1 ... \sigma_n)\bullet$ for a string graph representing the string $\sigma_1 ... \sigma_n$. We always consider the beginning of the string to be the label of the hypeperge which first attachment node has no other connections.

Figure 5.1 shows the string graphs corresponding to the empty string λ, the single character a and a the string $aabaab$.

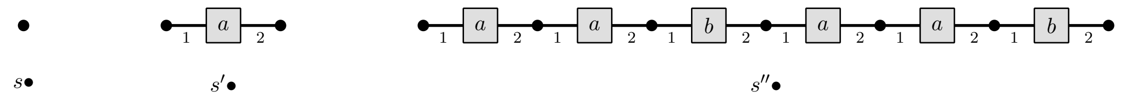


Figure 5.1: Example of string graphs

5.1 Context-Free String Grammars

While a string graph language is simply a language containing only string graphs, there is no restriction on the production of a string graph grammar to only have string graphs on their *rhs*. The only requirement is to produce the graphs in the language. In order to translate a string grammar directly into the setting of hypergraphs we need to remove the empty productions. In contexts where the merging of nodes after the replacement is considered [18], such a problem does not arise, because we can automatically join two consecutive hyperedges. Instead, dealing with this problem without merging the nodes results in several additional productions without producing any real advantages. So, the easiest way to translate a string grammar into the setting of hypergraphs is to first transform it into a known form without empty productions. The only exception being for the starting symbol if the empty string is included in the language. Among all the possible choices, the Chomsky normal form is the most suitable, because it directly translates into a grammar ready for the sampling. We propose a simple algorithm to prove the equivalence between string grammars and their hyperedge replacement coutnerpart. Since the process for transforming a context-free string grammar into a Chomsky normal form is well known [9], we suppose that the input of the algorithm is already a string grammar in *CNF*.

Proposition 5.1.1. *For every context-free string grammar G_S there exists a hyperedge replacement grammar G_H in CNF such that $L(G_H) = \{s\bullet \mid s \in L(G_S)\}$.*

Proof. Let $G_S = (N, T, P, S)$ be a context-free string grammar in *CNF*, then an equivalent *HRG* $G_H = (N, T, P', S, (mark_p)_{p \in P'})$ in *CNF* can be built such that for each $p \in P$ where $rhs(p) \neq \lambda$:

- if $p = A \longrightarrow BC$ then $P' = P' \cup \{p'\}$ where $lhs(p') = A$ and $rhs(p') = rhs(p)\bullet$ having $ext(rhs(p')) = (0, 2)$.
- if $p = A \longrightarrow a$ then $P' = P' \cup \{p'\}$ where $lhs(p') = A$ and $rhs(p') = rhs(p)\bullet$ having $ext(rhs(p')) = (0, 1)$.

if $\lambda \in L(G_S)$ then:

- $P' = P' \cup \{p'\}$ where $lhs(p') = S$, $E_{rhs(p')} = \emptyset$ and $|V_{rhs(p')}| = 1$.
- for each $p \in P^S$ $ext(rhs(p')) = \lambda$.

□

Let's consider the following string grammar in Chomsky normal form representing the language $\{a^n b^n, n \geq 0\}$:

$$
\begin{aligned}
S &::= AC \mid \lambda \\
C &::= DB \mid b \\
D &::= AC \\
A &::= a \\
B &::= b
\end{aligned}
$$

According to Proposition 5.1.1 it can be directly translated into the hyperedge replacement grammar in Figure 5.2.

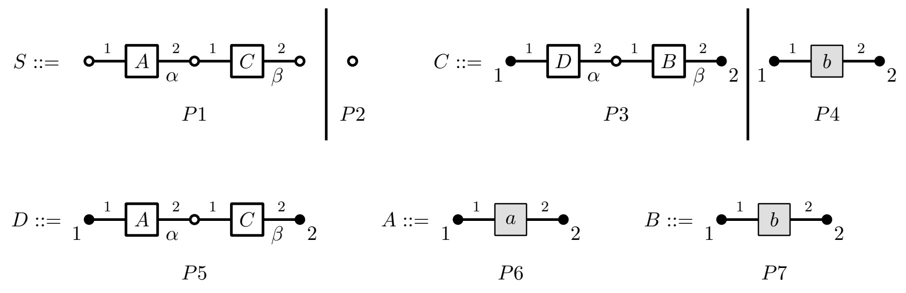


Figure 5.2: Non-ambiguous string graph grammar representing the language $\{a^n b^n \mid n \geq 0\}$

Observation 5.1.1. *Grammar AB is a non-ambiguous context-free string graph grammar representing the context-free language* $\{a^n b^n \mid n \geq 0\}$.

Proof. The proof of non-ambiguity of the *HRG* in Figure 5.2 representing the language $\{a^n b^n \mid n \geq 0\}$ comes from a simple analysis of the grammar. Since each non-terminal production has on its *rhs* a non-terminal leading to a single terminal production and there are no productions having the same *lhs* we assume that there are no options in the choices of either productions or splits. Table 5.1 results from the Algorithm **Pre** on input $(AB, 5)$ confirms that each choice during the generation is made with a probability of $1/1$, meaning that each string graph has a unique derivation. □

Table 5.1: Matrices M_1 and M_2 resulting from **Pre**$(AB, 5)$

M_1

N	1	3	5
S	0	0	1
C	1	0	1
D	0	1	0
A	1	0	0
B	1	0	0

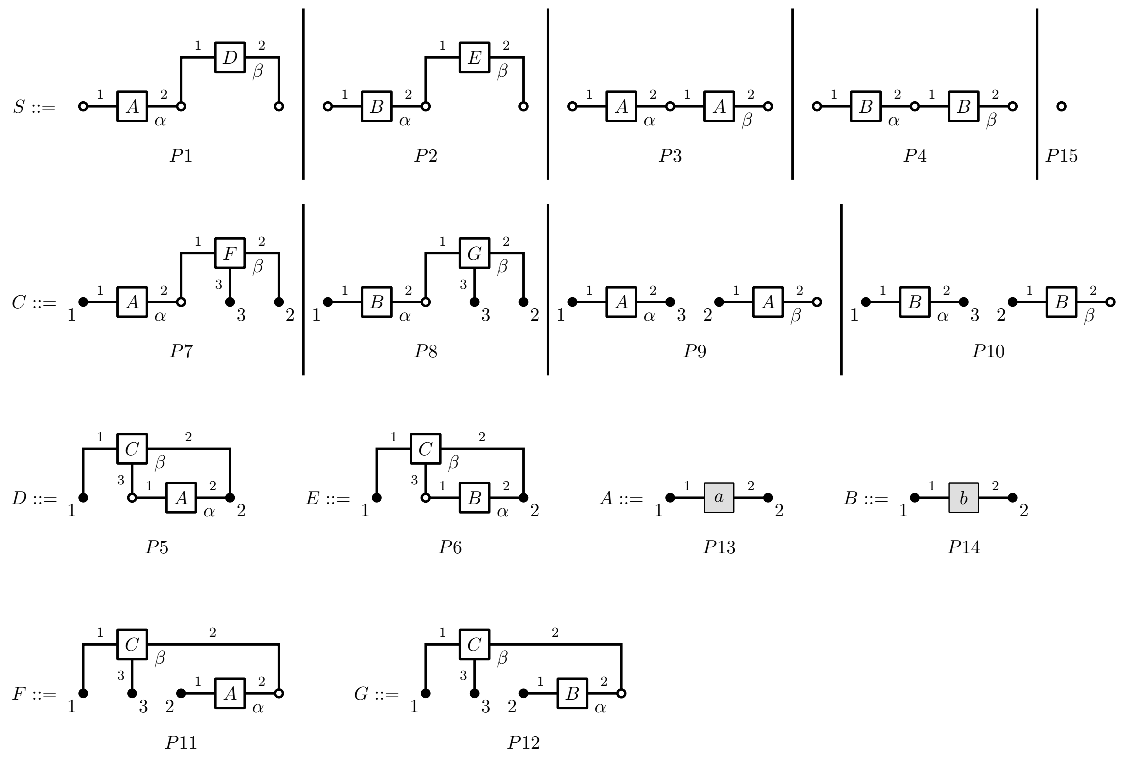
M_2

P	1	3	5
$S \xrightarrow{P1} AC, 3$	0	0	1
$C \xrightarrow{P3} DB, 1$	1	0	1
$D \xrightarrow{P5} AC, 1$	0	1	0
$A \xrightarrow{P6} a, 0$	1	0	0
$B \xrightarrow{P7} b, 0$	1	0	0
$C \xrightarrow{P4} b, 0$	1	0	0
$S \xrightarrow{P2} \lambda, 1$	0	0	0

5.2 Non-Context-Free String Languages

We know that the original Mairson's methods work with context-free string grammars and require it to be non-ambiguous to obtain a uniform random sampling. We are not aware of any systematic method that guarantees the sampling of non-context-free string grammars over a uniform distribution. Nevertheless, using an equivalent *HRG* in *CNF* we can overcome this problem. On the other hand, there is no systematic way to directly translate a non-context-free string grammar into its context-free graph counterpart. Engelfriet has show that these non-context-free string languages as $\{(a^n b)^m, n, m \geq 0\}$ or $\{w^n, w \in \{a,b\}^*, n \geq 0\}$, with $n, m \geq 0$, cannot be represented by an *HRG*.

We may notice that some of the hyperedges in the productions have the role to control the structure of the string while the others are simply replaced with their terminal



a

Figure 5.3: String graphs grammar representing the non-context-free language $L = \{ww \mid w \in \{a,b\}^*\}$.

counterpart. To ease the understanding of the grammars we have kept these hyperedges in the representations on two different levels.

Tables 5.2 show the results of running the Algorithm **Pre** on input $(WW, 13)$ where the 0-filled columns have been excluded. From the first row of matrix M_2 we may notice that WW grammar is also non-ambiguous. An example of unique derivation representing the generation of graph $(aabaab)\bullet$ is shown in Figure 5.4.

Corollary 5.2.1. *Let G_H be an HRG representing a non-context-free string grammar G_S such that $L(G_H) = \{s\bullet \mid s \in L(G_S)\}$ then Algorithm **Gen** generates a size-n-hypergraph $s\bullet \in L_n(G_H)$ equivalent to a size $m = (n-1)/2$ string $s \in L_m(G_S)$, provided that $L_n(G_H) \neq \emptyset$. If G_H is n-unambiguous and RNG is a uniform random number generator, the hypergraph is chosen uniformly at random.*

Proof. Let G_H be a hyperedge replacement grammar and let G_S be its corresponding string grammar. Since $L(G_H) = \{s\bullet \mid s \in L(G_S)\}$, according to Definition 5.0.1, every generated string graph $s\bullet \in L(G_H)$ has one and only corresponding string $s \in L(G_S)$. Then, generating graphs in $L(G_H)$ is equivalent to generating strings in $L(G_S)$. If G_H is n-unambiguous then the string s is generated uniformly at random. □

5.3 Inherently Ambiguous String Languages

In this section we explore the possibility to solve the uniform sampling problem for inherently ambiguous string languages using hyperedge replacement grammars. The methods presented in Chapter 3 have been proven to generate hypergraphs with a uniform distribution if the input grammar is non-ambiguous and we also know from Chapter 4 that some hyperedge replacement languages are inherently ambiguous due to the structure of their members.

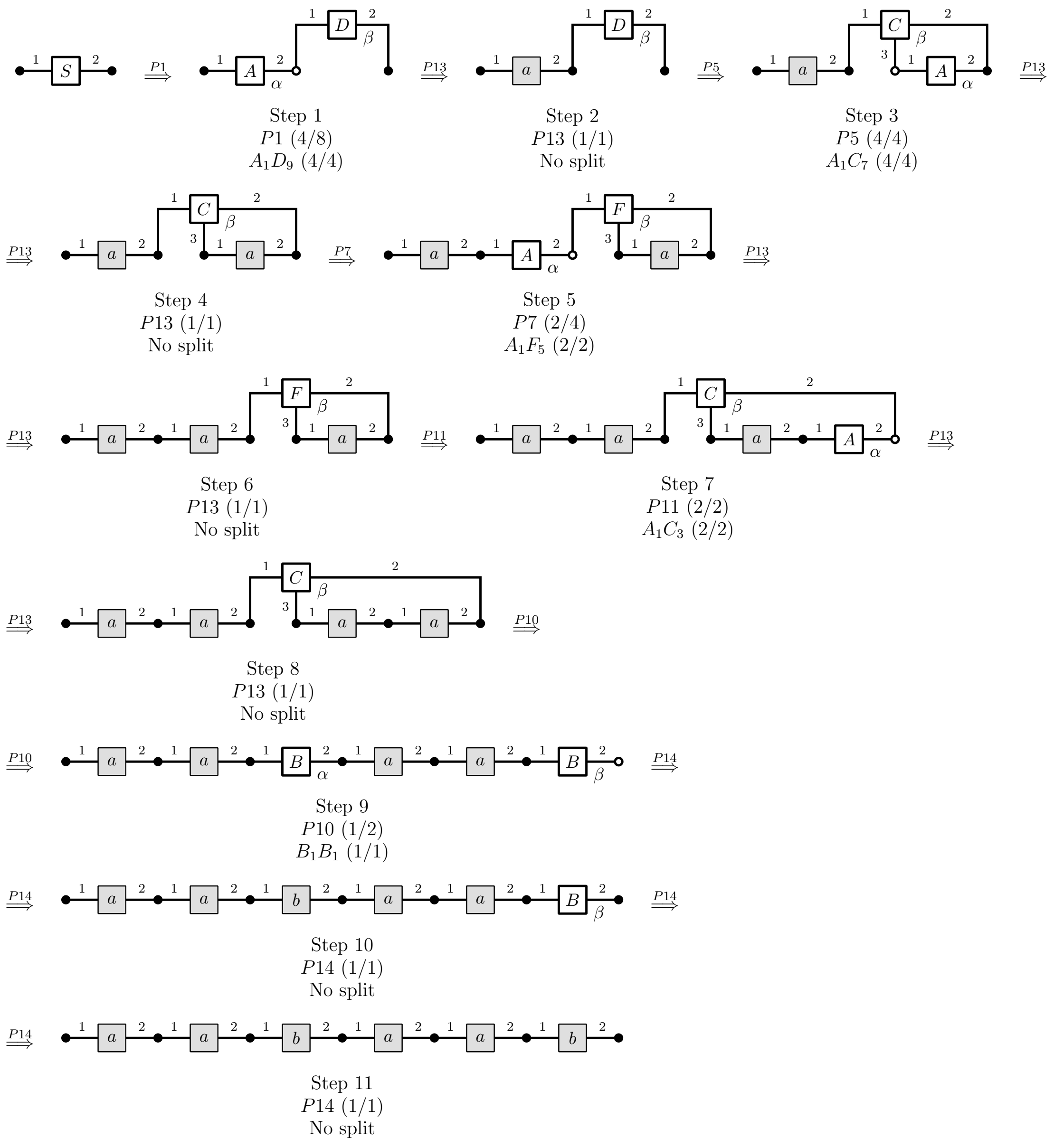


Figure 5.4: Derivations of graph $(aabaab)\bullet$

Table 5.2: Matrices M_1 and M_2 resulting from $\mathbf{Pre}(WW, 13)$

M_1

N	1	3	5	7	9	11	13
S	0	0	2	0	4	0	8
C	0	2	0	4	0	8	0
D	0	0	2	0	4	0	8
E	0	0	2	0	4	0	8
F	0	0	2	0	4	0	8
G	0	0	2	0	4	0	8
A	1	0	0	0	0	0	0
B	1	0	0	0	0	0	0

M_2

P	1	3	5	7	9	11	13
$S \xrightarrow{P1} AD, 3$	0	0	0	0	2	0	4
$S \xrightarrow{P2} BE, 3$	0	0	0	0	2	0	4
$S \xrightarrow{P3} AA, 3$	0	0	1	0	0	0	0
$S \xrightarrow{P4} BB, 3$	0	0	1	0	0	0	0
$C \xrightarrow{P7} AF, 1$	0	0	0	2	0	4	0
$C \xrightarrow{P8} BG, 1$	0	0	0	2	0	4	0
$C \xrightarrow{P9} AA, 1$	0	1	0	0	0	0	0
$C \xrightarrow{P10} BB, 1$	0	1	0	0	0	0	0
$D \xrightarrow{P5} AC, 1$	0	0	2	0	4	0	8
$E \xrightarrow{P6} BC, 1$	0	0	2	0	4	0	8
$F \xrightarrow{P11} AC, 1$	0	0	2	0	4	0	8
$G \xrightarrow{P12} BC, 1$	0	0	2	0	4	0	8
$A \xrightarrow{P13} a, 0$	1	0	0	0	0	0	0
$B \xrightarrow{P14} a, 0$	1	0	0	0	0	0	0

While string languages may also be inherently ambiguous, the shape of the members of their hypergraph counterparts is simply a chain of hyperedges. Thus, they cannot present any structural symmetry. Clearly, there isn't any known algorithm that can in general produce a non-ambiguous *HRG* from an ambiguous string grammar, otherwise it could be easily considered a decider for the ambiguity problem. Using this algorithm we could produce a new graph grammar that is surely non-ambiguous from the string one. Then, running our pre-processing algorithm on the graph grammar and Mairson's pre-processing algorithm on the string one, we could compare the results for a sufficiently large size. On the other hand, analyzing each language separately would not lead to any useful result.

In [24] Flajolet proposes a classification of string languages and proves their inherent ambiguity. Each class is composed by similar languages that are all reducible to a basic form, thus, finding a solution for the representative form of a class automatically solves the problem for all of its members. We prove that for some of them it is possible to find an equivalent non-ambiguous hyperedge replacement grammar to sample uniformly at random from, but, we also prove that it is not possible for all the classes.

Theorem 5.3.1. *Let G^2_H, G^3_H be context-free HRGs and $L(G^1_S) = L(G^2_S) \cup L(G^3_S)$ be a string language such that $L(G^2_H) = \{s\bullet \mid s \in L(G^2_S)\}$ and $L(G^3_H) = \{s\bullet \mid s \in L(G^3_S)\}$. If $L(G^2_H), L(G^3_H)$ and $L(G^2_H \cap G^3_H)$ are not inherently ambiguous, then there exists a non-ambiguous grammar G^1_H such that $L(G^1_H) = \{s\bullet \mid s \in L(G^1_S)\}$ and Algorithm **Gen** generates a size-n-hypergraph $s\bullet \in L_n(G^1_H)$ uniformly at random equivalent to a size $m = (n-1)/2$ string $s \in L_m(G^1_S)$.*

Proof. For $i \in \{1, 2, 3\}$ let G^i_S be string grammars and $G^i_H = (N_i, \Sigma_i, P_i, S_i, (mark_p)_{p\in P_i})$ context-free *HRG*s such that $L(G^i_H) = \{s\bullet \mid s \in L(G^i_S)\}$. Let G^2_H, G^3_H be non-ambiguous.

If $L(G^2_H) \cap L(G^3_H) = \emptyset$ then the grammar $G^1_H = (N_1, \Sigma_1, P_1, S_1, (mark_p)_{p\in P_1})$ where:

1. $N_1 = N_2 \cup N_3 \cup \{S_1\}$
2. $\Sigma_1 = \Sigma_2 \cup \Sigma_3$
3. $S_1 \notin N_2 \cup N_3$

4. $P_1 = P_2 \cup P_3 \cup \{(S_1, S_2^\bullet), (S_1, S_3^\bullet)\}$

representing the union of two distinct non inherently ambiguous context-free *HRL*s is trivially non-ambiguous since for each derivation $S_1^\bullet \Longrightarrow K \Longrightarrow^* s\bullet$, $s\bullet \in L(G_H^2)$ if and only if $K = S_2^\bullet$ or $s\bullet \in L(G_H^3)$ if and only if $K = S_3^\bullet$.

If $L(G_H^2) \cap L(G_H^3) \neq \emptyset$, let K be a sentential form in a derivation $S_1^\bullet \Longrightarrow^* K \Longrightarrow^* s\bullet$ with $s\bullet \in L(G_H^2) \cap L(G_H^3)$. Considering the total language tree of $L(G_H^1)$ we distinguish the following cases:

1. derivations $S_1^\bullet \Longrightarrow^* K' \Longrightarrow^* s'\bullet$ yielding graphs in $L(G_H^2) \cap L(G_H^3)$ having $K' \cong K$ and $s'\bullet \cong s\bullet$.

2. derivations $S_1^\bullet \Longrightarrow^* s'\bullet$ yielding graphs in $L(G_H^2) \oplus L(G_H^3)$ not having any sentential form $K' \cong K$.

3. derivations $S_1^\bullet \Longrightarrow^* K' \Longrightarrow^* s'\bullet$ yielding graphs in $L(G_H^2) \oplus L(G_H^3)$ having $K' \cong K$ but $s'\bullet \ncong s\bullet$.

In the first case, for the hypothesis, $L(G_H^2) \cap L(G_H^3)$ is not inherently ambiguous. Then there exists a non-ambiguous grammar $C1$ so that $L(C1) = L(G_H^2) \cap L(G_H^3)$.

In the second case, since the symmetric difference $L(G_H^2) \oplus L(G_H^3)$ presents 2 distinct sets of hypergraphs generated by productions applied to derivations having different sentential forms we already know that a non-ambiguous grammar $C2$ can be constructed for the subset of graphs corresponding to the case $L(G_H^2) \oplus L(G_H^3)$ not having any sentential form $K' \cong K$.

In the third case, we know that there are productions either in P^2 or P^3 that generate a hypergraph $s'\bullet$ from K' such that for each $s\bullet \in L(G_H^2) \cap L(G_H^3)$ $s'\bullet \ncong s\bullet$. Moreover we know that K' is not a sentential form, with the exception of a terminal graph, for any derivation involved in the production of graphs in $L(G_H^2) \oplus L(G_H^3)$. Finally, for the non-ambiguity of $C1$ we know that we can safely assume that integrating $C1$ with such productions cannot cause the generation of isomorphic graph, otherwise such graphs would have already been included in the first case.

Since there exist non-ambiguous context free *HRG*s $C1$ and $C2$ representing the language $L(G_H^1) = L(C1) \cup L(C2)$ and, again, $L(C1) \cap L(C2) = \emptyset$ are non inherently ambiguous context-free languages, then $L(G_H^1)$ is also non inherently ambiguous. □

5.4 Non-ambiguous *HRG*s for Inherently Ambiguous String Languages

Theorem 5.3.1 can be applied when considering a context-free string language L resulting from the union of two more context-free languages $L_1 \cup L_2$. For example, even if there exist a pair of non-ambiguous grammars representing them, L could still be inherently ambiguous. The intersection $L_1 \cap L_2$ may indeed be a non-context-free language.

The general idea is to separate a string language in pairwise distinct subsets. If all of them can be represented by a non-ambiguous *HRG*, then, we can find an equivalent non-ambiguous grammar representing the whole language.

The language $L(ABC_S) = \{a^n b^m c^p \mid n = m \textit{ or } m = p, \textit{with } n, m, p \geq 0\}$ may be represented by the following string grammar in *CNF*:

$$\begin{aligned}
S &::= DE \mid AF \mid AB \mid CE \mid HG \mid BI \mid BC \mid AH \mid a \mid c \mid \lambda \\
D &::= AF \mid AB \\
E &::= CE \mid c \\
F &::= DB \\
G &::= BI \mid BC \\
H &::= AH \mid a \\
I &::= GC \\
A &::= a \\
B &::= b \\
C &::= c
\end{aligned}$$

Such a grammar may separately produce words in both subsets of the language where $n = m$ or $m = p$. From the result of running Mairson's first method pre-processing algorithm on grammar ABC_S shown in Table 5.3 we may deduce that the entry $S[3]$ has a value of 4. Since there are only 3 strings in the 3th slide of the language (*aaa*, *abc*, *ccc*), it means that one of the strings is counted twice. Such a string is *abc* because it belongs to both subsets $n = m$ and $m = p$.

Table 5.3: Matrices M_1 and M_2 resulting from the pre-processing phase for the string grammar ABC_S and $n = 3$

M_1

N	1	2	3
S	2	4	4
D	0	1	0
E	1	1	1
F	0	0	1
G	0	1	0
H	1	1	1
I	0	0	1
A	1	0	0
B	1	0	0
C	1	0	0

M_2

P	1	2	3
$S \longrightarrow DE$	0	0	1
$S \longrightarrow AF$	0	0	0
$S \longrightarrow AB$	0	1	0
$S \longrightarrow CE$	0	1	1
$S \longrightarrow HG$	0	0	1
$S \longrightarrow BI$	0	0	0
$S \longrightarrow BC$	0	1	0
$S \longrightarrow AH$	0	1	1
$D \longrightarrow AF$	0	0	0
$D \longrightarrow AB$	0	1	0
$E \longrightarrow CE$	0	1	1
$F \longrightarrow DB$	0	0	1
$G \longrightarrow BI$	0	0	0
$G \longrightarrow BC$	0	1	0
$H \longrightarrow AH$	0	1	1
$I \longrightarrow GC$	0	0	1
$S \longrightarrow a$	1	0	0
$S \longrightarrow c$	1	0	0
$E \longrightarrow c$	1	0	0
$H \longrightarrow a$	1	0	0
$A \longrightarrow a$	1	0	0
$B \longrightarrow b$	1	0	0
$C \longrightarrow c$	1	0	0

Figure 5.5 shows a non-ambiguous *HRG* for the language $L(ABC_S)$. Observing the productions we may notice that an equal number of *a*s, *b*s and *c* is generated at first and then more *a*s, *c*s or groups of *ab* or *bc* are subsequently added if necessary.

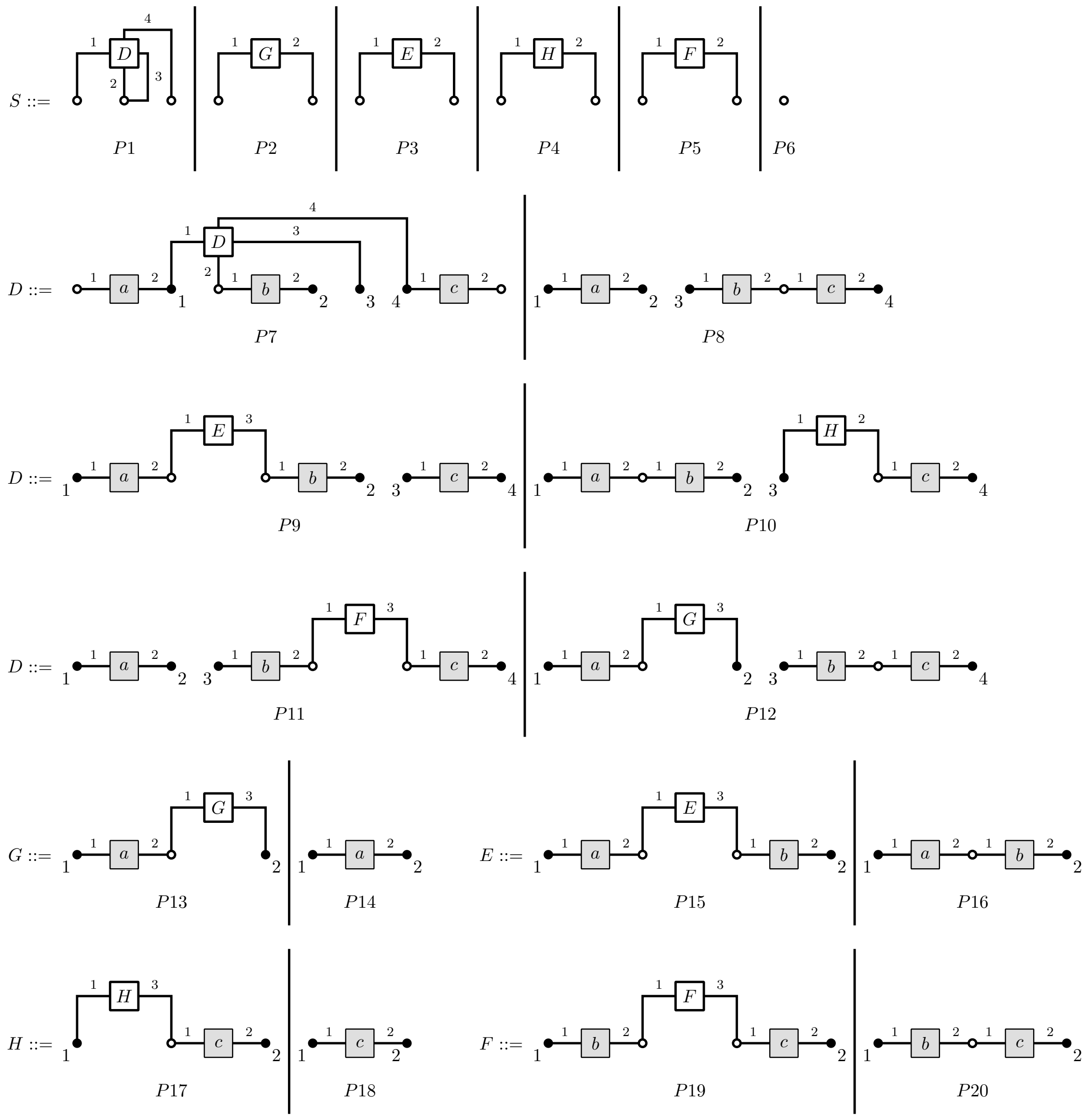


Figure 5.5: Non-ambiguous string graphs grammar representing the inherently ambiguous string language $L = \{a^n b^m c^p \mid n = m \textit{ or } m = p, \textit{with } n, m, p \geq 0\}$

The grammar ABC_H in Figure 5.6 is the Chomsky normal form obtained from the application of the rules defined in Section 2.4 to the grammar in Figure 5.5.

Running **Pre** on input $(ABC_H, 7)$ we obtain the Matrices in Table 5.4. The entry $S[7]$, corresponding to a string of size 3 now corresponds to a correct value of 3 showing that the grammar is non-ambiguous.

It is also particularly interesting to apply Theorem 5.3.1 to the first class of inherently ambiguous string languages proposed in [24]. It consists in languages that still have a constraint in the number of symbols in the word, but there is no constraint on their order. Languages like O_3 and Ω_3 are in this class. While it is argued that the inherent ambiguity of these and even more complex languages can be reduced to their simple counterpart $L(ABC_S)$, to prove their inherent ambiguity, we argue that this assumption should be re-evaluated in light of the present result on string graph languages that admit instead a non-ambiguous grammar.

As a proof of the existence of languages that are inherently ambiguous for both their string and graph representation, we consider the Goldstine language [24]. Let $n, p \in \mathbb{N}$ with $n \geq 0$ and $p \geq 1$ and let $\underline{n}$ denote the unitary representation of n in the form $a^n b$. Then we can define a class of languages with different constraints based on the variations of the original sequence $\underline{n}_1, \ldots, \underline{n}_p$. For example $G_{\neq} = (\underline{n}_1, \underline{n}_2, \ldots, \underline{n}_p \mid \exists j \colon n_j \neq j)$ is the language of all sequences of group of symbols $a^i b$ for $1 \leq i \leq p$ where there exists at least one component having its length not corresponding with the index of its position. The string *abaabaaabaabaaaaab* having the 4th component *aab* instead of *aaaab* is a member of this language. Similarly to $G_{\neq}$ we may consider constraints as $n_j = j$, $n_j < j$ or $n_j > j$.

Proposition 5.4.1. *The Goldstine HRL* $G_{\neq} = (\underline{n}_1, \underline{n}_2, \ldots, \underline{n}_p \mid \exists j \colon n_j \neq j)$ *where* $\underline{n}$ *denotes the unitary representation of* n *in the form* $(a^n b)\bullet$ *is inherently ambiguous.*

Proof. In $G_{\neq}$ every string has a common base made by a sequence of components $(a^i b)_j$, with $i = j < p$, but having at least one of them $i \neq j$. Moving to *HRG*s, a simple solution would be to generate the non corresponding component first and then generate the rest of the components with an unbound number of As. Even if we have correctly assumed that string graphs do not have structural symmetries, the generation of this sequence can be compared to a case non-finite unconnected components as defined in Section 4.2 for $(e^*)m$ star graphs. Imagining unconnected groups of As separated by Bs, according to Corollary 4.2.1, the language is indeed inherently ambiguous. $\square$

Moreover, if we want to attempt the construction of a non-ambiguous grammar, we would need to generate the sequence $(A^1 B A^2 B \ldots A^p B)\bullet$ of non-terminals and then add or remove at least 1 hyperedge labelled as A. As we may notice, such a sequence can be reduced to the language $(a^n b)^m$, know to be non-context-free [19]. Similar proofs can be constructed for the constraints $n_j = j$, $n_j < j$ and $n_j > j$.

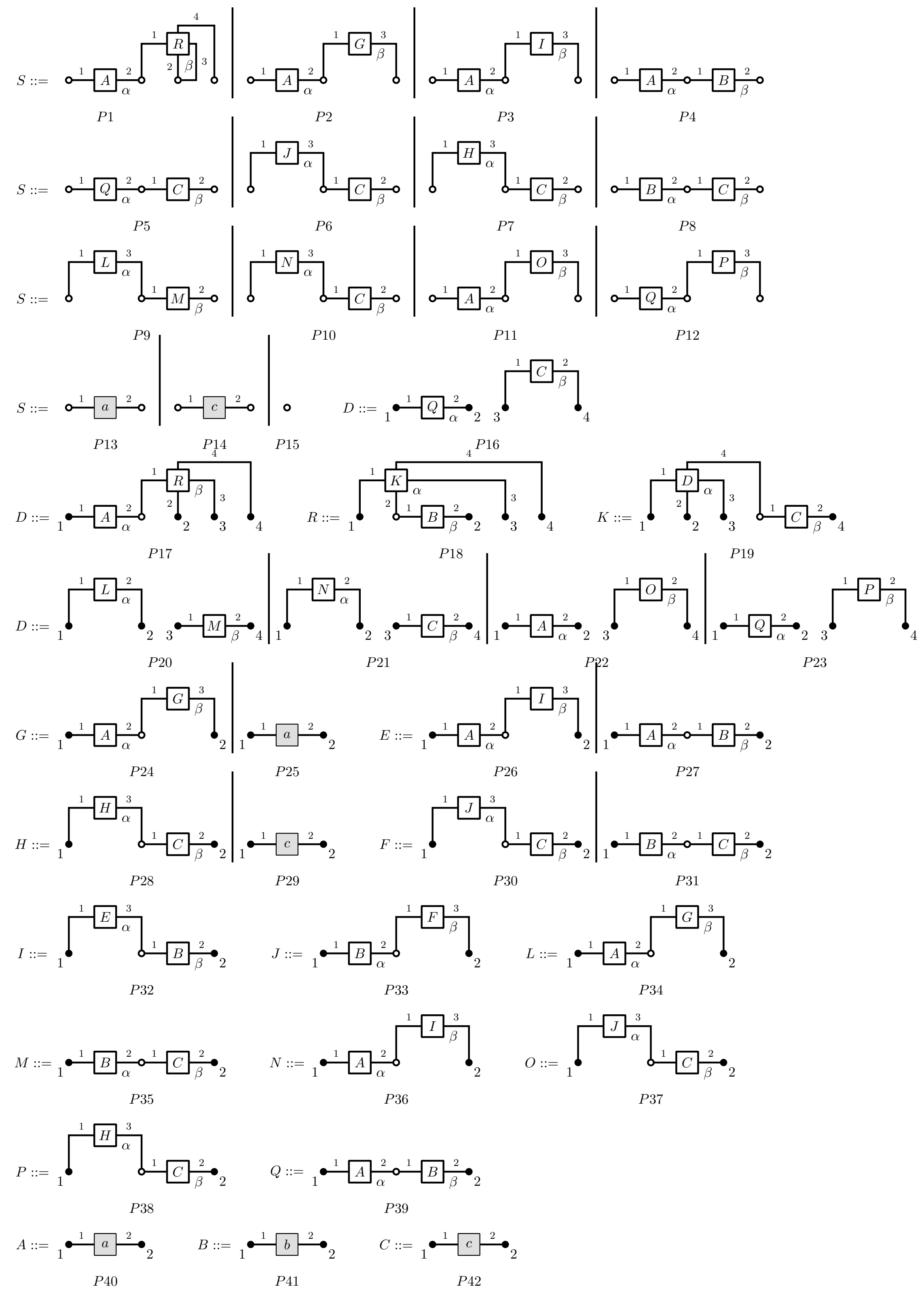


Figure 5.6: Non-ambiguous string graphs grammar in Chomsky normal form ABC_H representing the inherently ambiguous string language $L = \{a^n b^m c^p \mid n = m \ or \ m = p, with \ n, m, p \geq 0\}$

Table 5.4: Matrices M_1 and M_2 resulting from $\mathbf{Pre}(ABC_H, 7)$

M_1

N	1	2	3	4	5	6	7
S	0	0	2	0	4	0	3
D	0	0	0	1	0	3	0
E	0	0	1	0	0	0	0
F	0	0	1	0	0	0	0
G	1	0	1	0	1	0	1
H	1	0	1	0	1	0	1
I	0	0	0	0	1	0	0
J	0	0	0	0	1	0	0
K	0	0	0	0	0	1	0
L	0	0	1	0	1	0	1
M	0	0	1	0	0	0	0
N	0	0	0	0	1	0	0
O	0	0	0	0	1	0	0
P	0	0	1	0	1	0	1
Q	0	0	1	0	0	0	0
R	0	0	0	0	0	0	0
A	1	0	0	0	0	0	0
B	1	0	0	0	0	0	0
C	1	0	0	0	0	0	0

M_2

P	1	2	3	4	5	6	7
$S \xrightarrow{P1} AR, 4$	0	0	0	0	0	0	0
$S \xrightarrow{P2} AG, 3$	0	0	0	0	1	0	1
$S \xrightarrow{P3} AI, 3$	0	0	0	0	0	0	0
$S \xrightarrow{P4} AB, 3$	0	0	0	0	1	0	0
$S \xrightarrow{P5} QC, 3$	0	0	0	0	0	0	1
$S \xrightarrow{P6} JC, 3$	0	0	0	0	0	0	0
$S \xrightarrow{P7} HC, 3$	0	0	0	0	1	0	1
$S \xrightarrow{P8} BC, 3$	0	0	0	0	1	0	0
$S \xrightarrow{P9} LM, 3$	0	0	0	0	0	0	0
$S \xrightarrow{P10} NC, 3$	0	0	0	0	0	0	0
$S \xrightarrow{P11} AO, 3$	0	0	0	0	0	0	0
$S \xrightarrow{P12} QP, 3$	0	0	0	0	0	0	0
$D \xrightarrow{P16} QC, 0$	0	0	0	0	0	1	0
$D \xrightarrow{P17} AR, 1$	0	0	0	0	0	0	0
$R \xrightarrow{P18} KB, 1$	0	0	0	0	0	0	0
$K \xrightarrow{P19} DC, 1$	0	0	0	0	0	1	0
$D \xrightarrow{P20} LM, 0$	0	0	0	0	0	1	0
$D \xrightarrow{P21} NC, 0$	0	0	0	0	0	0	0
$D \xrightarrow{P22} AO, 0$	0	0	0	0	0	0	0
$D \xrightarrow{P23} QP, 0$	0	0	0	0	0	1	0
$G \xrightarrow{P24} AG, 1$	0	0	1	0	1	0	1
$E \xrightarrow{P26} AI, 1$	0	0	0	0	0	0	0
$E \xrightarrow{P27} AB, 1$	0	0	1	0	0	0	0
$H \xrightarrow{P28} HC, 1$	0	0	1	0	1	0	1
$F \xrightarrow{P30} JC, 1$	0	0	0	0	0	0	0
$F \xrightarrow{P31} BC, 1$	0	0	1	0	0	0	0
$I \xrightarrow{P32} EB, 1$	0	0	0	0	1	0	0
$J \xrightarrow{P33} BF, 1$	0	0	0	0	1	0	0
$L \xrightarrow{P34} AG, 1$	0	0	1	0	1	0	1
$M \xrightarrow{P35} BC, 1$	0	0	1	0	0	0	0
$N \xrightarrow{P36} AI, 1$	0	0	0	0	0	0	1
$O \xrightarrow{P37} JC, 1$	0	0	0	0	0	0	1
$P \xrightarrow{P38} HC, 1$	0	0	1	0	1	0	1
$Q \xrightarrow{P39} AB, 1$	0	0	1	0	0	0	0
$S \xrightarrow{P13} a, 2$	0	0	1	0	0	0	0
$S \xrightarrow{P14} c, 2$	0	0	1	0	0	0	0
$G \xrightarrow{P25} a, 0$	1	0	0	0	0	0	0
$H \xrightarrow{P29} c, 0$	1	0	0	0	0	0	0
$A \xrightarrow{P40} a, 0$	1	0	0	0	0	0	0
$B \xrightarrow{P41} b, 0$	1	0	0	0	0	0	0
$C \xrightarrow{P42} c, 0$	1	0	0	0	0	0	0

Chapter 6

Conclusion and Future Work

Our main results are that the methods presented in Chapter 3 efficiently generate hypergraphs from non-ambiguous hyperedge replacement grammars with a uniform distribution. Currently, the most promising application of our generation approach is the testing of programs working in graph-like domains. If the inputs of such programs are graphs in a context-free graph language, our method can generate test graphs uniformly at random in the domain of interest. This should allow to refine random testing approaches such as [35], [30], [8] and [40]. While the methods find their own unique space among existing algorithms for sampling graphs, their significant contribution of granting the user full freedom of choice, make them easily adaptable to a wide variety of contexts. Our method ensures that all and only the graphs of a specified size in the chosen domain will be taken into consideration during the sampling, without any successive need of testing for membership in order to guarantee that the resulting hypergraph possesses the sought properties. In future, it would be interesting to explore the possibility to extend the stochastic process of the generation phase by leveraging the symbolic representation of terminals in favor of the size of the grammar, improving even further the efficiency of the sampling.

Another result, described in Chapter 5, is the possibility to generate string graphs with non-ambiguous *HRGs* representing non-context-free string languages and, most importantly, certain classes of inherently ambiguous context-free string languages. This greatly extends the range of languages allowing for a sampling with a uniform distribution. Moreover, having identified a gap between inherently ambiguous string languages and their string graph counterpart, we should re-explore these languages in light of the current result. For example, the language $L(ABC_S) = \{a^n b^m c^p \mid n = m \textit{ or } m = p, \textit{with } n, m, p \geq 0\}$ does not represent an inherently ambiguous prototype for context-free string graph languages. A continuation of this research could answer the question of the existence of a similar fundamental structure for string graph languages to help discerning the ambiguity of their grammars. An interesting further analysis would be to explore the relation between the structure of an *HRG* and the degree of ambiguity of the corresponding string grammars as shown in [59]. Ultimately it could be relevant to study a hypergraph based meta-language for the direct transition from strings to graphs. Such a representation would ease the search for non-ambiguous solutions to even more inherently ambiguous string languages.

A third result, obtained from the cases examined in Chapter 4 is the possibility to study the ambiguity of graph grammars or even the inherent ambiguity of graph languages by reducing them to simple forms. We have already proven that languages such as the $(e^*)^m$ stars or the series-parallel graphs language are inherently ambiguous due to some particular structures found in their members. More complex languages may be proven inherently ambiguous by a direct reduction to the presented cases, simplifying the structures of their members to the symmetries proposed in section 2.3. Once identified, Variations of the sampling methods could then be investigated to solve the ambiguity problem for classes of languages presenting the same characteristics.

Among the several possible continuations of this work we have considered the following:

Weighted Hypergraphs In order to translate Mairson's methods from strings to hypergraphs we have defined the size of a graph to be the sum of the number of nodes and edges. This can also be interpreted as the sum of the elements of a graph all having the same weight of 1. In a more general context, we may consider different weights as an additional integer attribute of both nodes and edges, assigned, for example, to the elements of the *rhs* of a production. Our approach is then guaranteed to work as long as each production ensures a strictly positive increase of the overall sum of the weights during the derivation. That extends the application of our method in contexts where the merging of external nodes is allowed during the application of a production as proposed in [18]. Moreover it could be possible to present a novel method for the dynamic attribution of weights directly during the generation phase in order to investigate and improve the solution of weighted graph problems such as the dynamic shortest hyperpath as in [26] and [54].

Quasi-Polynomial-Time Approximation To generate graphs from a uniform distribution our method requires the input grammar to be non-ambiguous. If it's ambiguous, we cannot provide any information about the distribution the resulting hypergraph has been sampled from, nor can we improve the result without changing the grammar itself. So, an interesting topic would be to study the quasi-polynomial-time approximation algorithm of Gore et al. and extend it from strings to hypergraphs. In [31] they propose an algorithm that guarantees an approximated uniform distribution even for ambiguous context-free grammars. The algorithm is also based on Mairson's methods and compensates for the non-uniform distribution of the sampling, caused by the ambiguity of the input grammar, at the cost of efficiency. One of the challenges in the transition to graphs is to find an efficient test for graph isomorphism. A promising approach for this is the work of Babai [2], showing that the problem can be solved in quasi-polynomial time. This is a fundamental prerequisite for the elimination of duplicates and the consequent narrowing of the approximation of the sampling algorithm.

More Powerful Grammars Expanding the definition of context-free *HRG* presented in Chapter 2 by adding more features to the structure of the grammar may greatly extend the range of languages to sample from. For example, in [15] the so-called contextual hyperedge replacement which relaxes the restriction on the fixed type of the hyperedges we may include additional grammars to our environment. Having a special hyperedge connecting a variable number of nodes during a derivation allows to overcome the limitation of some of the languages proposed in our case studies, such as the stars graph language representing graphs with an arbitrary number of stars and unbounded points.

Another generalisation of the *HRGs* is the rendez-vous presented in [13], in which the productions may include the merging of external nodes with nodes outside the replacement context. This allows the generation of grids, a known limitation for context-free *HRGs*. Consequently, it overcomes the limitation of languages structurally equivalent to $(a^n b)^m$, providing the possibility to find more non-ambiguous solutions.

An even more powerful form of graph transformation is the Double-pushout representation [51]. Even if we conjecture that, in the setting of context-freeness, our methods offers the most general solution, it is worth to explore the boundaries of their application to a computationally complete approach such as *DPO*.

Parsing A very interesting aspect of sampling using ordered derivation trees is that they produce ordered hypergraphs. Since the hyperedges are replaced by a recursive function, when the generation algorithm terminates, the terminal hyperedges still hold the information about the order in which they have been created. It is straightforward then to rebuild the derivation tree given the resulting hypergraph. If we consider the complementary problem of parsing as in [5] and [16], it would be of great relevance to explore, given a hypergraph, this time without any ordering of the hyperedges, to what extent relying only on the information given by the ordered productions, improves the construction of parsing trees. In this case the ordering of hyperedges on the *rhs* may represent a useful constraint to help converging towards the correct parsing tree.

GP2 Implementation An implementation of the methods for the uniform generation of graphs would fit as a tool for efficiently testing graph based programs just as QuickCheck [40] is for Haskell. It offers an unbiased way to generate large domain specific datasets of graphs ensuring the most accurate results of the tests. It is also possible to consider an implementation as part of the rule based programming language GP2 [7] in which the algorithms could be directly built using the GP2 interface. That is, exploiting all the advantages of GP2 such as representing the rules directly as graph transformations and working with an ad-hoc graphical interface.

Beyond Software Testing Molecular biology and Cryptography are two important fields of application where our methods could find a concrete use besides software testing.

In [17] and [42] the representation of molecules through hypergraphs takes advantage of the possibility of hyperedge to describe multiple links simultaneously. An adaption of our methods to this setting would provide a key instrument for the exploration of new compounds, either by the possible definition of a context-free grammar capable of generating such hypergraphs and the study of the probability distribution underlying the generation process itself.

The proof of uniformity of the distribution of our methods is crucial for the development of cryptographic protocols. It is indeed a fundamental requirement to prove the computational security when a corresponding mathematical proof is missing. In [29] and [48] we may find some useful insights on how to model graph based algorithms. In the setting of *HRG*, we believe that there is an opportunity for the development of one-way functions based, for example, on the encoding of the grammar. The ordering of the hyperedges on the *rhs* may represent the key for the generation of hypergraph, then shared as public parameter. This is clearly in opposition with the previously described problem of parsing. The question *"Is a member of a context-free HRL easier to parse knowing that its derivation tree is ordered?"* leads indeed to an advantage despite of its answer.

Chapter 7

Bibliography

[1] S. Aguiñaga, R. Palácios, D. Chiang, and T. Weninger. Growing graphs with hyperedge replacement graph grammars. Computing Research Repository, abs/1608.03192, 2016.

[2] L. Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing (STOC 16), pages 684–697. Association for Computing Machinery, 2016.

[3] C. Berge. Théorie des Graphes et ses Applications. Dunod, 1958.

[4] C. Berge. Hypergraphs: Combinatorics of Finite Sets, volume 45. North-Holland Mathematical Library, 1984.

[5] H. Björklund, F. Drewes, P. Ericson, and F. Starke. Uniform parsing for hyperedge replacement grammars. Journal of Computer and System Sciences, 118:1–27, 2021.

[6] B. Bollobás. Random Graphs, pages 215–252. Springer, 1998.

[7] G. Campbell, B. Courtehoute, and D. Plump. Fast rule-based graph programs. Science of Computer Programming, 214, 2022. 32 pages.

[8] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse. Adaptive random testing: The art of test case diversity. Journal of Systems and Software, 83(1):60–66, 2010.

[9] N. Chomsky. On certain formal properties of grammars. Information and Control, 2(2):137–167, 1959.

[10] T. Coolen, A. Annibale, and E. Roberts. Generating Random Networks and Graphs. Oxford University Press, 2017.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. MIT press, 2022.

[12] B. Courcelle. An axiomatic definition of context-free rewriting and its application to NLC graph grammars. Theoretical Computer Science, 55(2):141–181, 1987.

[13] G. David, F. Drewes, and H.-J. Kreowski. Hyperedge replacement with rendezvous. In Proceedings Theory and Practice of Software Development (TAPSOFT 93), volume 18 of LNCS, pages 167–181. Springer, 1993.

[14] F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In Handbook of Graph Grammars and Computing by Graph Transformation, volume 1, pages 95–162. World Scientific, 1997.

[15] F. Drewes and B. Hoffmann. Contextual hyperedge replacement. Acta Informatica, 52:497–524, 2015.

[16] F. Drewes, B. Hoffmann, and M. Minas. Formalization and correctness of predictive shift-reduce parsers for graph grammars based on hyperedge replacement. Journal of Logical and Algebraic Methods in Programming, 2019.

[17] K. Ehrig, R. Heckel, and G. Lajios. Molecular analysis of metabolic pathway with graph transformation. In Proceedings of the Third international conference on Graph Transformations (ICGT 06), volume 4178 of LNCS, pages 107–121. Springer, 2006.

[18] J. Engelfriet. Context-free graph grammars. In Handbook of Formal Languages, volume 3, pages 125–213. Springer, 1997.

[19] J. Engelfriet and L. Heyker. The string generating power of context-free hypergraph grammars. Journal of Computer and System Sciences, 43(2):328–360, 1991.

[20] D. Eppstein. Parallel recognition of series-parallel graphs. Information and Computation, 98(1):41–55, 1992.

[21] P. Erdős and A. Rényi. On the evolution of random graphs. Publications of the Mathematical Institute of the Hungarian Academy of Sciences, 5(1):17–61, 1960.

[22] P. Erdős and A. Rényi. Asymmetric graphs. Acta Mathematica Academiae Scientiarum Hungarica, 14:295–315, 1963.

[23] L. Euler. Solutio problematis ad geometriam situs pertinentis. Commentarii Academiae Scientiarum Imperialis Petropolitanae, pages 128–140, 1741.

[24] P. Flajolet. Analytic models and ambiguity of context-free languages. Theoretical Computer Science, 49(2–3):283–309, 1987.

[25] P. Flajolet, P. Zimmermann, and B. V. Cutsem. A calculus for the random generation of labelled combinatorial structures. Theoretical Computer Science, 132(1–2):1–35, 1994.

[26] J. Gao, Q. Zhao, W. Ren, A. Swami, R. Ramanathan, and A. Bar-Noy. Dynamic shortest path algorithms for hypergraphs. Computing Research Repository, abs/1202.0082, 2012.

[27] D. T. Gillespie. Stochastic simulation of chemical kinetics. Annual Review of Physical Chemistry, 58:35–55, 2007.

[28] S. Ginsburg and J. Ullian. Ambiguity in context free languages. Journal of the ACM, 13(1):62–89, 1966.

[29] O. Goldreich. Candidate one-way functions based on expander graphs. In O. Goldreich, editor, Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation, pages 76–87. Springer, 2011.

[30] O. Goldreich. Introduction to Property Testing. Cambridge University Press, 2017.

[31] V. Gore, M. Jerrum, S. Kannan, Z. Sweedyk, and S. Mahaney. A quasi-polynomial-time algorithm for sampling words from a context-free language. Information and Computation, 134(1):59–74, 1997.

[32] S. A. Greibach. The undecidability of the ambiguity problem for minimal linear grammars. Information and Control, 6(2):119–125, 1963.

[33] A. Habel. Hyperedge Replacement: Grammars and Languages, volume 643 of LNCS. Springer, 1992.

[34] A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. Mathematical Structures in Computer Science, 11(5):637–688, 2001.

[35] R. Hamlet. Random testing. In Encyclopedia of Software Engineering. John Wiley and Sons, 2002.

[36] T. Hickey and J. Cohen. Uniform random generation of strings in a context-free language. SIAM Journal on Computing, 12(4):645–655, 1983.

[37] C. Hierholzer and C. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. Mathematische Annalen, 6(1):30–32, 1873.

[38] B. Hoffmann and M. Minas. Generalized predictive shift-reduce parsing for hyperedge replacement graph grammars. In Proceedings Language and Automata Theory and Applications: 13th International Conference, (LATA 2019), volume 11417 of Lecture Notes in Computer Science, pages 233–245. Springer, 2019.

[39] D. A. Huffman. A method for the construction of minimum-redundancy codes. Proceedings of the Iconic Science and Engineering, 40(9):1098–1101, 1952.

[40] J. Hughes. Software testing with QuickCheck. In Proceedings Central European Functional Programming School (CEFP 2009), volume 6299 of Lecture Notes in Computer Science, pages 183–223. Springer, 2009.

[41] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. Theoretical Computer Science, 43:169–188, 1986.

[42] H. Kajino. Molecular hypergraph grammar with its application to molecular optimization. In Proceedings 36th International Conference on Machine Learning, volume 97, pages 3183–3191. Proceedings of Machine Learning Research, 2019.

[43] S. Kannan, Z. Sweedyk, and S. Mahaney. Counting and random generation of strings in regular languages. In Proceedings Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 95), pages 551–557. Society for Industrial and Applied Mathematics, 1995.

[44] R. M. Karp. Reducibility among combinatorial problems. In Proceedings of a symposium on the Complexity of Computer Computations, pages 85–103. Springer US, 1972.

[45] P. S. le Marquis de Laplace. Théorie analytique des probabilits. In Œuvres Completes de Laplace, volume 7, pages 181–192. Gauthier-Villars, Imprimeur-Librarie, 3rd edition, 1820.

[46] G. N. Lewis. The atom and the molecule. Journal of the American Chemical Society, 38(4):762–785, 1916.

[47] H. G. Mairson. Generating words in a context-free language uniformly at random. Information Processing Letters, 49(2):95–99, 1994.

[48] S. Micali and R. L. Rivest. Transitive signature schemes. In Proceedings Topics in Cryptology (CT-RSA 2002), pages 236–243. Springer, 2002.

[49] M. Minas. Hypergraphs as a uniform diagram representation model. In Proceedings Theory and Application of Graph Transformations (TAGT 1998), volume 1764 of Lecture Notes in Computer Science, pages 281–295. Springer, 2000.

[50] M. Mosbah. Probabilistic hyperedge replacement grammars. Theoretical Computer Science, 159(1):81–102, 1996.

[51] D. Plump. Term graph rewriting. In Handbook of Graph Grammars and Computing by Graph Transformation, volume 2, pages 3–61. World Scientific, 1999.

[52] G. Robins, P. Pattison, Y. Kalish, and D. Lusher. An introduction to exponential random graph (p*) models for social networks. Social Networks, 29(2):173–191, 2007.

[53] S. S. Skiena. The Algorithm Design Manual, 3rd Edition. Springer, 2020.

[54] Sunita and D. Garg. Dynamizing Dijkstra: A solution to dynamic shortest path problem through retroactive priority queue. Journal of King Saud University - Computer and Information Sciences, 33, 2018.

[55] L. G. Valiant. The complexity of computing the permanent. Theoretical Computer Science, 8(2):189–201, 1979.

[56] L. G. Valiant. The complexity of enumeration and reliability problems. SIAM Journal on Computing, 8(3):410–421, 1979.

[57] F. Vastarini and D. Plump. Random graph generation in context-free graph languages. In Proceedings 13th International Workshop on Developments in Computational Models (DCM 2023), Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2024. To appear.

[58] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. Nature, 393:440–442, 1998.

[59] K. Wich. Sublinear ambiguity. In M. Nielsen and B. Rovan, editors, Proceedings Mathematical Foundations of Computer Science 2000, pages 690–698. Springer, 2000.