Graph Compression using Graph Grammars

Fabian Peternek

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2017

Abstract

This thesis presents work done on compressed graph representations via hyperedge replacement grammars. It comprises two main parts. Firstly the RePair compression scheme, known for strings and trees, is generalized to graphs using graph grammars. Given an object, the scheme produces a small context-free grammar generating the object (called a “straight-line grammar”). The theoretical foundations of this generalization are presented, followed by a description of a prototype implementation. This implementation is then evaluated on real-world and synthetic graphs. The experiments show that several graphs can be compressed stronger by the new method, than by current state-of-the-art approaches.

The second part considers algorithmic questions of straight-line graph grammars. Two algorithms are presented to traverse the graph represented by such a grammar. Both algorithms have advantages and disadvantages: the first one works with any grammar but its runtime per traversal step is dependent on the input grammar. The second algorithm only needs constant time per traversal step, but works for a restricted class of grammars and requires quadratic preprocessing time and space. Finally speed-up algorithms are considered. These are algorithms that can decide specific problems in time depending only on the size of the compressed representation, and might thus be faster than a traditional algorithm would on the decompressed structure. The idea of such algorithms is to reuse computation already done for the rules of the grammar. The possible speed-ups achieved this way is proportional to the compression ratio of the grammar. The main results here are a method to answer “regular path queries”, and to decide whether two grammars generate isomorphic trees.

Lay Summary

Data can take many forms. For one example, consider social networks. Here we have people and the relations (say, friendships) between them. In general terms, such data is called a *graph*. A graph consists of nodes (e.g., people) and edges connecting the nodes (e.g., if two people are friends, there is an edge between them). As there are many people using such networks, this data can get very large. When dealing with large data, it is common to use *compression* methods. These are algorithms, that represent the data given in a smaller way than the explicit representation. In our example, consider two people Alice and Bob, who know each other and thus have many friends in common. There is no need to store these common friends twice, it would instead suffice to represent them once and link both, Alice and Bob, to this common representation.

One particular method of compression is called *grammar-based* or *dictionary compression*. It works by finding recurring elements in the input data, removes them, and instead represents these elements using just one small symbol. This small symbol is further attached to a rule that stores the original recurring data, but only once instead of many times. Thus, we can recover the original data from the representing symbols. This thesis presents one way of doing this with graph data.

Compression usually comes at the cost that accessing the data takes longer than before, because it needs to be decompressed first. One advantage of grammar-based compression is, that this is not always the case. This is, because there exists a hierarchy on the symbols, and thus the recurring elements of the original data. With some queries, it is possible to use this hierarchy, and compute partial results of the query on only the recurring elements. These partial results can then be reused further up in the hierarchy and thus do not need to be computed again. This has the potential to lead to faster computation of specific queries, with a speed-up proportional to the compression ratio, which is by how much the representation is smaller than the original data. We present two queries, for which this is possible in the proposed representation.

Acknowledgements

Firstly, I would like to express my gratitude to my principal supervisor Dr. Sebastian Maneth, who took a lot of time, patiently read drafts, and always managed to point me in interesting directions in our personal meetings.

Secondly, I thank my thesis examiners for their careful examination of the thesis, helpful comments and a very pleasant defence.

Thirdly, I thank my office mates for their support and many enlightening discussions by the coffee machine providing just the right amount of distraction. Last but most certainly not least, I thank my family for their never wavering and continuing support in my pursuits. Even if it meant that I moved to a different country and worked in a profession somewhat removed from their own.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Fabian Peternek*)

Table of Contents

Chapter 1

Introduction

Part of the nature of technology is its constant evolution. Amongst other areas, this is very visibly exemplified by our capacity to store data. For example, in the ten years between 1997 and 2007 the capacity of commercially available hard disk drives grew by more than an order of magnitude: the IBM Deskstar 16GP of 1997 had a capacity of 16 GB, whereas 2007 saw the release of the first 1 TB hard disk drive (Hitachi Deskstar 7K1000) – at about the same price the IBM drive was 10 years prior. Development has slowed since, with hard drives as of 2017 being in the realm of 10 TB capacities, but networked storage in the “cloud” has been becoming of increasing importance offering virtually unrestricted capacities. At the same time, the ability to generate and collect more data, has sped up by at least the same amount, in some areas, possibly even faster. It is therefore of paramount importance to find ways to computationally deal with this data. As it is not feasible to wait for the storage capacity to catch up, *compression* is a popular method to do so. This allows to store the same amount using less space, while preserving the ability to access the data, albeit usually at slower speeds, as it needs to be decompressed first. Some of this dramatically increased amount of data is naturally represented using graphs. To give a few examples: internet search engines often use a network representation of the internet to rank the search results (they use the structure of the graph to estimate, which pages are most important, see e.g. PageRank [BP98]). The rising popularity of online social networks is another example. They have attracted a vast amount of users, whose connections are naturally modelled as a graph. Social network analysis is a key method of modern sociology with applications in many other fields (see e.g. [OR02]). Particularly visible to the general public is the use of this data in journalism, e.g. through the use of Twitter trends to provide insight into current events. A third example is the semantic web, an effort to annotate websites with semantic information, enabling new ways to search and explore data on the internet. A key technology of the semantic web is the Resource Description Framework (RDF), commonly represented in a graph format. These applications may intersect. Facebook introduced *graph search* in 2013 (however, it has since been deactivated again) combining social graphs with semantic information. The search allowed users to explore the underlying graph structure of the social network using semantic search terms given in natural language. In conclusion, this makes graph compression a specific area of interest.

This thesis presents a method of *grammar-based compression* for graphs, generalizing a well-known approximation algorithm from trees (and strings) to graphs. It further discusses the algorithmic aspects, particularly regarding speed-up algorithms. These are algorithms that can use the structure of the compressed representation to achieve a faster computation in comparison with an algorithm that runs on the uncompressed data, despite the usual slow-down in query-performance with compressed data. The remainder of this introduction gives a high-level overview of the methods used in the thesis and some of the results achieved.

1.1 Grammar-Based Compression

A popular formalism for compression is to use a context-free grammar G syntactically restricted such that its language is guaranteed to consist of only one element, referred to as $\mathsf{val}(G)$. These grammars are called *straight-line grammars*. Consider as an example the grammar

$$S \to B\texttt{\_is\_}B\texttt{\_and}A\texttt{\_}B$$
$$A \to \texttt{\_I\_am}$$
$$B \to \texttt{what}A$$

It represents the string "`what_I_am_is_what_I_am_and_I_am_what_I_am`", which is 41 characters long. The grammar on the other hand only consists of 23 characters (we count the size of a grammar by summing over the sizes of its right-hand sides). The grammar thus is a compressed representation of the original string. A reason for the success of grammar-based compression is its mathematical simplicity and the ability to capture classical compression schemes, such as variants of the popular Lempel-Ziv family of compressors [CLL$^+$05]. Unfortunately, that same paper also proves that finding a grammar of minimal size is NP-complete, making exact solutions infeasible. It then considers various approximations. One of these is the RePair-algorithm invented by Larsson and Moffat [LM00], which follows a simple yet effective idea: a most frequently occurring *digram* (pair of adjacent symbols) is replaced by a fresh nonterminal generating the digram as its rule. This process is recursively repeated until no digram appears more than once. For example: given the string $abcabcab$, RePair first replaces every occurrence of ab by a new nonterminal A, yielding the string $AcAcA$. Now the digram Ac appears twice and is thus replaced by B, which results in the final grammar

$$S \to BBA$$
$$A \to ab$$
$$B \to Ac$$

of size 7 compared to the original string, which has size 8. RePair was generalized to node-labelled, ordered, ranked trees using context-free tree grammars in [LMM13]. A major part of the present work is the further generalization of this method to edge-labelled graphs, using *graph grammars*. In the graph grammars used here, edges may be labelled by a nonterminal.

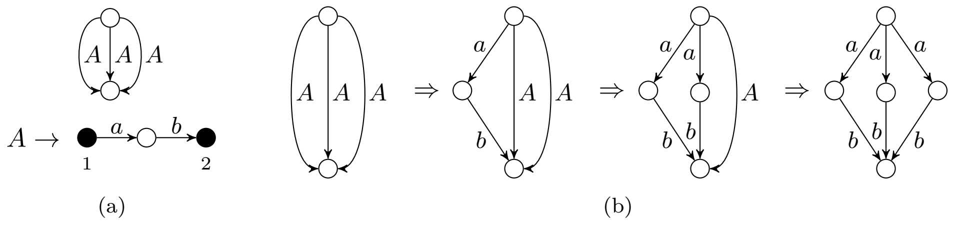


Figure 1.1: Example of a graph grammar with three nonterminal edges (a) and its full derivation (b).

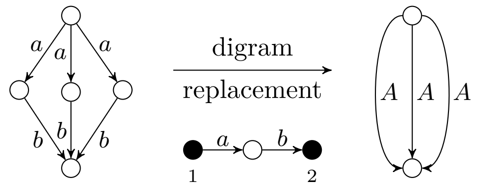


Figure 1.2: A digram replacement step resulting in the grammar of Figure 1.1a.

A nonterminal edge is, during the derivation, replaced by a right-hand side of an associated rule, just like with context-free string grammars. The difference is, that a string automatically provides an order (from left to right) and thus the insertion of the right-hand side is well defined. With graphs more information is necessary: replacing an edge should logically involve the nodes it is incident with, but further information is needed in the right-hand side of the rule to specify how and where it is connected with the remaining graph. This is done with specially marked *external nodes*, which are merged, in order, with the nodes incident with the nonterminal edge being replaced. Consider for example Figure 1.1a. It shows a graph with three nonterminal edges labelled A, and the corresponding rule for A. The right-hand side of this rule has two external nodes, which we fill black and label by their order. Figure 1.1b shows a full derivation of this grammar. During a derivation step, a nonterminal edge is removed and a disjoint copy of the right-hand side is added to the graph. Then the external nodes are merged, in order, with the nodes that were incident to the removed edge. As with strings above, we see that this structure is compressing: the startgraph has three edges and two nodes, the right-hand side of A has two edges and three nodes, resulting in a total size of 10. The derived graph on the other hand has five nodes and six edges, i.e., a size of 11.

Of course, to compress a graph using graph grammars this process is done in reverse. One of the first steps to generalize RePair is the definition of a digram. For strings, a digram is a pair of consecutive symbols. For trees a digram is a node, its j-th edge to a child, and that child node. When generalizing RePair to graphs, we define a digram as a pair of edges that share an incident node. Thus, the digram replacement shown in Figure 1.2 leads to the grammar of Figure 1.1a. Note that the digram used – an a edge followed by a b edge – was the most frequent one with three *non-overlapping* occurrences. The three a-edges (or the b-edges) also make three distinct occurrences, but any two of those have one edge in common. Thus they overlap and only one non-overlapping occurrence exist. This is important, as two overlapping occurrences cannot be replaced. In a first step, we want to find an initial set of non-overlapping occurrences of maximal

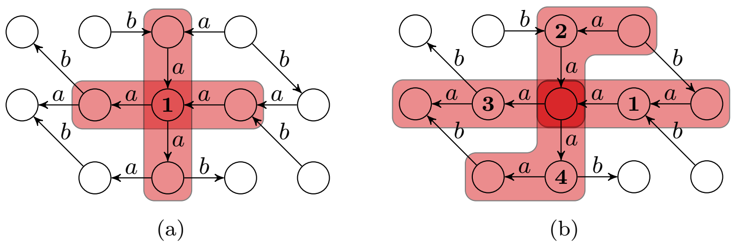


Figure 1.3: Two different traversals for counting digram occurrences.

size for every digram. This is one of the most important challenges faced in this generalization. It is possible to find such a set for a digram in at most $O(n^4)$ time, but this is not feasible in practice. With strings and trees it can be easily shown that searching for occurrences in a *greedy traversal* following a specific order (left to right for strings, post-order for trees) always results in a set of non-overlapping occurrences of maximal size. Unfortunately such an order is unknown for graphs and the order during a greedy traversal can have a significant effect on how many occurrences are found. Consider for example Figure 1.3. We count occurrences of digrams consisting of two a-labelled edges by using the edges incident to the currently visited central node. Starting in the centre node of the graph (node 1 in Figure 1.3a) two such occurrences are found. If on the other hand the central node is avoided and the nodes are visited in the order indicated in Figure 1.3b the maximal amount of four occurrences can be found. For this reason we evaluate four different node-orders using a prototype implementation. Out of those, an order inspired by the Weisfeiler-Lehman method to test for isomorphism [WL68] works best in our evaluation. It orders the nodes by first ordering them by their degrees and then refines this order using the neighbourhoods until a fixed point is reached.

Consider further the graph in Figure 1.4a: it is a slight variation of the graph in Figure 1.2. This small variation has an important effect. The digram replacement used before in Figure 1.2 is no longer possible. This is because, when replacing the same occurrences as before, the two edges labelled c are now missing the nodes they are attached to. However, by that reasoning there is no pair of edges that can be replaced by a simple nonterminal edge, as every node incident to any chosen pair is also incident to at least one edge not part of that pair. As this is a common phenomenon in graphs, we introduce *hyperedges*, which are edges that can be attached to more than two nodes. We refer to the number of nodes an edge is attached to, as its rank. Consequently, the right-hand side of a rule for a hyperedge may also have more than two external nodes. In fact, it is enforced that two edges of the same label always have the same rank. Further, if a nonterminal has edges of rank k, then the right-hand side of its associated rule must have k external nodes. Otherwise the derivation is not well defined. The use of hyperedges complicates the size definition of a graph. Consider the digram replacement in Figure 1.4b: it is the equivalent of the previously used one, now using hyperedges. Hyperedges are drawn using boxes with the label, which are connected to the nodes the edge is attached to by numbered lines. The numbers indicate the order the nodes are attached in, which is necessary to match the external nodes in order. It should be intuitively clear that this grammar cannot be

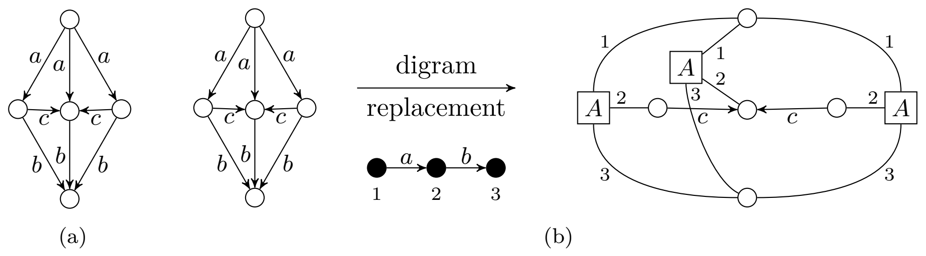


Figure 1.4: A variant of the graph in Figure 1.2 without any digrams that can be replaces by a simple edge (a), and a replacement using hyperedges of rank 3 instead (b).

considered smaller than the original graph: every "line" represents information that needs to be stored, and the graph on the right (even disregarding the additional size of the rule) has more such lines than the original graph on the left (11 vs. 8). Finding a good size definition is another challenge of our generalization. Note that these *hyperedge replacement grammars* have been well studied (proposed by [BC87, HK86, Hab92], see also e.g. [DKH97]), but not with respect to their size. A further consideration is whether to allow the rank of nonterminal edges to be unbounded. It turns out that this is generally not a good idea. In fact, it is possible to show that some graphs only compress well with a bounded maximal rank, whereas others compress better with the rank unbounded (a phenomenon already well-known for tree RePair [LMM13]). The runtime for some algorithms on the grammar might also depend on the maximal rank of a nonterminal, keeping it low is thus desirable. The experimental evaluation shows that, on the test data used, a rank of 4 gives the best, or close to the best, results.

1.2 Algorithms over Compressed Input

There is usually a trade-off between compression ratio and query performance. If a structure is compressed more, answering queries about it takes more time. In general this is true for grammars as well. We present algorithms to compute the neighbourhood of a given node without a full decompression of the graph, but the runtime of this method is proportional to the height of the grammar's derivation tree per neighbour. There are a few ways to get around this limitation for specialised queries, or for restricted grammars.

1.2.1 Constant-Delay Traversal

One way to improve query performance is to precompute a data structure of similar size to the grammar (thus still much smaller than the uncompressed data), which allows for a single traversal step, to be carried out in constant time. Such data structures are known for grammar-compressed strings [GKPS05]. The method works by essentially providing a data structure that can be used to navigate from leaf to leaf in the derivation tree, (i.e., from one symbol of the represented string to the next). This method is extended to tree grammars in [LMR16]. The right-hand sides of rules in tree grammars contain *parameter nodes* (or just parameters), which fulfil the same role as external nodes in graph grammars. During derivation they are replaced by subtrees. There exists a normal form for tree grammars that only allows four types of rules, which have at most one parameter node in their right-hand sides [LMS12]. The traversal algorithm assumes that grammars are given in this normal form. It structures the grammar: it allows to partition the derivation into a set of “spine-trees”. Such a spine consists of a line of terminals that replace the parameter nodes, and any other children are nonterminals whose right-hand sides do not have a parameter node. It is then possible to represent these spine-trees as strings using a straight-line grammar computed directly from, and not larger than, the given tree grammar. Navigating along the spine is then the same as navigating along the string, using the previous result for grammar-compressed strings. When diverting from the spine, the only action necessary is to “bookmark” the current position in the string-grammar and restart the navigation along the string rooted at a different nonterminal. Both are possible in constant time. With graphs, one problem immediately arises: when traversing a cycle, there is always at least one pair of nodes, where the traversal from the first to the second node does not represent a step from one leaf to the next in the derivation tree. Instead, a larger jump within the derivation tree is necessary. This makes it unlikely that the same method can be generalized to graphs. For a restricted set of grammars we present a precomputed pointer structure, which allows for constant-delay traversal on the represented graphs. Notably, this also includes tree grammars of any parameter size, but needs more memory than the method in [LMR16]. We give the general idea of our method assuming an input grammar G of rank k. For every node x within the grammar and every alphabet symbol σ, the method precomputes the node y (which may be in a different rule) that represents the σ-neighbour of x. This is enough to find the correct node, but without the path in the derivation tree to get from the rule containing x to the rule containing y, there is no way to determine the value of y within $\mathsf{val}(G)$. Thus, the method also precomputes a “tableau” which represents this path in the derivation tree. Such a tableau is an $h + 1 \times k$-matrix where h is the length of the path in the derivation tree, the tableau represents. It contains a set of pointers, which map every external node on the path to their corresponding internal node. These pointers are structured in such a way that two or more tableaux can be combined to represent longer paths within the derivation tree in constant time (specifically in time $O(k)$, which we assume to be fixed). This prevents an exponential amount of precomputation, as only tableaux for immediate steps have to be precomputed. The only assumption on the grammar the method therefore makes, is that its rank is bounded by a constant. No other normal form is necessary.

1.2.2 Speeding Computation up

An important property of grammar-based compression is that some queries can be answered *faster* on a grammar than on the explicit representation. One of the first such methods was a pattern-matching algorithm for LZW-compressed files [ABF96]. This was later also extended to LZ77-compressed strings [FT98], which provide better compression ratios. Generally, membership to any given deterministic finite automaton can be checked using grammar-compressed strings as input in time $O(|G|n)$, where $|G|$ is the size of the given grammar and n the number of states of the automaton [PR99]. The method to do so exploits the hierarchical nature of a straight-line grammar. The automaton is evaluated for every right-hand side separately starting at every one of its n states. This is done bottom-up on the grammar's hierarchy, such that when encountering a nonterminal, the previous computation is reused. While no generally accepted automata model for graphs exists, the same method can still be applied to straight-line graph grammars for specific queries. We take a closer look at *regular path queries*. Such a query consists of a regular expression α, a graph g and two nodes s, t of g. The question is, whether there exists a path from s to t in g such that the edge labels on the path, seen as a string, match α. It is commonly solved [Woo12] by computing the nondeterministic finite automaton $\mathcal{A}_\alpha$, which decides the same language that α matches. Then the graph g is considered to be an automaton itself, with starting state s and final state t. Now the product of $\mathcal{A}_\alpha$ and g deciding their intersection is constructed. There exists a path matching α if and only if the language of this product automaton is non-empty, which can be checked with a reachability test. Interestingly, the same method can be applied to grammar-compressed graphs, by combining a grammar G and an automaton $\mathcal{A}_\alpha$ to achieve a new grammar G', which generates the product automaton of $\mathcal{A}_\alpha$ and $\mathsf{val}(G)$ but is only of size $O(|G||\mathcal{A}_\alpha|)$. A reachability test can then be used to check (also in $O(|G||\mathcal{A}_\alpha|)$ time) for emptiness, like with the explicit representation above.

For grammar-compressed strings it is possible to check in polynomial time, whether two different grammars produce the same string, as was independently proved in [HJM96, MSU97, Pla94]. This result can be extended to ordered trees, as it is possible to convert a tree grammar into a string grammar that represents a canonical traversal of the tree. For strings and ordered trees, equality is the same as isomorphism. This is not the case for graphs or unordered trees, and in both cases the question of isomorphism is usually more interesting, as isomorphic structures often can be seen as semantically equivalent. However, even the complexity of the isomorphism problem for grammar-compressed unordered trees was unknown. It is shown in this thesis that testing two tree-generating grammars for isomorphism is indeed possible in polynomial time. The proof uses the technique of canonical ordering previously used in [Lin92] to show that isomorphism of (uncompressed) trees can be decided using logarithmic space. A major problem is that the rules of a tree grammar cannot always be canonically ordered as the specific position of the parameter node is unknown. To overcome this problem the proof reuses a central idea of the constant-delay traversal method in [LMR16]. Namely that tree grammars can be brought into a normal form, which partitions the tree into "spines" that can be derived separately. This is important because, while this spine may be of exponential length, it can be shown that only a polynomial amount of different re-orderings are necessary along the nonterminals on the spine. This argument then inductively applies to the entire derivation tree.

1.3 Outline of this Thesis

The thesis is structured similarly to this introductory chapter. Chapter 2 covers some preliminaries to introduce notation and definitions used throughout the thesis. Then the gRePair compressor is introduced in Chapter 3. This consists of two main parts: the first introduces the algorithm, explains the encoding of the final grammar, and discusses some theoretical results about its parameters, its relation to RePair compression of strings and trees, and the choices of formalism made. Afterwards a prototype implementation is experimentally evaluated and compared to existing compressors. This shows that gRePair is able to compress some graphs much better than other compressors. After this, Chapters 4 and 5 discuss algorithmic considerations. Chapter 4 first introduces general algorithms to perform neighbourhood queries on a graph grammar, with runtime proportional to the grammar's height. Then a data structure is presented which enables constant time traversal steps on a restricted set of grammars. Finally Chapter 5 introduces speed-up algorithms for directed reachability, regular path queries, and to test isomorphism of two given straight-line tree grammars. Every chapter (except for the preliminaries) ends with an overview of literature related to the content of that chapter.

Some work presented in this thesis has been previously published. Specifically, Section 5.3 was published at the International Colloquium on Automata, Languages and Programming (ICALP) [LMP15] and much of Chapter 3 and Section 5.1 was published at the International Conference on Data Engineering (ICDE) [MP16]. Furthermore, the contents in Chapter 3 and Chapter 5 (excluding Section 5.3) are currently under review for publication in *Information Systems* (preprint available at [MP17]).

Chapter 2

Preliminaries

We denote the set of natural numbers by $\mathbb{N} = \{1, 2, \ldots\}$ and by $\mathbb{N}_0$ the set $\mathbb{N} \cup \{0\}$. For $k \geq 0$ we denote by $[k]$ the set $\{1, \ldots, k\}$ and by $[0, k]$ the set $[k] \cup \{0\}$. An *alphabet* Δ is a finite, non-empty set of symbols. A *ranked alphabet* consists of an alphabet Σ together with a mapping $\mathsf{rank} : \Sigma \to \mathbb{N}_0$ that assigns a rank to every symbol in Σ. Occasionally we may use rank_Σ to refer to the rank-mapping of the ranked alphabet Σ specifically, if otherwise this would be ambiguous. For the remainder of this thesis, let $\Sigma = [\mu]$ for some $\mu \in \mathbb{N}$ be a ranked alphabet, and $\Delta = [\nu]$ for some $\nu \in \mathbb{N}$ an alphabet, unless otherwise specified. Examples use lower-case letters $a, b, c, \ldots$ (possibly with indices) as elements of either of these alphabets instead of numbers, to improve readability.

We sometimes use matrices as underlying structures. For an $m \times n$ matrix a we denote by $a(i, j)$ (with $i \in [m]$ and $j \in [n]$) the element in the i-th column and j-th row. Similarly, for an n-tuple t we denote its i-th element by $t(i)$ (for $i \in [n]$).

2.1 Strings, Trees, and Graphs

We first define the basic data structures used throughout this thesis. Let Δ be an alphabet.

Definition 2.1. A *string* is an ordered sequence $u = a_1 a_2 \cdots a_k$ ($a_i \in \Delta$ for $i \in [k]$) of symbols from Δ.

We denote the empty string by ε, and for two strings u, v the concatenation of u and v by $u \cdot v$, or uv if that is unambiguous. The size of such a u (as in Definition 2.1) is defined as $|u| = k$ with $|\varepsilon| = 0$, and we write $a_i \in u$ to express that a_i is part of the string u. Any string $v = a_i a_{i+1} \cdots a_j$ for some $i, j \in [k]$ is called a *substring* of u, and we explicitly denote such a substring by $u[i..j]$ with $u[i..0] = \varepsilon$. We use the shorthand $u[i]$ for $u[i, i]$. For a set of strings X we denote by $X^* = \{u_1 \cdots u_n \mid n \in \mathbb{N}_0 \text{ and } u_i \in X \text{ for all } i \in [n]\}$ the set of all strings that can possibly be obtained by concatenating strings from X. We also let $X^+ = X^* \setminus \{\varepsilon\}$. Let $<_\Delta$ be an order on Δ. We define the *lexicographical ordering* $<_{\mathsf{lex}}$ on equal-length strings $u, v \in \Delta^*$

as: $u <_{\mathsf{lex}} v$ if and only if there exist $p, u', v' \in \Delta^*$ and letters $a, b \in \Delta$ with $a <_\Delta b$ such that $u = pau'$ and $v = pbv'$.

Definition 2.2. *Trees* over a (possibly ranked) alphabet Σ are recursively defined as the smallest set of strings T, such that for $k \geq 0$, if $t_1, \ldots, t_k \in T$, then $\sigma(t_1, \ldots, t_k) \in T$, provided that $\sigma \in \Sigma$ and $\mathsf{rank}(\sigma) = k$, if Σ is ranked.

We speak of ranked trees if Σ is ranked and unranked trees otherwise. We denote by T_Σ the set of all such trees with labels from Σ. For two trees $t_1, t_2 \in T_\Sigma$ we call t_2 a *subtree* of t_1, if t_2 is a substring of t_1. The nodes of a tree t can be addressed by their *Dewey addresses* $D(t)$. For a tree $t = \sigma(t_1, \ldots, t_k)$, $D(t)$ is recursively defined as $D(t) = \{\varepsilon\} \cup \bigcup_{i \in [k]} i.D(t_i)$. Thus ε addresses the root node, and $u.i$ the i-th child of u. Again we write uv for $u.v$ if that is unambiguous. For $u \in D(t)$ we denote by $t[u] \in \Sigma$ the symbol at u. The size of t is $|t| = |D(t)| - 1$, i.e., the number of edges in t, and the height of a tree is defined by $\mathsf{height}(t) = \max\{|u| + 1 \mid u \in D(t)\}$. The rank of a node u is its number of children.

Unranked trees can be converted into ranked trees using a different alphabet. Let Δ be an unranked alphabet and t a tree from T_Δ. Let r be the highest rank of any node appearing in t. Finally, let $\Sigma = \{\delta_i | \delta \in \Delta \text{ and } 0 \leq i \leq r\}$ with $\mathsf{rank}(\delta_i) = i$ for any symbol $\delta \in \Delta$ be a ranked alphabet based on Δ. Then we can define $t' \in T_\Sigma$ by using the appropriate symbol from Σ for every node. For example if a node u in t is labelled δ and has 3 children, then the node u in t' is labelled δ_3. Thus we achieve a ranked tree representing the same information.

Definition 2.3. A *hypergraph over* Σ (or simply *graph*) is a tuple $g = (V, E, \mathsf{att}, \mathsf{lab}, \mathsf{ext})$ where $V \subset \mathbb{N}$ is a finite set of nodes, E is a finite set of edges, $\mathsf{att} : E \to V^+$ is the attachment mapping, $\mathsf{lab} : E \to \Sigma$ is the label mapping, and $\mathsf{ext} \in V^*$ is a string of *external nodes.*

We define the rank of an edge as $\mathsf{rank}(e) = |\mathsf{att}(e)|$ and require that $\mathsf{rank}(e) = \mathsf{rank}(\mathsf{lab}(e)) \geq 1$ for every edge e in E. The latter requirement (and the attachment mapping mapping to V^+) mean that we do not allow edges that are not attached to any nodes. We add the following two restrictions on hypergraphs:

(C1) for all edges $e \in E : \mathsf{att}(e)$ contains no node twice, and

(C2) ext contains no node twice.

For a hypergraph $g = (V, E, \mathsf{att}, \mathsf{lab}, \mathsf{ext})$ we use $V_g, E_g, \mathsf{att}_g, \mathsf{lab}_g$, and ext_g to refer to its components. We may omit the subscript if the hypergraph is clear from context. Let x be a node of a graph g. We call x *incident* to an edge $e \in E_g$ if $x \in \mathsf{att}_g(e)$. Two distinct nodes x, y are *adjacent* to each other, if there is an edge $e \in E_g$ such that $x \in \mathsf{att}_g(e)$ and $y \in \mathsf{att}_g(e)$. We denote by $N(x) = \{y \in V_g \mid \exists e \in E_g : x, y \in \mathsf{att}_g(e)\}$ the set of adjacent nodes of x (or *neighbourhood of* x) and by $E(x) = \{e \in E_g \mid x \in \mathsf{att}_g(e)\}$ the set of edges incident with x. We consider hyperedges to be directed from the first attached node to the remaining ones. Thus we also define $N^-(x) = \{y \in V_g \mid \exists e \in E_g : \mathsf{att}_g(e) = x\alpha y\beta \text{ for some } \alpha, \beta \in V^*\}$ and $N^+(x) = \{y \in V_g \mid \exists e \in E_g : \mathsf{att}_g(e) = y\alpha x\beta \text{ for some } \alpha, \beta \in V^*\}$, the *outgoing* and *incoming neighbourhoods of* x, respectively. These can further be restricted as $N^-_\sigma(x)$ $(N^+_\sigma(x))$ which include only nodes reachable by an edge labelled σ. An edge is *simple*, if its rank equals two. A hypergraph g is simple, if all its edges are simple, and for any distinct edges e_1, e_2 of g it

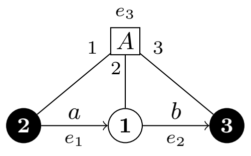


Figure 2.1: Example of a drawing of a hypergraph with three nodes (two of which external) and three edges, two of which are simple.

holds that $\mathsf{att}(e_1) \neq \mathsf{att}(e_2)$ or $\mathsf{lab}(e_1) \neq \mathsf{lab}(e_2)$, i.e., g has no "multi-edges". The rank of a hypergraph g is defined as $\mathsf{rank}(g) = |\mathsf{ext}_g|$. Nodes that are not external are called *internal*. We often refer to the node $\mathsf{ext}_g[i]$ as the *i-th external node*. We define the *node size* of g as $|g|_V = |V|$, the *edge size*[1] as

$$|g|_E = |\{e \in E_g \mid \mathsf{rank}(e) \leq 2\}| + \sum_{e \in E_g, \mathsf{rank}(e) > 2} \mathsf{rank}(e),$$

and the *size* as $|g| = |g|_V + |g|_E$. We denote the set of all hypergraphs over Σ by $\mathsf{HGR}(\Sigma)$. For two nodes $s, t \in V$ we say that there is a *path* p from s to t, if there exists a sequence of edges $p = (e_1, \ldots, e_n) \in E^n$ for some $n \in \mathbb{N}$, such that $\mathsf{att}(e_1)[1] = s$, $t \in \mathsf{att}(e_n)[2..|\mathsf{att}(e_n)|]$, and for every $i \in [n-1]$ if $\mathsf{att}(e_i) = uv_1 \cdots v_k$ then there exists a $j \in [k]$ such that $\mathsf{att}(e_{i+1}) = v_j\alpha$ for some $\alpha \in V^*$. For an edge e let $\mathsf{head}(e)$ be the first node e is attached to. For a path $p = (e_1, \ldots, e_n)$ we refer to the nodes $(\mathsf{head}(e_2), \ldots, \mathsf{head}(e_n))$ as the *nodes on the path p*. We call p internal, if all the nodes on p are internal. Note that the nodes s and t are not considered to be on the path.

Example 2.4. An example of a hypergraph can be seen in Figure 2.1. Formally, the pictured graph has $V = \{1, 2, 3\}$, $E = \{e_1, e_2, e_3\}$, $\mathsf{att} = \{e_1 \mapsto 2 \cdot 1, e_2 \mapsto 1 \cdot 3, e_3 \mapsto 2 \cdot 1 \cdot 3\}$, $\mathsf{lab} = \{e_1 \mapsto a, e_2 \mapsto b, e_3 \mapsto A\}$, and $\mathsf{ext} = 2 \cdot 3$. Note that external nodes are filled black, their position within ext is given implicitly by their number (because it is normalized, see below). Hyperedges of rank > 2 have indices indicating the order of the attached nodes, whereas simple edges are drawn directed, from their first to second attachment node.

In the following we sometimes omit the indices and the node numbers. In these cases specific order is irrelevant for the example.

A graph $g = (V, E, \mathsf{att}, \mathsf{lab}, \mathsf{ext})$ is called *normalized* if there exist $n, i \in \mathbb{N}, i \leq n$ such that $V = [n]$ and $\mathsf{ext} = i \cdot (i+1) \cdots n$ or $\mathsf{ext} = \varepsilon$. For an injective function $\rho : V_g \to \mathbb{N}$ let $\rho^* : V_g^* \to \mathbb{N}^*$ be its natural extension to strings. We call ρ a *node renaming on g* and define $g[\rho] = (\{\rho(v) \mid v \in V_g\}, E_g, \mathsf{att}^\rho, \mathsf{lab}_g, \rho^*(\mathsf{ext}_g))$, where $\mathsf{att}^\rho(e) = \rho^*(\mathsf{att}_g(e))$ for all $e \in E_g$. Note that for any graph g there exists a node renaming ρ such that $g[\rho]$ is normalized. For this reason graphs appearing in any example (including Example 2.4 above) and most other graphs in this thesis are (assumed to be) normalized. We call two graphs g and h *isomorphic* (denoted $g \cong h$), if there exists a node renaming ρ such that $V_{h[\rho]} = V_g$, $\mathsf{ext}_{h[\rho]} = \mathsf{ext}_g$, and there is a bijection $\varphi : E_h \to E_g$ such that $\mathsf{att}_g(\varphi(e)) = \rho^*(\mathsf{att}_h(e))$ and $\mathsf{lab}_g(\varphi(e)) = \mathsf{lab}_h(e)$ for every $e \in E_h$. We call h a *subgraph* of g, if $V_h \subseteq V_g$, $E_h \subseteq E_g$, $\mathsf{att}_h(e) = \mathsf{att}_g(e)$ and $\mathsf{lab}_h(e) = \mathsf{lab}_g(e)$ for every $e \in E_h$, and ext_h is the string that remains when removing every symbol in $V_g \setminus V_h$ from ext_g.

[1]This definition may seem unusual. It is motivated by the encoding we use, see Section 3.5.3 below.

Occasionally we use simple graphs without edge labels. In these cases we may denote the graph as a tuple (V, E) where V is the set of nodes and $E \subseteq V \times V$ the set of directed edges. In the above model an edge (x, y) corresponds to an edge e with $\mathsf{att}(e) = xy$ and $\mathsf{lab}(e)$ is arbitrary, but the same for every edge.

2.2 Languages and Grammars

We call a set of strings (or trees, graphs) L a *string language* (or tree language, graph language, respectively). Often we will simply say language, when the contents are clear from context. There are many formalisms to specify languages, we mainly use (context-free) grammars. Grammars are defined over a (potentially ranked) alphabet Σ, whose symbols are called *terminals*. Below we define three grammar formalisms. They all have in common that a grammar is a triple $G = (N, P, S)$, where N is a finite set of *nonterminals* with $N \cap \Sigma = \emptyset$, P is a set of rules, and S is a start symbol. A rule $p \in P$ always has the form (A, x) (usually written as $A \to x$), where $A \in N$ and x is a string, tree, or graph. We refer to the individual components of a rule as the right-hand side ($\mathsf{rhs}_G(p) = x$) and the left-hand side ($\mathsf{lhs}_G(p) = A$). For every type of grammar we define a relation $\Rightarrow_G$, which we call a *derivation step* and its transitive and reflexive closure $\Rightarrow_G^*$ called a *derivation*. We now define the specifics for each of the three grammar types.

Definition 2.5. Let Δ be an alphabet. A *context-free string grammar over* Δ is a triple $G = (N, P, S)$, where N is a finite, non-empty set of nonterminals, $S \in N$ is the start nonterminal, and $P \subseteq N \times (\Delta \cup N)^+$ is a set of rules.

For two strings $u, v \in (\Delta \cup N)^*$ we write $u \Rightarrow_G v$, if there exist $\alpha, \beta \in (\Delta \cup N)^*$, a nonterminal $A \in N$ and a rule $A \to w$ such that $u = \alpha A \beta$ and $v = \alpha w \beta$. The language of G is $L(G) = \{u \in \Delta^* \mid S \Rightarrow_G^* u\}$. We generally omit the subscript G, unless it is ambiguous. The size of a string grammar G is defined as $|G| = \sum_{A \to u \in P} |u|$.

We fix a set $Y = \{y_1, y_2, \ldots\}$ of *parameters* and for any $k \in \mathbb{N}$ we let $Y^k = \{y_1, \ldots, y_k\}$ be a ranked alphabet with $\mathsf{rank}(y_i) = 0$ for every $i \in [k]$. We denote by $T_\Delta(Y)$ the union of all sets $T_{\Delta \cup Y^k}$ for any $k \in \mathbb{N}$. For trees $t, t_1, \ldots, t_k \in T_\Delta(Y)$ we use $t[y_j \leftarrow t_j \mid j \in [k]]$ to denote the tree obtained by replacing, in parallel, every occurrence of y_j by t_j for $j \in [k]$.

Definition 2.6. Let Δ be an alphabet. A *context-free tree grammar over* Δ is a triple $G = (N, P, S)$ where N is a ranked alphabet of nonterminals, $S \in N$ is the initial nonterminal (with $\mathsf{rank}(S) = 0$), and P is a finite set of rules of the form $A(y_1, \ldots, y_k) \to t$, where $A \in N$, $\mathsf{rank}(A) = k$, and $t \in T_{N \cup \Delta \cup Y^k}$.

For two trees $t, r \in T_{N \cup \Delta}(Y)$ we write $t \Rightarrow_G r$ if t contains a subtree $A(t_1, \ldots, t_k)$, and r is obtained by replacing this subtree with $s[y_j \leftarrow t_j \mid j \in [k]]$ where $A(y_1, \ldots, y_k) \to s$ is a rule in P. A tree grammar G defines the tree language $L(G) = \{t \in T_\Delta \mid S \Rightarrow_G^* t\}$. We say a rule $A(y_1, \ldots, y_k) \to t$ has *rank* k (i.e., the rank of a rule is defined by the number of distinct parameters it has). As with string grammars, the size of a tree grammar G is defined as $|G| = \sum_{A \to t \in P} |t|$. A context-free tree grammar G is called *linear*, if for every rule $A \to t$ and

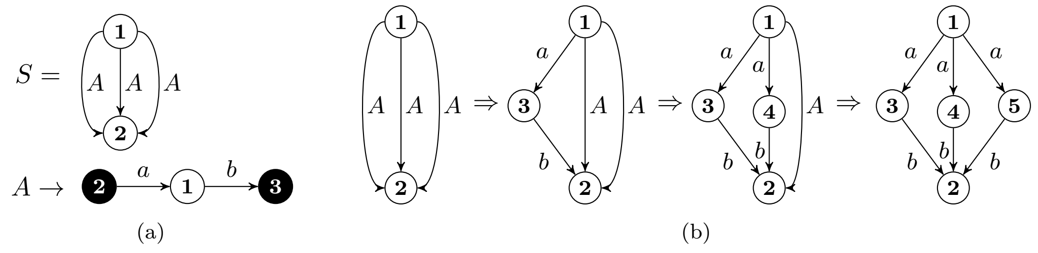


Figure 2.2: Example of a grammar (a) and its full derivation (b)

every $y \in Y$, y occurs at most once in t. Every context-free tree grammar considered in this thesis is linear.

Definition 2.7. Let Σ be a ranked alphabet. A *hyperedge replacement grammar (HR grammar) over* Σ is a tuple $G = (N, P, S)$, where $N \subset \mathbb{N}$ is a ranked alphabet of nonterminals, $P \subset N \times \mathsf{HGR}(\Sigma \cup N)$ is the set of rules such that for every $(A, g) \in P$ the graph g is normalized and $\mathsf{rank}(A) = \mathsf{rank}(g)$, and $S \in \mathsf{HGR}(\Sigma \cup N)$ is the (normalized) start graph.

The rank of an HR grammar G is defined by $\mathsf{rank}(G) = \max\{\mathsf{rank}(A) \mid A \in N\}$. An edge is called *terminal* if it is labelled by a terminal and *nonterminal* otherwise. Examples generally use upper-case letters $A, B, C, \ldots$ (possibly with indices) as nonterminals instead of numbers, to improve readability.

To define the replacement of nonterminal edges by a graph, we first introduce some notation. For a hypergraph g and an edge $e \in E_g$ we denote by $g[-e]$ the hypergraph obtained from g by removing the edge e. For two hypergraphs g, h we denote by $g \cup h$ the *union* of the two hypergraphs defined by $(V_g \cup V_h, E_g \uplus E_h, \mathsf{att}_g \uplus \mathsf{att}_h, \mathsf{lab}_g \uplus \mathsf{lab}_h, \mathsf{ext}_g)$. Note that this union

1. merges nodes that exist in both g and h,

2. creates disjoint copies of E_h, att_h, and lab_h, and

3. uses the external nodes of g only.

Now, let g, h be normalized hypergraphs and $e \in E_g$ such that $\mathsf{rank}(e) = \mathsf{rank}(h)$. Let ρ be a node renaming on h such that $\mathsf{ext}_{h[\rho]} = \mathsf{att}_g(e)$ and $\rho(v) = |V_g| + v$ for every internal node v of h. The *replacement of* e *by* h *in* g is defined as $g[e/h] = g[-e] \cup h[\rho]$. Let g be a graph with a nonterminal edge e. For two graphs $g, h \in \mathsf{HGR}(\Sigma \cup N)$ we write $g \Rightarrow_G h$ if there exists a nonterminal edge $e \in E_g$ and a rule $\mathsf{lab}(e) \to g' \in P$ such that $h = g[e/g']$. The language of an HR grammar G is defined as $L(G) = \{g \in \mathsf{HGR}(\Sigma) \mid S \Rightarrow^* g\}$. In line with the size definition of hypergraphs, we define two sizes for an HR grammar G: the node size is defined as $|G|_V = |S|_V + \sum_{A \to g \in P} |g|_V$. Analogously the edge size is defined as $|G|_E = |S|_E + \sum_{A \to g \in P} |g|_E$ and the total size $|G| = |G|_V + |G|_E$.

Example 2.8. Figure 2.2 revisits the example grammar from Chapter 1, but now with the node-IDs added in. It also shows how the node-IDs are chosen during the derivation. Note that this is the only derivation of this grammar, as it has only one rule and deriving the nonterminal edges in a different order will still yield the same graph up to a bijective renaming of the edges.

Comparing the three grammar formalisms, we can see that external nodes of HR grammars and parameters in tree grammars perform the same role: both define how the right-hand side of a rule is combined with the structure during a derivation step. The specific method however, is quite different: parameter nodes in a tree grammar are *replaced* by another tree, which is then connected at the root. External nodes on the other hand are *merged* with the nodes a nonterminal edge attaches to. There is another graph grammar formalism – node replacement grammars – which uses a replace/connect-method of derivation. We discuss its relation to HR grammars with respect to compression and why we choose the latter in Section 3.5, but already mention here that this leads to a more expressive grammar formalism. Finally note that string grammars have no notation equivalent to parameters or external nodes. This is due to strings being a simpler structure which is always ordered from left to right. This order is naturally used during a derivation step.

2.3 Straight-Line Grammars

Grammars generally generate languages, i.e., sets containing many (possibly infinitely many) strings (or trees/graphs). For grammar-based compression, this is not practical. We instead need a restriction on grammars that ensures the grammar can only generate exactly one string (tree/graph). A grammar like this can be generated by the compression algorithm. The *straight-line (SL)* property ensures exactly this and is purely syntactic, meaning it can be checked for a grammar whether it holds. A grammar $G = (N, P, S)$ is straight-line, if it is deterministic and acyclic. Deterministic means, that for every $A \in N$ there is exactly one rule $p \in P$ with $\mathsf{lhs}(p) = A$. We define acyclicity separately for each grammar formalism, as the relation is occasionally used in other contexts. Informally it means that, starting from a nonterminal A, it is not possible to derive a structure again containing A. We formally define straight-line variants for the three grammar formalisms defined in Section 2.2.

Definition 2.9. Let $G_s = (N_s, P_s, S_s)$, $G_t = (N_t, P_t, S_t)$, and $G_g = (N_g, P_g, S_g)$ be a string, tree, and HR grammar, respectively. We call

- G_s a *straight-line program (SLP)* if it is deterministic, and the relation

  $$\leq^s_{\mathsf{NT}} = \{(A_1, A_2) \mid \exists (A_1, u) \in P_s : A_2 \in u\}$$

  is acyclic,

- G_t a *straight-line tree grammar (SLT grammar)* if it is linear, deterministic, and the relation

  $$\leq^t_{\mathsf{NT}} = \{(A_1, A_2) \mid \exists (A_1, t) \in P_t : A_2 \in t\}$$

  is acyclic, and

- G_g a *straight-line hyperedge replacement graph grammar (SL-HR grammar)* if it is deterministic, and the relation

  $$\leq^g_{\mathsf{NT}} = \{(A_1, A_2) \mid \exists h : (A_1, h) \in P_g, \exists e \in E_h : \mathsf{lab}_h(e) = A_2\}$$

is acyclic.

We further require such grammars to have no useless (unreachable) rules, i.e., for every nonterminal $A \in N$ there exists at least one nonterminal $B \in N$ such that (B, A) is a pair in $\leq^s_{\mathsf{NT}}$ (or $\leq^t_{\mathsf{NT}}/\leq^g_{\mathsf{NT}}$, respectively). The exception is S, which is in turn required to appear as (S, A) for at least one nonterminal $A \in N$ in $\leq^s_{\mathsf{NT}}$ (or $\leq^t_{\mathsf{NT}}$).

For any straight-line grammar $G = (N, P, S)$ we let $\mathsf{rhs}(A) = x$ for any nonterminal $A \in N$ where $(A, x) \in P$, since the right-hand side for a nonterminal is unique in straight-line grammars. SL-HR grammars differ slightly from SLPs/SLTs in that S is a graph instead of a nonterminal. By convention, whenever we state something over all right-hand sides of an SL-HR grammar, this includes the start graph (i.e., S is assumed to be implicitly in N and $\mathsf{rhs}(S) = S$). The height of an SL-grammar $\mathsf{height}(G)$ is defined as the longest distance of any two nonterminals in their respective $\leq_{\mathsf{NT}}$-relation. Note that, for SLPs G_s and SLTs G_t, their languages $L(G_s)$ and $L(G_t)$ only contain exactly one element. We refer to this element by $\mathsf{val}(G_s)$ and $\mathsf{val}(G_t)$, respectively. For an SL-HR grammar G_g, this is slightly more involved: Depending on the order the nonterminals are derived in, the node numbers in the result change and thus $L(G_g)$ contains many (isomorphic) graphs. To solve this, we fix this derivation order: For a graph g let E^{nt}_g be the set of nonterminal edges in g. We define the *sibling-tuple* $\mathsf{sib}(E^{\mathsf{nt}}_g)$ as a tuple $(e_1, \ldots, e_n)$ such that $\{e_1, \ldots, e_n\} = E^{\mathsf{nt}}_g$, and if $i < j$ (for $i, j \in [n]$), then

- $\mathsf{att}_g(e_i) <_{\text{lex}} \mathsf{att}_g(e_j)$ (here $<_{\text{lex}}$ is the lexicographical order), or
- $\mathsf{att}_g(e_i) = \mathsf{att}_g(e_j)$ and $\mathsf{lab}_g(e_i) \leq \mathsf{lab}_g(e_j)$.

Note that (e_1, e_2) and (e_2, e_1) are both possible sibling tuples of a graph with $\mathsf{att}(e_1) = \mathsf{att}(e_2)$ and $\mathsf{lab}(e_1) = \mathsf{lab}(e_2)$, but the definition of $\mathsf{sib}(E^{\mathsf{nt}}_g)$ picks one arbitrarily. We now define $\mathsf{val}(A)$ for every nonterminal A, and define $\mathsf{val}(G)$ as $\mathsf{val}(S)$. Simultaneously, we define the *derivation-tree* $\mathsf{dt}(A)$ of A, and the derivation-tree of G ($\mathsf{dt}(G)$) as $\mathsf{dt}(S)$. Note that $\mathsf{dt}(A)$ is a ranked tree with nodes labelled by rules of G; the number of children of a node is equal to the number of nonterminal edges in the right-hand side of the rule in the label. Let $g = \mathsf{rhs}(A)$. If $E^{\mathsf{nt}}_g = \emptyset$ then $\mathsf{val}(A) = g$ and $\mathsf{dt}(A) = (A \to g)$ (that is, a single leaf, labelled by the rule $A \to g$), otherwise let $\mathsf{sib}(E^{\mathsf{nt}}_g) = (e_1, \ldots, e_n)$ and define $\mathsf{val}(A) = g[e_1/\mathsf{val}(\mathsf{lab}_g(e_1))] \cdots [e_n/\mathsf{val}(\mathsf{lab}_g(e_n))]$ and $\mathsf{dt}(A) = (A \to g)(\mathsf{dt}(\mathsf{lab}_g(e_1)), \ldots, \mathsf{dt}(\mathsf{lab}_g(e_n)))$.

For a derivation tree $s = \mathsf{dt}(A)$ we refer to a node u of s as an *s-context.* For any s-context u we denote by $g_{s,u}$ the graph $\mathsf{rhs}(s[u])$. If $s = \mathsf{dt}(S)$ we omit the s-prefix and just say context and g_u. For a nonterminal A we let $\mathsf{nodes}(A)$ be the number of internal nodes in $\mathsf{val}(A)$. For a derivation tree $s = \mathsf{dt}(A)$ and an s-context u we recursively define $\mathsf{first}_s(u)$ as $\mathsf{first}_s(u) = 0$ if $u = \varepsilon$, and if $u = v.i$ for some $i \in \mathbb{N}$ and $\mathsf{sib}(E^{\mathsf{nt}}_{g_{s,u}}) = (e_1, \ldots, e_n)$ then we define

$$\mathsf{first}_s(u) = \mathsf{first}_s(v) + \sum_{j<i} \mathsf{nodes}(\mathsf{lab}(e_j)) + |V_{g_{s,v}}| - |\mathsf{ext}_{g_{s,v}}|.$$

Now any internal node x in $g_{s,u}$ becomes within $\mathsf{val}(A)$ the node $\mathsf{node}_s(u, x) = \mathsf{first}_s(u) + x$. Again if $s = \mathsf{dt}(S)$ we omit the subscript. For a context u and an internal node x of g_u, the tuple (u, x) can therefore be seen as a representation of the node $\mathsf{node}(u, x)$ within $\mathsf{val}(G)$. We refer to such a tuple as a *G-representation* of $\mathsf{node}(u, x)$.

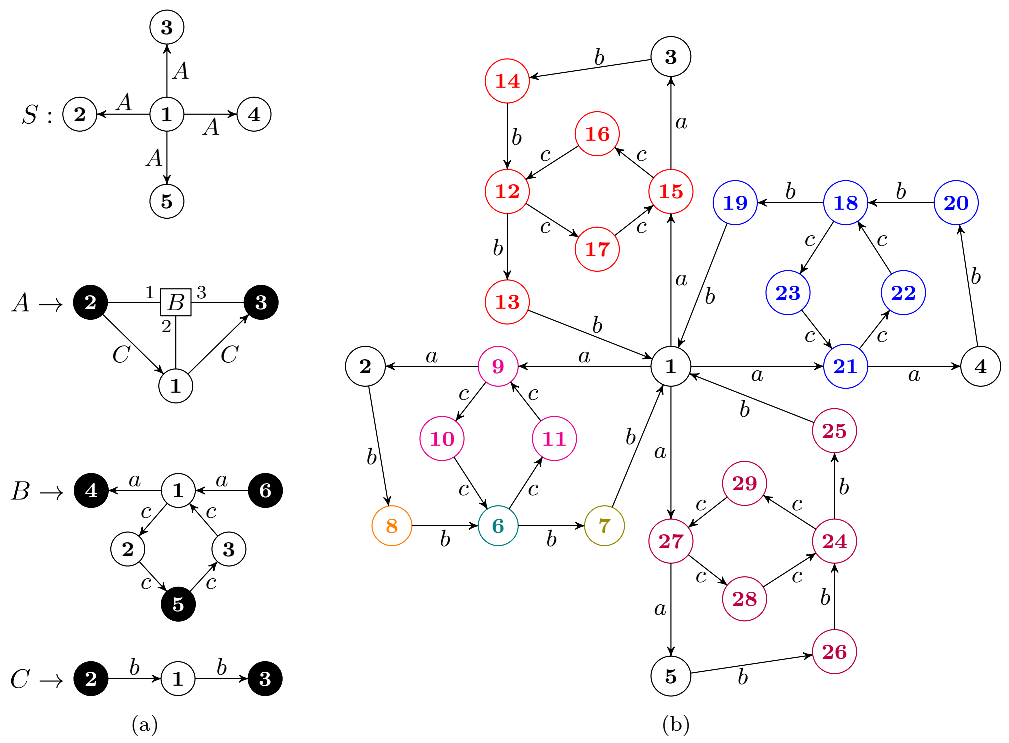


Figure 2.3: Example of an SL-HR grammar G (a), and the graph val(G) it represents (b). The graph val(G) is created according to dt(G) given in Figure 2.4. The nodes in this figure and in the derivation tree in Figure 2.4 are coloured to show where in dt(G) they originate from.

Example 2.10. A full example for an SL-HR grammar is given in Figure 2.3a. It has the derivation tree shown in Figure 2.4, and thus represents the unique graph val(G) given in Figure 2.3b. Figure 2.4 also includes an example for the computation of a node-ID. To expand on it, the precise computation of first(u_3) is

$$\begin{aligned}
\mathsf{first}(u_3) &= \mathsf{first}(u_2) + |V_{\mathsf{rhs}(A)}| - |\mathsf{ext}_{\mathsf{rhs}(A)}| + \mathsf{nodes}(C)\\
&= \mathsf{first}(u_1) + |V_S| - |\mathsf{ext}_S| + |V_{\mathsf{rhs}(A)}| - |\mathsf{ext}_{\mathsf{rhs}(A)}| + \mathsf{nodes}(E)\\
&= 0 + 5 - 0 + 3 - 2 + 1 = 7.
\end{aligned}$$

And therefore node 1 in rhs(C) represents the node first$(u_3) + 1 = 8$ in val(G) (compare to Figure 2.3b to confirm).

2.4 Encoding Strings and Trees as Graphs

Strings and trees can be conveniently encoded as hypergraphs. As hypergraphs are more complex, this affects the size. We make use of these encodings, when discussing the relation of graph compression to previously proposed string- and tree-compression in Section 3.4. Let $w = a_1 \cdots a_n$ be a string. We define the hypergraph s-graph(w) representing the string w as $(\{0, 1, \ldots, n\}, E_w, \mathsf{att}_w, \mathsf{lab}_w, \mathsf{ext}_w)$, where $E_w = \{e_1, \ldots, e_n\}$. For an edge $e_i \in E_w$ we let $\mathsf{att}_w(e_i) = i - 1 \cdot i$ and $\mathsf{lab}(e_i) = w_i$. Finally the external nodes are $\mathsf{ext}_w = 0 \cdot n$. The size of

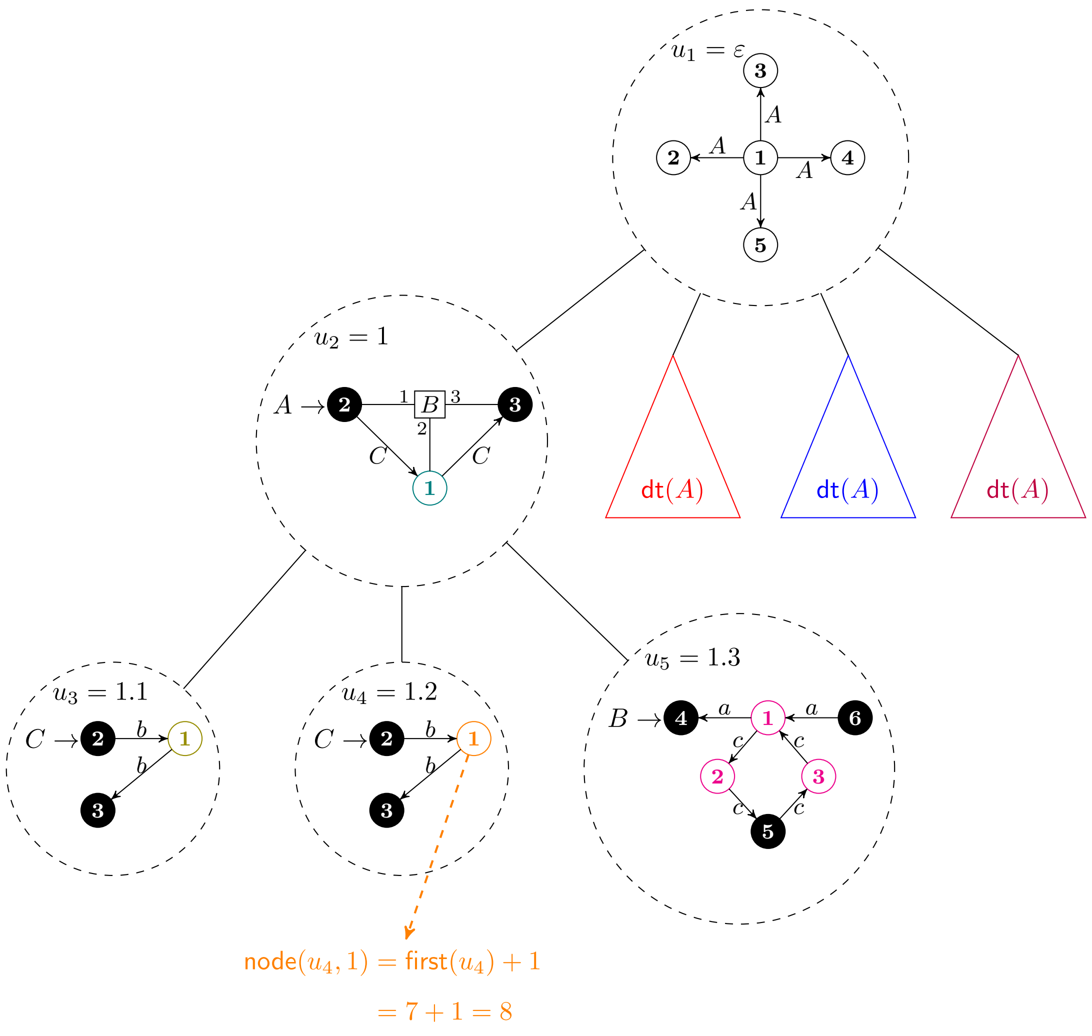


Figure 2.4: Every node of this derivation tree contains its respective context. The graph represented by this derivation tree is shown in Figure 2.3b. Nodes in both figures are coloured to show their origin.

$\mathsf{s\text{-}graph}(w)$ is $2n + 1 = 2|w| + 1$. Note that the external nodes indicate beginning and end of the string. Figure 2.5a shows $\mathsf{s\text{-}graph}(w)$ for $w = abca$.

The encoding for trees is a little less straight-forward, because the node labels are moved into the edges, and the children of a node in a tree are ordered. We convert a node with k children into a hyperedge of rank $k + 1$ (as it is connected to the parent as well). The order of the hyperedge retains the order of the children and the parent is always located at the first node attached to a hyperedge. This is a well-known encoding, see e.g. [EH92]. Note that graphs are always ranked and thus this encoding requires a ranked tree. As discussed in Section 2.1 above, every unranked tree can be converted into a semantically equivalent ranked tree by extending the alphabet. Let t be a ranked tree with parameters $y_1, \ldots, y_k$, none of which appear more than once in t, and let $<_t$ be the lexicographical order on $D(t)$. For an address $i \in D(t)$ we denote by $\mathsf{pos}(i)$ the position of i within the order $<_t$, such that $\mathsf{pos}(\varepsilon) = 0$. Let $\mathsf{pos}(Y) = \{i \mid t[\mathsf{pos}(i)]$ is a parameter$\}$ and let $w(Y) = i_1 \cdots i_k$ be the string of positions in $\mathsf{pos}(Y)$ in ascending order.

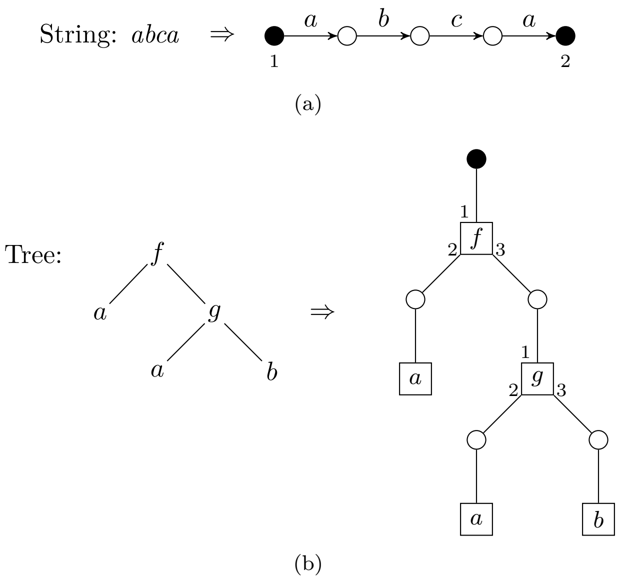


Figure 2.5: Example of a string-graph (a), and a tree-graph (b).

We define the hypergraph $\mathsf{t\text{-}graph}(t)$ representing the tree t as $(\{0,1,\dots,|D(t)|\}, \{e_i \mid i \in D(t)$ and $t[i]$ is not a parameter$\}, \mathsf{att}, \mathsf{lab}, 0 \cdot w(Y))$, where for $i \in D(t)$ and $k = \mathsf{rank}(t[i])$,

- $\mathsf{att}(e_i) = (\mathsf{pos}(i)) \cdot (\mathsf{pos}(i)+1) \cdots (\mathsf{pos}(i)+k)$, and
- $\mathsf{lab}(e_i) = t[i]$.

The size of $\mathsf{t\text{-}graph}(t)$ is between $2|t|+2$ (for a monadic tree t) and $3|t|+2$ (for a tree where every inner node has rank greater than 1). Figure 2.5b shows $\mathsf{t\text{-}graph}(t)$ for $t = f(a, g(a,b))$.

We can also encode a string- or tree grammar as an HR graph grammar. Using the above encodings this is a straight-forward process. Let $G_s = (N_s, P_s, S_s)$ be a string grammar, then the grammar $\mathsf{s\text{-}grammar}(G_s) = (N_s, P, S)$ is the grammar with $P = \{A \to \mathsf{s\text{-}graph}(w) \mid A \to w \in P_s\}$ and $S = (\{1,2\}, \{e\}, \mathsf{att}(e) = 1 \cdot 2, \mathsf{lab}(e) = S_s, \mathsf{ext} = 1 \cdot 2)$. This way $L(\mathsf{s\text{-}grammar}(G_s)) \cong \{\mathsf{s\text{-}graph}(w) \mid w \in L(G_s)\}$. The sets are isomorphic, because the nodes of the graphs in $L(\mathsf{s\text{-}grammar}(G))$ will have different IDs than the nodes of $\mathsf{s\text{-}graph}(w)$ for $w \in L(G_s)$.

Similarly, we encode a tree grammar $G_t = (N_t, P_t, S_t)$ as $\mathsf{t\text{-}grammar}(G_t) = (N, P, S)$ with $N = N_t$, but $\mathsf{rank}_N(A) = \mathsf{rank}_{N_t}(A) + 1$ for every $A \in N$, $P = \{A \to \mathsf{t\text{-}graph}(t) \mid A \to t \in P_t\}$ and $S = (\{1,\dots,k\}, \{e\}, \mathsf{att}(e) = 1 \cdots k, \mathsf{lab}(e) = S, \mathsf{ext} = 1)$, where $k = \mathsf{rank}_N(S_t)$. Thus $L(\mathsf{t\text{-}grammar}(G_t)) \cong \{\mathsf{t\text{-}graph}(t) \mid t \in L(G_t)\}$, with the isomorphism again due to different node-IDs.

2.5 RDF Graphs

A particular type of graph mainly used in the experimental evaluation in Section 3.6, are RDF graphs. The Resource Description Framework (RDF) is a fairly recent specification, first standardized by the W3C in February 2004. The current version is RDF1.1[2] from February 2014. It is used to represent linked data and semantic information. Its relationship to graphs is similar to XML's relationship to trees, in that graphs are a natural representation of the structure defined by RDF. Roughly speaking, RDF data is represented as a set of triples (s, p, o), connecting a subject s with an object o by a predicate p. Notably, the domains for these can overlap to some degree (for example, values used as predicates or objects in some triples may be subjects in others). Such a set of triples can be represented by having nodes for the subjects and objects, and edges for the predicates. Thus (s, p, o) becomes an edge from s to o labelled p. Note that while the standard allows for predicates to also appear as subjects this is unusual. Even if it happens, it is not necessary to model overlapping domains as edges pointing to other edges. Instead, there may be a node label that also appears as an edge label. As RDF graphs encode semantic information, the concrete values for subjects, predicates, and objects can be long strings (for example URIs). It is a common practice (see e.g. [FGM10, MFC12, ÁBF$^+$15]) to map the possible values to integers using a dictionary and to represent the graph using triples of integers. We also use this encoding.

2.6 Computational Model and Direct Graph Representations

Our computational model is a word RAM model, where registers have a fixed bit length β. The space of a data structure is measured by the number of registers it uses. Registers are addressed by numbers, which we refer to as pointers, and can be randomly accessed in constant time. Arithmetic operations (we only need addition) and comparisons of the contents of two registers can be carried out in $O(1)$.

We present some direct (non-succinct) data structures to represent graphs $g = (V, E, \mathsf{att}, \mathsf{lab}, \mathsf{ext})$ with $V = \{1, \ldots, n\}$ and $E = \{e_1, \ldots, e_m\}$ as a baseline. For directed simple graphs without edge-labels there are essentially two widely known representations: adjacency lists and adjacency matrices. An adjacency list maps each node to the set of nodes it is adjacent to. For normalized graphs, it uses $O(\log_2(n)(n + m))$ bits of space and the question "is u an out-neighbour of v?" (an *adjacency query*) can be answered in $O(\log d)$ time, where d is the number of neighbours of v using a binary search (assuming the lists are ordered by the IDs). An adjacency matrix on the other hand is a $n \times n$ matrix a where $a(i, j) = 1$ if and only if there is an edge from i to j, every other entry in a is 0. This needs $O(n^2)$ bits of space, but adjacency queries can be done in constant time. Note that an adjacency matrix cannot represent parallel edges (i.e., more than one edge connecting the same two nodes), but multiple matrices can be used to represent the parallel edges.

There are no such widely used representations for edge-labelled graphs and/or hypergraphs as used here. We present some direct generalisations of the above methods. An adjacency list could include labels by mapping nodes to tuples (u, l) where u is the neighbouring node and l is the edge label. This increases the space used to $O(\log_2(n)(n + m) + \log_2(|\Sigma|)m)$ bits. For a hyperedge of rank k, the adjacency mapping maps a node to a k-tuple containing the

[2]`https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`

$k-1$ neighbours and the label. It is sufficient to do so for the first node of every hyperedge in terms of representation (i.e., all the information to reconstruct the graph is available) using $O(\log_2(n)(n+m)k + \log_2(|\Sigma|)m)$ space. Adjacency matrices can be extended to using labels by using one matrix for every label, increasing the space needed to $O(|\Sigma|n^2)$ bits, and increasing the time for an adjacency query to $O(|\Sigma|)$ (however, it remains constant time, if only neighbours with a certain label are queried). There is no direct generalisation of adjacency matrices to hyperedges. Instead, one can use an *incidence matrix*, which is an $m \times n$ matrix a where $a(i,j) = 1$ if and only if $j \in \mathsf{att}(e_i)$. However, this keeps no information about the order of $\mathsf{att}(e_i)$. One way to keep this order is to extend the matrix, such that $a(i,j) = x$ if and only if $\mathsf{att}(e_i)[x] = j$. Let r be the maximal rank appearing in g, then this needs $O(\log_2(r)|E||V||\Sigma|)$ bits of space. Another option is to store the order separately for every edge in another mapping $\mathsf{p} : E \to [r]^r$, which lists the positions of the nodes relative to their order as natural numbers. As an example: if the edge e_i has $\mathsf{att}(e_i) = 3 \cdot 6 \cdot 1$ and $r = 3$, then the relevant row of the incidence matrix has $a(i,1) = a(i,3) = a(i,6) = 1$ and $a(i,j) = 0$ for $j \notin \{1,3,6\}$, and $\mathsf{p}(e) = (2,3,1)$. This way we need $O(|E||V|\Sigma + |E|\log_2(r)^2)$ bits of space. A neighbourhood-query on an incidence matrix needs $O(|E|)$ time, as the check whether both $a(i,u) = 1$ and $a(i,v) = 1$ needs to be made for every edge.

Remark. The times given above for adjacency queries are based on relatively naïve approaches and do not consider whether there are data structures that can significantly improve the runtime.

2.7 Necessary Complexity Theoretic Terms

The present work is not complexity theoretic in nature, but we use some well-known terms, which we briefly recap. A *decision problem* is essentially any algorithmic problem that can be answered with “yes” or “no”. For example “given a graph g, does it have a complete subgraph g' with at least k nodes?” is a decision problem, as a graph either has the property or not. Formally, a decision problem is a language L of all correct inputs, i.e., all inputs for which the answer is “yes”. An algorithm deciding this problem is therefore deciding a language membership problem. In the example above, L contains (an encoding of) every graph g which has a complete subgraph with at least k nodes. Let $f, h : \mathbb{N} \to \mathbb{R}$ be some functions. We say that a problem L can be decided in runtime $O(f(n))$ and space $O(h(n))$ if there exists an algorithm A that, given an input x correctly determines whether $x \in L$ using $O(f(|x|))$ computation steps and $O(h(|x|))$ registers of memory in addition to the input. We say that A *deterministically decides* L in $O(f(n))$ time and $O(h(n))$ space (or just A decides L). An algorithm can be non-deterministic, which means that given an input x (e.g. an encoding of a graph) and a witness y (e.g. a complete subgraph of x with k nodes), the algorithm can verify whether $x \in L$ by checking, in $O(f(|x|))$ time and $O(h(|x|))$ space, that the witness is a proof for $x \in L$. For example, an algorithm for the above problem of finding a complete subgraph with k nodes would check that the witness y is a complete graph with at least k nodes and would then verify that it indeed is a subgraph of x. However, such an algorithm can not compute the witness itself, it can only verify the correctness of a given one. Intuitively, this can be considered “guessing” a solution: the witness is guessed and then verified in polynomial time. We say A *non-deterministically decides* L in $O(f(n))$ time and $O(h(n))$ space. A *complexity class* $\mathcal{C}$ is a set of decision problems which can be decided within a certain time or space bound. There are some well-known such classes, most notably P (*polynomial time*) which contains every decision problem for which there exists an algorithm that (deterministically) decides the problem in $O(n^c)$ time for some constant c. Conversely, the class NP (*non-deterministic polynomial time*) contains every decision problem, for which there exists an algorithm that non-deterministically decides the problem in $O(n^c)$ time for some constant c. It is often assumed that $\mathsf{P} \subsetneq \mathsf{NP}$, but while $\mathsf{P} \subseteq \mathsf{NP}$ is obvious, there is no known proof for $\mathsf{P} \neq \mathsf{NP}$ at the time of writing.

For two decision problems L, L' we write $L \leq_p L'$ if there exists a function f such that

1. $f(w) \in L'$ if and only if $w \in L$, and
2. $f(w)$ can be computed in polynomial time for any input w.

Informally, this means that any instance of the problem L can be encoded into an instance of the problem L' and thus an algorithm deciding L' can also be used to decide L. We say L can be *reduced* to L'. For a complexity class $\mathcal{C}$ and a decision problem L we say L is *$\mathcal{C}$-hard* if $L' \leq_p L$ for every $L' \in \mathcal{C}$. A problem L is *complete for $\mathcal{C}$ (or $\mathcal{C}$-complete)* if it is $\mathcal{C}$-hard and in $\mathcal{C}$.

Chapter 3

RePair for Graphs

This chapter presents a way to extend the popular RePair compression scheme [LM00] to graphs using SL-HR grammars. We will first briefly recall the classic algorithm for strings and the previous extension to trees. Afterwards we discuss the extension of the algorithm to graphs, and give detailed explanations for some of the challenges unique to graphs. We then compare the result with the previous versions for strings and trees, and discuss the graph grammar formalism we use, highlighting a possible other choice. Finally we provide an experimental evaluation on a varied set of graphs based on a prototype implementation and close the chapter with a discussion on related work on graph compression and some directions for future work.

3.1 The RePair Compression Scheme

Let us first explain the classical RePair compressor for strings and trees. The RePair compression scheme is an approximation algorithm for a smallest context-free grammar for a given string. The relevant decision problem for finding a smallest grammar is the problem of deciding, given a string u and a number $k \in \mathbb{N}$, whether there exists a straight-line grammar G of size at most k generating u. It was shown by Charikar et al. [CLL$^+$05] that the smallest grammar problem is NP-complete. Hence, already for string-graphs it follows that finding a smallest grammar is an NP-hard optimization problem. Several approximations exist to work around this problem. One of these is Larsson and Moffat's RePair compression scheme [LM00]. The idea of this compressor is to repeatedly replace all (non-overlapping) occurrences of a most frequent "digram" by a new nonterminal. In a string, a digram consists of two consecutive symbols. The process ends when no repeating digrams occur. Consider as an example the string

$$u = abcabcabc.$$

It contains occurrences of the digrams ab (3 times), bc (3 times), and ca (2 times). If ab is replaced by A then we obtain $AcAcAc$. If now Ac is replaced by B, then we obtain this grammar

$$\begin{aligned} S &\to BBB \\ B &\to Ac \\ A &\to ab \end{aligned}$$

Note that the original string u has size $|u| = 9$, whereas the grammar has size 7. Note that overlapping occurrences only exist for digrams of the form aa.

To compute in linear time such a grammar from a given string requires a set of carefully designed data structures. The input string is represented as a doubly linked list. Additionally a list of active digrams (digrams that occur at least twice) is maintained. Every entry in the list of active digrams points to an entry in a priority queue Q of length $\sqrt{n}$ (with n being the length of the input string) containing doubly linked lists. The list with priority i in Q contains every digram that occurs i times, the last list contains every digram occurring $\sqrt{n}$ or more times. The list items also contain pointers to the first occurrence of the respective digram. This queue is used to find the most frequent digram in constant time. Larsson and Moffat [LM00] prove that $\sqrt{n}$-length guarantees a linear runtime for the complete algorithm. Basically, in a string of length n the most occurrences a digram can have is $n/2$ and in that case no other digram can occur as often in the string. There may, in general, be more than one digram occurring more than $\sqrt{n}$ times, but the last list can not have more than $\sqrt{n}$ entries, and the frequency of the most frequent digram is monotonely decreasing. This overall leads to a linear running time. All these data structures are updated whenever an occurrence is removed. Consider removing one occurrence of ab in the example above: when doing so, one occurrence of bc and possibly ca need to be removed from the list. On the other hand, new occurrences of Ac and possibly cA are created.

An even smaller grammar than the one above can be obtained through *pruning*, which removes nonterminals that are referenced only once, i.e., pruning would remove the nonterminal A in the grammar above, so that the B-rule becomes $B \to abc$. This reduces the size of the grammar from 7 to 6. Note that pruning can never increase the size of the grammar (but may not decrease it). Also note that sometimes it may be beneficial not to prune, as the encoding of the grammar can then be defined under the assumption that every right-hand side has size 2. This may make the encoding smaller, even though the size of the grammar is larger.

RePair was generalized to trees by Lohrey et al. [LMM13]. Here a digram consists of two nodes, and the i-edge between them, with i meaning that the second node is the i-th child of the first node, denoted by (a, i, b). Here overlapping occurrences can only happen for digrams of the form (a, i, a). A digram (a, i, b) has, in a binary tree, at most three "dangling edges". Dangling edges in context-free tree grammars are represented by parameters of the form $y_1, \ldots, y_k$. Note that only the first of the two replacements shown in Figure 3.1 make the resulting grammar smaller and would thus be considered *contributing*: while the digram represented by the B-rule occurs twice, replacing it would make the grammar larger, as the two edges removed from the tree are represented by a rule using 4 edges. This is an effect that can never happen for

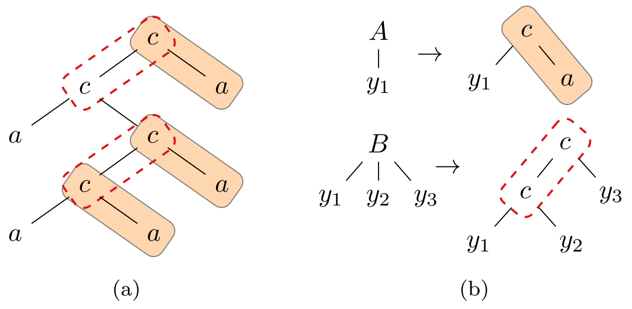


Figure 3.1: Different digrams TreeRePair considers in a tree and their replacement rules.

strings and makes the pruning step for trees (and graphs) more involved as we further discuss in Section 3.2.2 below. Replacing the digram represented by the A-rule on the other hand reduces the size of the tree by 3 and the right-hand side of the rule has size two, thus leading to a grammar that is by one smaller than the original graph. Were the digram to occur once less however, the replacement would become not contributing. The rank of a grammar also has an impact on further algorithms run on it, see e.g. [LM06, LMS12]. Keeping the rank small is thus desirable. Therefore, TreeRePair has a user-defined “maxRank” parameter.

3.2 The GraphRePair Algorithm

The first step in generalizing RePair to graphs, is defining the notion of a digram in a graph. For simple graphs two options come up naturally: two neighbouring nodes and the edge between them, or two edges with a common node. The first option is not viable, as it does not allow to compress some very basic graphs: consider a cycle C_n consisting of n consecutive edges (and n nodes). Replacing a digram of the first kind does not remove nodes nor edges (because both nodes in the digram need to be external), and only relabels edges by a nonterminal. Thus no compression is achieved. We therefore use the latter option.

Definition 3.1. A *digram* over Σ is a hypergraph $d \in \mathsf{HGR}(\Sigma)$, with $E_d = \{e_1, e_2\}$ such that

1. for all $v \in V_d$, $v \in \mathsf{att}_d(e_1)$ or $v \in \mathsf{att}_d(e_2)$,
2. there exists a $v \in V_d$ such that $v \in \mathsf{att}_d(e_1)$ and $v \in \mathsf{att}_d(e_2)$, and
3. $\mathsf{ext}_d \neq \varepsilon$.

Every possible digram over undirected, unlabelled edges is shown in Figure 3.2. Note that some digrams are grouped. This is because, one can represent one by the other by using a different order in the att-relation of the nonterminal edge. Another example for digrams are the right-hand sides of the two A-rules in Figure 3.3. Note that both grammars in the figure generate the graph on the left. However, they differ in size: the grammar in the middle has size 12, while the grammar on the right has size 9 (recall that simple edges have size 1).

As mentioned before, RePair replaces a digram that has the largest number of non-overlapping occurrences.

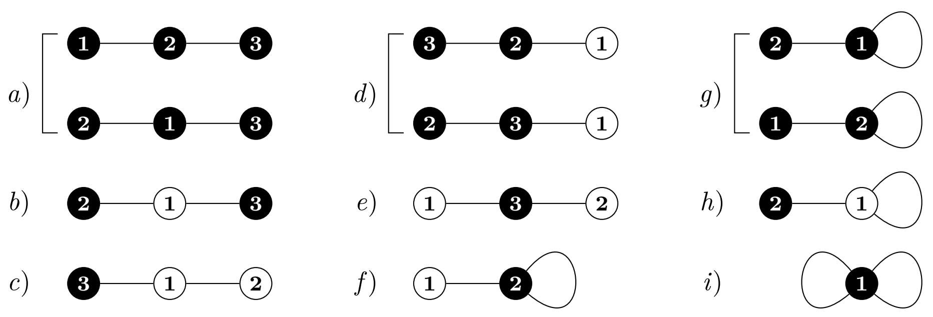


Figure 3.2: Every possible digram using two unlabelled, undirected simple edges.

Definition 3.2. Let $g, d \in \mathsf{HGR}(\Sigma)$ such that d is a digram. Let e_1^d, e_2^d be the two edges of d. Let $o = \{e_1, e_2\} \subseteq E_g$ and let V_o be the set of nodes incident with the edges in o. Then o is an *occurrence* of d in g, if there is a bijection $\psi : V_o \to V_d$ such that for $i \in \{1, 2\}$ and $v \in V_o$

1. $\psi(v) = \mathsf{att}_d(e_i^d)[j]$ if and only if $v = \mathsf{att}_g(e_i)[j]$ for $j \in [\mathsf{rank}(e_i)]$,
2. $\mathsf{lab}_d(e_i^d) = \mathsf{lab}_g(e_i)$, and
3. $\psi(v) \in \mathsf{ext}_d$ if and only if $v \in \mathsf{att}_g(e)$ for some $e \in E_g \setminus \{o\}$ or $v \in \mathsf{ext}_g$.

The first two conditions of this definition ensure that the two edges of an occurrence induce a graph isomorphic to d. The third condition requires that every external node of d is mapped to a node in g that is incident with at least one other edge. Thus, the edges marked in the graph on the left of Figure 3.3 constitute an occurrence of the digram (b) of Figure 3.2, which is the same as the one of the A-rule of the right grammar, but not an occurrence of the digram of the A-rule in the middle grammar (digram (a) of Figure 3.2). We call the nodes in V_o that are mapped to external nodes of d, *attachment nodes* of o, and the ones mapped to internal nodes, *removal nodes* of o. Two occurrences o_1, o_2 of the same digram d are called *overlapping* if $o_1 \cap o_2 \neq \emptyset$. Otherwise they are *non-overlapping.* If there are at least two non-overlapping occurrences of d in a graph g, we call d an *active digram.*

Let A be a symbol of rank k and d a digram of rank k. The *replacement of an occurrence o of d in g by A* is the graph obtained from g by removing the edges in o from g, removing the removal nodes of o, and adding an edge labelled A that is attached to the attachment nodes of o, in such a way that applying the rule $A \to d$ yields the original graph (when using ψ from Definition 3.2 as the node renaming). Consider again Figure 3.3: the start graph of the right grammar is the replacement of the shaded occurrence of d in the left graph by A (where d is $\mathsf{rhs}(A)$ of the grammar on the right).

Using these definitions, given a graph g, we define the main loop of gRePair as the steps given in Algorithm 1. After this loop finishes, the algorithm continues by connecting the disconnected components of the graph with virtual edges (identified by a special label used only for these) and runs the main loop again. Finally, the grammar is pruned and in a final step the virtual edges are removed from the grammar. This improves the compression on graphs with disconnected

Algorithm 1 Main replacement-loop of the gRePair algorithm

```
Input: Graph $g = (V, E, \mathsf{att}, \mathsf{lab}, \mathsf{ext})$
Output: Grammar $G$ with $\mathsf{val}(G) \cong g$ and $|G| \leq |g|$
 1: $N = P = \emptyset$
 2: $S \leftarrow g$
 3: $L(d) \leftarrow$ list of non-overlapping occurrences for every digram $d$ appearing in $g$
 4: while $|L(d)| > 1$ for at least one digram $d$ do
 5:     Select a most frequent digram $mfd$
 6:     $A \leftarrow$ fresh nonterminal of rank $\mathsf{rank}(mfd)$
 7:     In the graph $S$, replace every occurrence $o$ in $L(mfd)$ by a new $A$-edge
 8:     $N \leftarrow N \cup \{A\}$
 9:     $P \leftarrow P \cup \{A \to mfd\}$
10:     Update the occurrence lists
11: return $G = (N, P, S)$
```

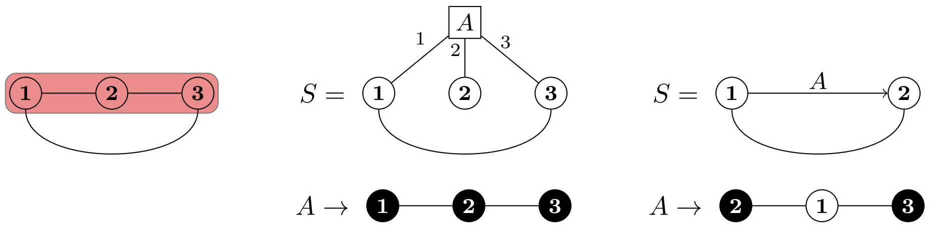


Figure 3.3: Two ways of replacing a pair of edges by a nonterminal. Only the one on the right is considered by gRePair in this case.

Algorithm 2 Full *gRePair*-Algorithm.

```
1: function GraphRePair(Graph $g = (V, E, \mathsf{att}, \mathsf{lab}, \mathsf{ext})$)
2:     $G_1 \leftarrow$run Algorithm 1 on $g$
3:     Connect disconnected components in $S_{G_1}$ with edges labelled $v \notin \Sigma \cup N_{G_1}$.
4:     $G_2 \leftarrow$run Algorithm 1 on $S_{G_1}$ (using fresh nonterminals)
5:     $G \leftarrow (N_{G_1} \cup N_{G_2}, P_{G_1} \cup P_{G_2}, S_{G_2})$
6:     Prune $G$
7:     Remove virtual edges from $G$
8:     return $G$
```

components. This leads to the full algorithm given in Algorithm 2. We now provide more details on the steps in lines 3 and 10 of Algorithm 1, and on the pruning step.

3.2.1 Occurrence Lists: Data Structures, Generation, and Updates

To keep track of all digram occurrences, the data structures used are a direct generalization to graphs of the data structures used for strings [LM00] and trees [LMM13, Figure 11]. The occurrences are managed using doubly linked lists for every active digram. The list items are also referenced from the individual edges of the graph, to allow for expected constant time access to the list of digrams an edge is occurring in. For example, if an edge e is part of an occurrence o of a digram d, then e also contains a (hashed) set containing d, and a mapping that maps d to the list-item representing o, within the occurrence list for d. Whenever an occurrence is added or removed from one of the occurrence lists, these mappings are updated in constant time. This is possible because the occurrence always contains links to the two participating edges. Of further importance is a priority queue, which uses the frequency of a digram as the priority. It is used to find the most frequent digram. Following Larsson and Moffat [LM00] the length of this queue is chosen as $\sqrt{n}$, where n is the number of edges of the original input graph to gRePair.

Generating Occurrence Lists There are two parts of the main replacement loop given in Algorithm 1, which directly involve the occurrence lists: lines 3 and 10. In the first of these lines, we aim to find a set of non-overlapping occurrences for every digram occurring in g, that is of maximal size. This can be solved in $O(|E|^4)$ time by reducing it to maximum matching (a maximal cardinality set of edges X such that no two edges $e_1, e_2 \in X$ have a common node). Let g be a graph with n nodes and m edges, and d a digram. The first step is to compute a set O containing all occurrences of d in g, including overlapping ones. This already takes $O(m^2)$ time, as $|O| \in O(m^2)$. To see this, consider a “star”; a graph with one central node connected to m other nodes, which in turn have no neighbours except the central node. There are $m(m-1)/2$ pairs of edges and thus as many overlapping occurrences of the same digram. We encode this set of occurrences into a graph g_O such that every occurrence $\{e_1, e_2\}$ in O is represented by an edge from e_1 to e_2 in g_O. Thus, g_O potentially has a node for every edge in g, and an edge for every occurrence in O. It can be shown that a maximum matching on g_O corresponds to a maximal non-overlapping subset of O. Computing a maximum matching in graphs can be done in $O(|V|^2|E|)$ by using, e.g. the Blossom-algorithm [Edm65]. As g_O has $O(m)$ nodes, and $|O|$ edges, the total running time is in $O(m^2|O|) = O(m^4)$.

As this solution is already prohibitively expensive for just one digram, we use an approximation. Let ω be an order on the nodes of g. We traverse the nodes of g in this order, and at every node iterate through occurrences centred around this node. Some care has to be taken to do this in linear time: as noted above, a centre node v of degree k has $O(k^2)$ different pairs of edges centred around the node and thus $O(k^2)$ possible occurrences. Consider for now the case of unlabelled and undirected edges. Then, after choosing one of the $O(k^2)$ occurrences and adding it to the occurrence list for a digram, the two involved edges are taken out of further consideration, because every other occurrence using one of these edges would overlap. Let E be the edges incident with v that have not been added into occurrence lists yet. We partition E into two sets $E_1 = \{e_1, \ldots, e_n\}$ and $E_2 = \{f_1, \ldots, f_m\}$ where $m - n \in \{0, 1\}$. We then add $\mathsf{Occ}(E_1, E_2) = \{\{e_i, f_i\} \mid 1 \leq i \leq n\}$ as the occurrences around v to the list. Note that only if all occurrences around v are occurrences of the *same* digram, this procedure guarantees to produce a maximum non-overlapping set of occurrences around v (even though the digrams use the same edge labels, they can still differ in the external nodes). From here, adding labels (or directions, which can be viewed as labels) is straightforward. For two labels σ_1 and σ_2 let $E_{\sigma_1,\sigma_2}(v)$ be the set of edges incident with v labelled σ_1 and not yet added to an occurrence list for a digram with an edge labelled σ_2. These sets can be precomputed for every node and

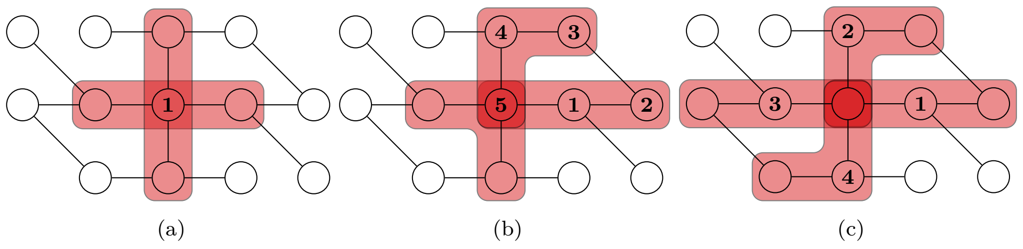


Figure 3.4: Three different traversals visiting the nodes in the numbered order to find occurrences of digram (a) of Figure 3.2. The occurrences found for each traversal are marked.

kept up to date with every replacement of an occurrence. Then for distinct symbols σ_1 and σ_2 add the occurrences $\mathsf{Occ}(E_{\sigma_1,\sigma_2}(v), E_{\sigma_2,\sigma_1}(v))$ and for every σ add the occurrences where both edges have the same symbol by splitting $E_{\sigma,\sigma}(v)$ as above. This takes $O(|\Sigma|^2)$ time (where $|\Sigma|$ is expected to be small).

The node order ω heavily influences the compression behaviour. Consider the graph in Figure 3.4. We want to find the non-overlapping occurrences of the digram a) in Figure 3.2. Note that all three nodes are external, that is, we are looking for three nodes u, v, w such that u has an edge e_1 to v, v has an edge e_2 to w, and all three nodes also have edges to other nodes (different from e_1, e_2). Figure 3.4a shows the non-overlapping occurrences found if we start in the central node of the graph. Using the DFS-type order starting at a different node given by the numbers in Figure 3.4b, three occurrences are determined. Using the "jumping" order in Figure 3.4c, a maximum set of four non-overlapping occurrences is found. As any order that traverses along the edges has to visit the centre node before all occurrences are found, such a jumping order is necessary to find all occurrences in this graph. Note that for strings and trees, maximum sets of non-overlapping occurrences can be obtained in linear time using left-to-right and post order, respectively, and assigning occurrences in a greedy way, in much the same method as we do for graphs. For graphs however, we know of no order that always guarantees a maximum set of non-overlapping occurrences using this approximation. Our implementation offers a choice of four different orders, which are detailed in Section 3.3.1.

Updating during Digram Replacement The above method works during initial traversal, however every time an occurrence of a digram is replaced, the occurrence lists of affected digrams need to be updated (line 10 in the algorithm). Let o be an occurrence of d that is being replaced and let F be the set of edges in g that are incident with the attachment nodes of o. Removing o from the graph can only affect the occurrence lists of digrams that have occurrences using edges in F. In particular, for the two edges in o (e_1 and e_2) we need to remove every occurrence that either e_1 or e_2 participate in from the corresponding occurrence list. This can be done efficiently, as we store a list of digrams with every edge for which the edge appears in an occurrence. After the replacement let e' be the new A-labelled nonterminal edge in g. Then for every $e \in F$ the pair $\{e', e\}$ is an occurrence of a digram, and is thus inserted into the appropriate occurrence list (and the frequency counts are updated accordingly). Again, this leads to potential complexity issues. Let k be the sum of degrees of all attachment nodes of o. Then there are $O(k)$ pairs of edges to be considered as occurrences with the new nonterminal edge. This is not a problem in itself, but examine the following situation: let there be an attachment node v with degree k. Further, let every one of the k edges around v be part of a distinct occurrence of the digram d being replaced. As explained, when replacing one of these occurrences, pairs of the other $k-1$ edges and the new nonterminal are possible occurrences of a new digram. Now however, when replacing the next occurrence, the remaining $k-2$ are again possible pairings with the new nonterminal. Thus, during all the replacement steps, we would again need to consider $O(k^2)$ occurrences. To solve this issue, we again first look at the case of a graph without edge labels and directions. Then, if node v has degree k, only one of the edges will make a new occurrence with e', every other one would overlap (at e'). Thus, only one of the $O(k)$ edge pairs in F actually needs to be added as a new occurrence (per label), and this pair needs to be picked in constant time, to guarantee a bound of $O(k)$ for the entire process. Our implementation does this by storing a list of available edges (E_{σ_1,σ_2}) for every pair of edge labels attached to every node of the graph. For every edge label the first edge e in the respective list is selected to create the occurrence $\{e', e\}$. This takes $O(|\Sigma|)$ time, i.e., constant time per label.

3.2.2 Pruning the Grammar

As mentioned in Section 3.1, pruning can reduce the size of a grammar. For string grammars, pruning removes every nonterminal that is referenced only once in the grammar. The presence of parameters for tree grammars, or external nodes for HR grammars, complicates the pruning step. While every nonterminal only referenced once can be removed just like before, it is then possible to still have nonterminals which are referenced more than once but do not contribute to the compression (this is *not* the case for strings). For example, consider the following tree grammar (from [LMM13, Example 8]):

$$\begin{aligned} S &\to f(A(a,a), B(A(a,a))) \\ A(y_1,y_2) &\to f(B(y_1), y_2) \\ B(y_1) &\to f(y_1, a) \end{aligned}$$

In this grammar A and B are each referenced twice, but do not "contribute" to compression. To see this, note that the grammar above has size 12 (recall that the size of a tree is defined as its number of edges) and consider the grammars obtained by removing either nonterminal (left: after removing A, right: after removing B):

$$\begin{aligned} S &\to f(f(B(a),a), B(f(B(a),a))) \qquad & S &\to f(A(a,a), f(A(a,a),a)) \\ B(y_1) &\to f(y_1, a) & A(y_1,y_2) &\to f(f(y_1,a), y_2) \end{aligned}$$

These grammars now have sizes 11 and 12, respectively. Clearly, both rules were therefore not contributing in the full grammar above, as neither grammar is now larger than before. However, removing A first did make the grammar *smaller* than it was before. Further, if we were to remove one more nonterminal, we would in both cases get the original tree

$$f(f(f(a,a),a), f(f(f(a,a),a),a)),$$

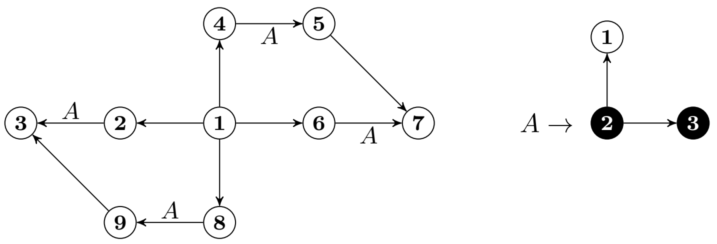


Figure 3.5: Example of an SL-HR grammar to be encoded.

which is still of size 12. This shows that the A rule was still not contributing after removal of the B-rule, but the B-rule turns into a contributing rule if the A-rule is removed first. Thus the order in which the nonterminals are removed may also affect the quality of the pruning. Finding an optimal order is a complex optimization problem as mentioned in [LMM13, Section 3.2].

Let us consider pruning for SL-HR grammars. For a nonterminal A of rank n we define $\mathsf{handle}(A) = (\{v_1, \ldots, v_n\}, \{e\}, \mathsf{lab}(e) = A, \mathsf{att}(e) = v_1 \cdots v_n, \mathsf{ext} = v_1 \cdots v_n)$. The *contribution of* A is defined as

$$\mathsf{con}(A) = |\mathsf{ref}(A)| \cdot (|\mathsf{rhs}(A)| - |\mathsf{handle}(A)|) - |\mathsf{rhs}(A)|,$$

where $\mathsf{ref}(A) = |\{e \in E_S \mid \mathsf{lab}(e) = A\}| + \sum_{B \in N} |\{e \in E_{\mathsf{rhs}(B)} \mid \mathsf{lab}(e) = A\}|$ is the number of edges labelled A in the grammar. The contribution of A counts by how much the size of the grammar changes when every instance of the nonterminal is derived, i.e., it measures how much A contributes towards compression. If $\mathsf{con}(A) > 0$ then we say that A *contributes towards compression*. The grammar in Figure 2.2a represents the graph of Figure 3.4. Here, the A-rule has $\mathsf{con}(A) = 4 \cdot (5 - 3) - 5 = 3$ and thus contributes to the compression. The reader may verify that the sizes of this grammar and the graph differ by exactly three. Note that, as we remove rules, the contribution of other nonterminals might change as edges are added or deleted. Therefore, the effectiveness of pruning depends on the order in which the nonterminals are considered. For TreeRePair, a bottom-up hierarchical order works well in practice. We use a similar approach. First every nonterminal A with $\mathsf{ref}(A) = 1$ is removed, because, by definition, they do not contribute towards compression. The order does not matter for this step. To remove A we apply its rule to each A-edge in the grammar and remove the A-rule. Then we traverse the nonterminals in bottom-up $\leq^g_{\mathsf{NT}}$-order (see Section 2.3), removing each nonterminal with $\mathsf{con}(A) \leq 0$.

3.2.3 Encoding SL-HR Grammars

We encode the start graph and the productions in different ways. As an example, consider the grammar in Figure 3.5. The start graph is encoded using k^2-trees [BLN14], using $k = 2$ as this provides the best compression. This data structure partitions the adjacency matrix into k^2 squares and represents it in a k^2-ary tree. Consider the left adjacency matrix in Figure 3.6. The 9×9-matrix is first expanded with 0-values to the next power of two; i.e., 16×16. If one partition has only 0-entries, a leaf labelled 0 is added to the tree. This happens for the 3rd and

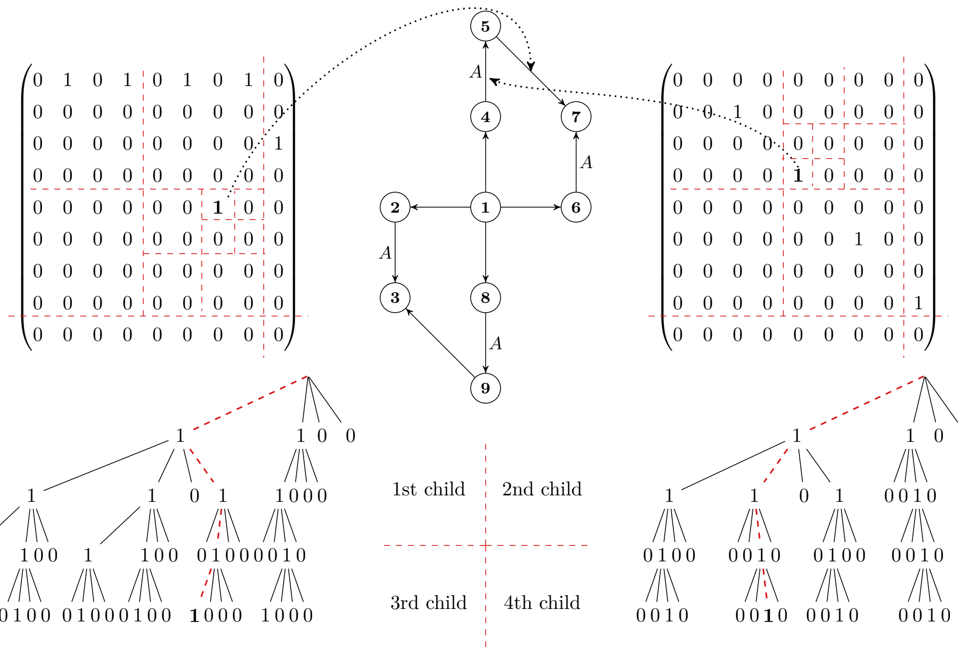


Figure 3.6: Start graph (middle): terminal edges (left) and nonterminal edges (right) and their k^2-tree representations (below) with $k = 2$.

4th partition in this case (the partitions are numbered left to right, top to bottom as indicated in the bottom centre of the figure). Thus the 3rd and 4th child of the root are 0-leaves. The other two have at least one 1-entry, therefore inner nodes labelled 1 are added and the square is again partitioned into k^2 squares. This is continued at most until every square covers exactly one value. At this point the values are added to the tree as leaves. As we need to consider edges with different labels, we use a method similar to the representation of RDF graphs proposed in [ÁBF+15] and mentioned as a possible representation of hypergraphs in Section 2.6. Let $E_\sigma \subseteq E$ be the set of all edges labelled σ. For every label σ appearing in S we encode the subgraph (V, E_σ). If $\mathsf{rank}(\sigma) = 2$, then this is encoded as an adjacency matrix. Otherwise we use an incidence matrix, i.e., a matrix that has one row for every edge and a column for every node. Thus, a 1 in row i, column j of the incidence matrix means, that edge i is attached to node j. All of these matrices are encoded as k^2-trees. Figure 3.6 is an example with two edge labels. Note that this example only uses edges of rank 2. For a hyperedge e, the incidence matrix only provides information on the set of nodes attached to e, but not the specific order of $\mathsf{att}(e)$. For this reason we also store a permutation for every edge to recover $\mathsf{att}(e)$. We count the number of distinct such permutations appearing in the grammar and assign a number to each. Then we store the list encoded in a $\lceil \log n \rceil$-fixed length encoding, where n is the number of distinct permutations.

For the rules we use a different format, as we expect the right-hand-sides to be very small graphs (due to pruning, they may be larger than just a digram). We store an edge list for every production, encoding the nodes using a variable-length δ-code [Eli75]. One more bit per node is used to mark external nodes. As the order of the external nodes is also important, we make sure that the order induced by the IDs of the external nodes is the same as the order of the external nodes (this is automatically the case for normalized graphs). Every production begins with the edge count (again, using δ-codes). For every edge we first use one bit to mark terminal/nonterminal edges, then store the number of attached nodes, followed by the δ-codes of the list of IDs. Finally, we also use a δ-code for the edge label. The edges are stored using the order given in Section 2.3. This way, nonterminals appear in the same order as in the sibling-tuple. For the rule in Figure 3.5 this leads to the following encoding:

$\delta(2)$	two edges
$0\delta(2)$	edge is terminal (0), has two nodes
$1\delta(2)0\delta(1)\delta(1)$	nodes 1 (external) and 2 (internal), label 1
$0\delta(2)$	next edge is terminal, has two nodes
$1\delta(2)1\delta(3)\delta(1)$	nodes 2 (external) and 3 (external), label 1

This is a bit sequence of length 33.

3.3 Important Parameters

Two user-definable parameters have potentially big impact on the compression performance. The node order has an influence on the size of the sets of non-overlapping occurrences found in step 3 of Algorithm 1, as already mentioned in Section 3.2.1. The second parameter is the maximal rank a digram may have during the compression. In the next two sections we discuss the specific node orders available in the gRePair implementation, and we show that there cannot be an optimal choice for the maximal rank of a digram that always produces the best result.

3.3.1 Node Order

As discussed before the node order heavily influences the digram counting, which in turn influences the compression behaviour. A node order $<_V$ is given by a bijective function $f : V \to [|V|]$ such that $u <_V v$ if $f(u) < f(v)$. Some of the orders below are modelled by functions $h : V \to [k]$ for some $k < |V|$, i.e., there exist nodes u, v such that $h(u) = h(v)$, but a strict order is needed to run the algorithm. For this reason, we consider sets of functions $F = \{f : V \to [|V|] \mid \forall u, v \in V : f(u) < f(v) \Rightarrow h(u) \leq h(v)\}$ to be an admissible set of orders. This means, that whenever we state that a node order was used, we actually use an arbitrary order out of the set of admissible orders. We evaluate these orders:

1. *natural order (nat)* uses the node numbers as given in the source graph,
2. *BFS* order follows a breadth-first traversal,
3. *FP* computes a fixpoint on the node neighbourhoods starting from the degrees, and
4. *FP0*, which is a degree order.

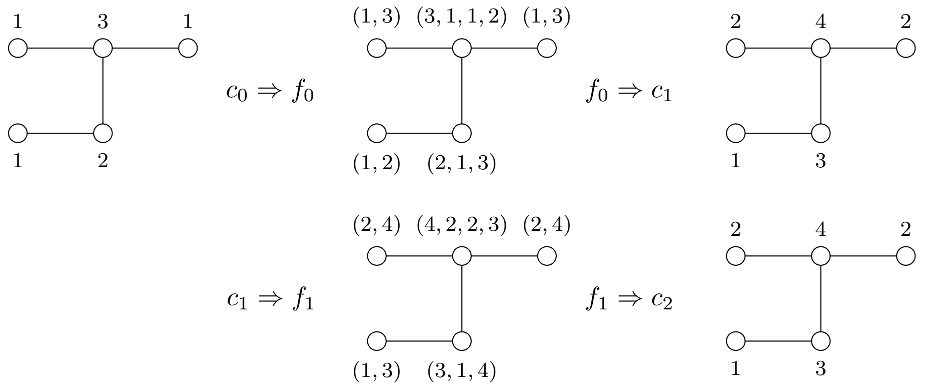


Figure 3.7: Computing the FP-order of a small graph.

Formally, the natural order is just the identity. In the case of BFS there exists an additional element of nondeterminism. We choose as the first node v (i.e., the one with $f(v) = 1$) any node of lowest degree. For any other node u the function $h(u)$ evaluates to the length of a shortest path from v to u. This forms the basis for an admissible set of orders. Note that, for graphs with more than one connected component, one node v has to be picked for every connected component. All of these initial nodes evaluate to 1 by h. We now define the FP and FP0 orders. For a graph g let $c_i : V_g \to \mathbb{N}$ be a family of functions that colour every node with an integer. We first define $c_0(v) = d(v)$, where $d(v)$ is the degree of v. This is the order FP0. Now we map every node v to the tuple $f_0(v) = (c_0(v), c_0(v_1), \ldots, c_0(v_n))$, where $v_1, \ldots, v_n$ are the neighbours of v ordered by their values in c_0. We sort these tuples lexicographically and let $c_1(v)$ be the position of $f_0(v)$ in this lexicographical order. This process is iterated until $c_{i+1} = c_i$. The process terminates due to the lexicographical ordering of f_i: it ensures that the relative position of a node v within c_i and c_{i+1} remains unchanged, i.e., for every node u with $c_i(u) < c_i(v)$ we also have $c_{i+1}(u) < c_{i+1}(v)$ and vice-versa. Thus the order c_{i+1} is always finer than the order c_i and a fixpoint is guaranteed. Now c_i can be used as a basis for an admissible set of orders. This computation of the order works for undirected, unlabelled graphs, but can be straightforwardly extended to directed labelled graphs. We call this order FP.

Example 3.3. Figure 3.7 shows an example of the FP-order. The graph on the left is annotated by c_0, the graph in the middle shows f_0, which is then ordered lexicographically to get c_1 on the right. This is the fixpoint for this graph, as $c_2 = c_1$.

Note that FP is not a strict order and thus also implies an equivalence relation on the nodes ($v \cong_{\text{FP}} u$ if and only if $c_i(v) = c_i(u)$). The number of equivalence classes of $\cong_{\text{FP}}$ has an interesting correlation with the compression ratio of gRePair, as discussed in Section 3.6.2.

3.3.2 Maximal Nonterminal Rank

The maximal rank is a user defined parameter of gRePair that specifies the maximal rank of a digram (and thus the maximal rank of a nonterminal edge) that the compressor considers. Digrams with a higher rank are ignored and not counted. It was shown already for TreeRePair [LMM13, Theorems 9 and 10] that choosing this parameter too high or too small can strongly degrade the compression ratio. The two families of trees given there, can be converted into families of graphs showing the same relation to the maximal rank for gRePair. We briefly recap the arguments, but as the proof details are almost identical to the case for TreeRePair, we do not repeat them here.

High Rank Let $g_{n,k}$ be the graph $\mathsf{t\text{-}graph}(s_{n,k})$, where $s_{n,k}$ is a tree consisting of a right comb of 2^n nodes labelled f_k, a symbol of rank k. The first $k-1$ children of these nodes are leaves labelled a, the last child is the next f_k-labelled node (except for the last of these, which only has a-children). This is the same tree as given in [LMM13, Theorem 9]. The size of $g_{n,k}$ depends on k: for $k = 1$ the f-edges are of rank 2 (i.e., size 1) and thus we get $|g_{n,1}| = 2^n + 2^n + 1$. For $k > 1$ the size of each f-edge is $k + 1$ and thus $|g_{n,k}| = 2^n \cdot (k + 1) + 2^n + 1$.

Lemma 3.4. *Given $g_{n,k}$ gRePair produces a grammar of size $O(k^2 + n)$ if the maximal rank allowed is at least k, and does not compress at all if the maximal rank is less than k.*

It is easy to see that there can be no compression with a maximal rank less than k: $g_{n,k}$ does not contain occurrences of any digram with a rank smaller than k. Once the allowed rank is at least k, gRePair reduces the width of the tree by 1 with every iteration by combining one of the a-leaves with its f_k-/nonterminal parent. Finally a line of 2^n nonterminal edges remains, which all have the same label and can be compressed exponentially. This argument is identical to the one in the proof of [LMM13, Theorem 9], only the notation changes to the one used for graphs.

Small Rank We give an example of graphs which gRePair compresses best, if the maximum rank is limited to 2. Our example is similar to the one for TreeRePair [LMM13, Theorem 10]. The main difference is that we use a graph with 2^n edges labelled f, instead of nodes. Let T_n be a graph over $\Sigma = \{f, a_0, a_1, a_2, a_3, a_4\}$ with $2^{n+1} + 2$ nodes $\{0, \dots, 2^{n+1} + 1\}$ and 2^{n+1} edges $\{e_1, \dots, e_{2^{n+1}}\}$. The attachment relation is defined as $\mathsf{att}(e_i) = i - 1 \cdot i$ for $i \in \{1, \dots, 2^n\}$, and $\mathsf{att}(e_i) = i - 2^n \cdot i$ for $i \in \{2^n + 1, \dots, 2^{n+1} + 1\}$. They are labelled by $\mathsf{lab}(e_i) = f$ for $i \in \{1, \dots, 2^n\}$ and

$$\mathsf{lab}(e_i) = \begin{cases} a_0 & \text{if } i \equiv 0 \pmod 5 \\ a_1 & \text{if } i \equiv 1 \pmod 5 \\ a_2 & \text{if } i \equiv 2 \pmod 5 \\ a_3 & \text{if } i \equiv 3 \pmod 5 \\ a_4 & \text{if } i \equiv 4 \pmod 5 \end{cases},$$

for $i \in \{2^n + 1, \dots, 2^{n+1}\}$. This graph is a tree consisting of a path of 2^n edges labelled f, and every node attached to one of these f-edges, is also attached to an edge labelled a_0, a_1, a_2, a_3, or a_4. These edges appear in this order, i.e., if a node is attached to an edge labelled a_i, then the previous node (one f-edge up in the tree) is labelled a_{i-1} (mod 5), and the next node (one f-edge down in the tree) is labelled a_{i+1} (mod 5). The size of T_n is $2^{n+2} + 2$.

Lemma 3.5. *Given T_n, gRePair produces a grammar of size $O(n)$ if the maximal rank is restricted to 2 and compresses at best to 75% of the original size if the maximal rank is unbounded.*

The argument for this is mostly identical to [LMM13, Theorem 10]. With a maximal rank of 2 it compresses well, because gRePair then only has limited choice in digrams. It will first replace the pairs of f- and a_l-labelled ($0 \leq l \leq 4$) edges, because every other possible digram has a rank greater than 2. After this is done, a string-graph remains that can now be compressed exponentially. If the maximal rank is unrestricted, the digrams of higher rank (pairs of 2 f-edges in the first iteration) occur more frequently and are thus replaced by gRePair. This will replace all the f-edges, but all of the nodes remain in the start graph. Furthermore, in the next step, pairs of nonterminal edges will be replaced as the most frequent digram, again without sharing any nodes. This continues until, after the last iteration, the start graph still has all $2^{n+1} + 2$ nodes, and all the edges labelled a_l for $0 \leq l \leq 4$. Furthermore, it has 2 nonterminal edges of rank 2^{n-1}. Thus,

$$|S| = 2^{n+1} + 2^n + 2 \cdot 2^{n-1} + 4 = 2^{n+2} + 4.$$

Regardless of the rules and the pruning step, $|S|$ is already larger than $|T_n|$. In particular note that, not counting the nonterminal edges, the size of S is still $x = 2^{n+1} + 2^n + 4$. Therefore

$$\frac{x}{|T_n|} = \frac{2^{n+1} + 2^n + 4}{2^{n+1} + 2^{n+1} + 2} \overset{n\to\infty}{\longrightarrow} \frac{3}{4}.$$

The pruning step reduces the grammar's size, but the nodes and terminal edges in the start graph remain. Thus, the compression ratio cannot be better than 3/4. Note that for TreeRePair the compression with unbounded rank is at best 50% instead of 75%. The reason for this is, that the size definition for trees only counts edges, whereas we consider nodes and edges in graphs.

3.4 GraphRePair in Relation to Variants for Strings and Trees

In the previous section we already drew some parallels between RePair for Trees and gRePair. Before going on, we look in more detail on the relation of gRePair with the previous variants for strings [LM00] and trees [LMM13]. Let w be a string, and $d_1, \ldots, d_n$ be the digrams that string-RePair would replace during its run on w. If we run gRePair on $\mathsf{s\text{-}graph}(w)$, and use a left-to-right node order, then the digrams $d'_1, \ldots, d'_n$ that gRePair replaces are exactly $\mathsf{s\text{-}graph}(d_1)$ to $\mathsf{s\text{-}graph}(d_n)$. Therefore, we obtain a graph-encoding of the string-grammar that the original RePair generates. The comparison with TreeRePair is less clear. TreeRePair compresses ordered trees where the nodes, not the edges, are labelled from a finite alphabet. For gRePair to compress trees in the same way, we need to represent the tree as a tree-graph as given in Section 2.4 and use a similar postOrder-traversal as TreeRePair does. We can experimentally confirm that gRePair achieves comparable (within the same order of magnitude) compression ratios as TreeRePair. Therefore, generally, gRePair is a generalisation of these previous variants.

An SL tree grammar achieves compression by representing only once repeating *connected subgraphs* of the given tree. An SL-HR grammar is more general, because it can share repeating *disconnected* subgraphs. Does this allow, for an SL-HR grammar, to compress a tree more effectively than any SL tree grammar? We show that this is *not* the case by proving that every SL-HR grammar G that represents a tree (string), can be converted to an SL-HR grammar, that

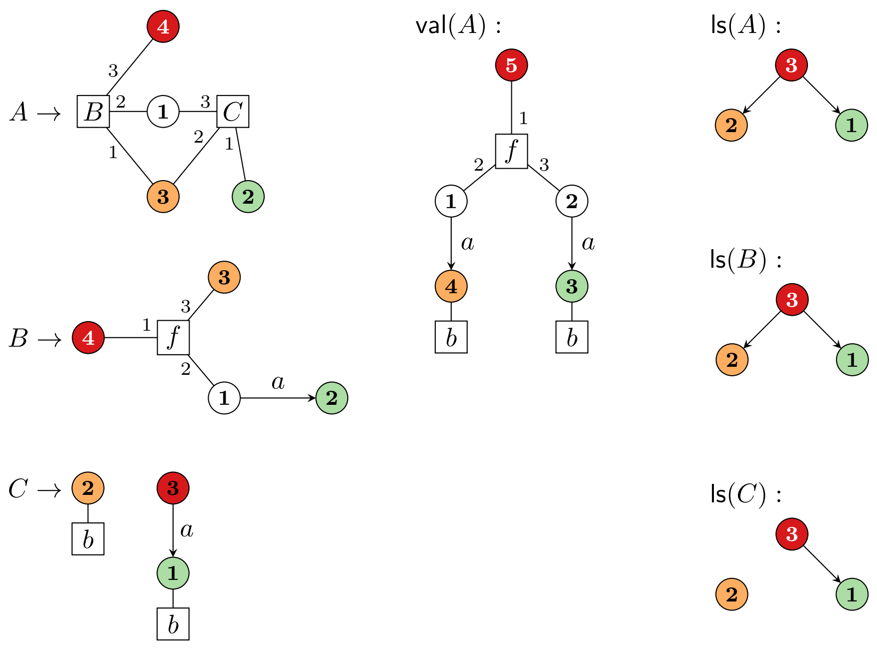


Figure 3.8: A set of tree-generating rules, the tree they represent, and their line-structures.

is only slightly larger than G and for which every right-hand side is a tree (string). This is not trivial, as the expressive power of graph grammars is higher than the one of context-free string grammars. For example, it is possible to generate the language $\{\mathsf{s\text{-}graph}(w) \mid w = a^n b^n c^n, n \in \mathbb{N}\}$ using HR grammars, but the string-language represented here is not context-free. We call an SL-HR grammar G *tree generating*, if $\mathsf{val}(G)$ is a tree-graph as defined in Section 2.4. Similarly, G is called *string-generating*, if $\mathsf{val}(G)$ is a string-graph. Recall that a path from a node u to a node v in a hypergraph is a tuple $(e_1, \dots, e_n)$ of edges, which connect u to v, such that an edge of rank k is considered to have one source (the first attached node) and $k - 1$ target nodes, and that we call a path internal, if it uses only internal nodes (with the possible exception of u and v). For example, in Figure 3.8 the right-hand side of A has a path (C, B) from the green to the red node, but it is *not* an internal path, because it crosses the orange node.

Definition 3.6. Let g be a graph with the set X of external nodes. Then the *line-structure* $\mathsf{ls}(g) = (X, E)$ is the directed, unlabelled graph such that for all $u, v \in X, (u, v) \in E$ if and only if there is an internal path from u to v in $\mathsf{val}(g)$.

For a nonterminal A of an SL-HR grammar we denote $\mathsf{ls}(\mathsf{rhs}(A))$ by $\mathsf{ls}(A)$. For an example of a line-structure, see Figure 3.8. Note that if a grammar G is tree-generating, $\mathsf{ls}(A)$ is always a rooted forest for every nonterminal A in G (otherwise there would be a cycle in $\mathsf{val}(A)$, because G is tree-generating and A is useful). We refer to this property by (LG). However, the right-hand side of A may not be a tree-graph itself, as in the A-rule in Figure 3.8. There are two ways in which $\mathsf{rhs}(A)$ for a rule in a tree-generating grammar can not be a tree-graph: it can contain cycles, or nodes with multiple parents. Both of these cases can only happen with nonterminal edges, however. The line-structure is a tool to split these nonterminal edges, such that the resulting graphs are tree-graphs. The following Theorem is a variant of [Dre99, Theorem 7.4] assuming straight-line grammars, and considering how the transformation affects the size.

Theorem 3.7. *Let $G = (N, P, S)$ be a tree-generating SL-HR grammar. We can construct an SL-HR grammar $G' = (N', P', S')$ such that*

1. $\mathsf{val}(G') = \mathsf{val}(G)$,
2. $|G'| \leq 2|G|$, *and*
3. $\mathsf{rhs}_{G'}(A)$ *is a tree-graph for every* $A \in N' \cup \{S'\}$.

Proof. We describe three transformations on G, which, applied in order, yield the desired grammar G'. In the first transformation, the set of nonterminals remains unchanged. We use the line-structure to order the external nodes in a top-down way, i.e., if u is an ancestor of v within the line-structure, then u should be ordered before v. Let $\mathsf{ls}(A) = (V, E)$ for some nonterminal A, and let $u, v \in V$. We define the partial order $\prec_A$ such that $u \prec_A v$ if and only if there is a directed path from u to v in $\mathsf{ls}(A)$. Due to the property (LG) mentioned above, $\prec_A$ is well-defined. Let (A, g) be a rule, and u, v two of its external nodes, such that $u \prec_A v$, but v appears before u in ext_g. We reorder ext_g (and by extension, the attachment relation of every nonterminal edge labelled A in the entire grammar) such that they are sorted according to some (arbitrary, but fixed) strict order consistent with $\prec_A$. This step does not change the structure of the grammar or affect its size. As an example, consider $\mathsf{ls}(X)$ for $X \in \{A, B, C\}$ in Figure 3.8. In all three cases, the red external node, needs to come first to be consistent with $\prec_X$, but it is currently third. The orange and green nodes can be ordered arbitrarily after that. Permissible orders for ext thus are red < orange < green or red < green < orange (see Figure 3.9 for the final grammar).

In the next two steps, additional nonterminals are (in general) added to the set of nonterminals. We iterate over N in a bottom-up fashion, i.e., in reverse $\leq_{\mathsf{NT}}$-order. If we encounter an $A \in N$ such that $\mathsf{ls}(A)$ has at least two connected components $g_1, \ldots, g_k$ then we add rules (A_i, h_i) for $i \in [k]$, where A_i is a new nonterminal of rank $|V_{g_i}|$, and h_i is a subgraph of $\mathsf{rhs}(A)$. Note that $\sum_{i\in[k]} A_i = \mathsf{rank}(A)$. The set of nodes of h_i is defined as

$$U_i = \{u \in \mathsf{rhs}(A) \mid \exists v \in V_{g_i} : \text{ there is a path from } v \text{ to } u \text{ in } \mathsf{rhs}(A)\},$$

i.e., U_i contains the nodes of $\mathsf{rhs}(A)$ that can be reached from the external nodes in g_i. The set of edges of h_i contains every edge of $\mathsf{rhs}(A)$, which is attached only to nodes in U_i. Let $w = a_1 \cdots a_n$ be a string and $X = \{x_1, \ldots, x_m\} \subseteq [n]$ a set of indices. By $w|X$ we denote the string $a_{x_1} \cdots a_{x_m}$ such that $x_1 < x_2 < \cdots < x_m$. We define the set of i-indices as

$$\mathsf{idx}[i] = \{j \mid \exists v \in V_{g_i} \text{ such that } v \text{ is at position } j \text{ within } \mathsf{ext}_{\mathsf{rhs}(A)}\}.$$

Using this, we define $\mathsf{ext}[i] = \mathsf{ext}_{\mathsf{rhs}(A)}|\mathsf{idx}[i]$, and for some edge e labelled A we define $\mathsf{att}[i](e) = \mathsf{att}(e)|\mathsf{idx}[i]$. Now, we replace every occurrence of an edge e labelled A within the grammar by k edges, such that for $i \in [k]$ the edge e_i is labelled A_i and attached to the nodes $\mathsf{att}[i](e)$. We remove the nonterminal A and its rule from the grammar. Note that now all right-hand sides are tree graphs. Note further that $\sum_{i\in[k]} \mathsf{rank}(e_i) = \mathsf{rank}(e)$, and $\sum_{i\in[k]} |h_i| \leq |\mathsf{rhs}(A)|$ and therefore the size of the new grammar is at most $|G|$.

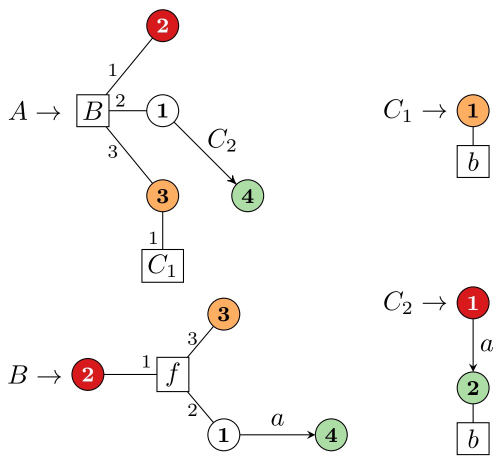


Figure 3.9: The rules of Figure 3.8 after converting them to tree-graphs.

In the third step we eliminate rules that have external nodes, which are neither root nor leaves. To do so, we again split the rules up. Let (A, g) be a rule which has at least three external nodes u, v, w such that there is a path from u to v and from v to w, i.e., v is an inner node and not root or leaf. Let $g_1, \ldots, g_k$ be the maximally sized subgraphs of g, such that their union is g, and for every $i \in [k]$, g_i is a tree-graph which *does not* have a triple of external nodes u, v, w such that paths exist from u to v and v to w, i.e., every external node of g_i is either a root or a leaf. Let e be an A-labelled edge of the grammar and let $\mathsf{ext}[i]$ and $\mathsf{att}[i](e)$ be defined as above. Now we again replace (A, g) by the rules $(A_1, g_1), \ldots, (A_k, g_k)$ and every nonterminal edge e with $\mathsf{lab}(e) = A$ by the edges $e_1, \ldots, e_k$, where $\mathsf{lab}(e_i) = A_i$ and $\mathsf{att}(e_i) = \mathsf{att}[i](e)$. In this case, an external node that is neither root nor leaf will appear in more than one of the graphs $g_1, \ldots, g_k$, thus this step increases the size of the grammar. This can be bounded by

$$\sum_{i\in[k]} \mathsf{rank}(A_i) \leq 2\mathsf{rank}(A) \text{ and } \sum_{i\in[k]} |g_i|_V \leq 2|g|_V - 1.$$

The factor 2 comes from the fact that, in the worst case, we may split at every external node. Thus the node- and edge-sizes are at most doubled. Therefore, for the grammar G' we have $|G'| \leq 2|G|$. □

As an example, we provide in Figure 3.9 the conversion of the rules presented in Figure 3.8. We first adjust the order of the external nodes to one consistent with $\prec_X$ for $X \in \{A, B, C\}$ by using red < orange < green as the order of the external nodes in all three cases. Thus, the first external node becomes the third one, and the third external node becomes the first one. In the same way, the appropriate nonterminal edges (B and C in $\mathsf{rhs}(A)$) have their attachment relation reordered. We then apply the second step, splitting the C-rule into two rules C_1 and C_2. In this case the conversion was already finished after the second step, thus the new rules are not larger than the original ones (in fact, they are smaller, because the C_2-edge now is of rank 2, which counts as size 1).

Consider again Theorem 3.7. It clearly follows from the construction of G' that $V_{S'} = V_S$. This allows us to easily transfer the result also to string-graphs. Let G be a string-generating

SL-HR grammar. Recall that G's start graph S has two external nodes (because it generates a string-graph), and note that every nonterminal of G has rank ≥ 2. We convert G into a tree-generating SL-HR grammar by making the second external node v in S internal. The resulting grammar generates a monadic tree, and we can use the result of Theorem 3.7 to convert it into a grammar, where every right-hand side is a monadic tree-graph (of rank 2). Finally, this grammar can be converted back into a string-generating grammar, by again making v external.

Corollary 3.8. *Let $G = (N, P, S)$ be a string-generating SL-HR grammar. We can construct an SL-HR grammar $G' = (N', P', S')$ such that*

1. $\mathsf{val}(G') = \mathsf{val}(G)$,
2. $|G'| \leq 2|G|$, *and*
3. $\mathsf{rhs}_{G'}(A)$ *is a string-graph for every* $A \in N' \cup \{S'\}$.

3.5 Choice of Formalism

We discuss briefly the reasoning for some of our choices regarding the grammar formalism used. This includes the choice of hyperedge replacement over a different replacement method, the restrictions we further enforce on hypergraphs, and how the size of a hypergraph is defined.

3.5.1 Hyperedge vs. Node Replacement

There are two well-known types of context-free graph grammars (see, e.g. [Eng97]):

- context-free hyperedge replacement grammars (HR grammars), and
- context-free node replacement grammars (NR grammars).

In terms of graph language generating power, NR grammars are strictly more expressive than HR grammars. For instance, NR grammars can describe the set of all complete graphs, whereas this is not possible with HR grammars. If we use NR grammars to produce bipartite graphs that encode hypergraphs, then the resulting hypergraph language generating power is exactly the same as that of HR grammars (see [Eng97, Theorem 4.28]). Note that for any given SL-HR grammar one can construct an equivalent (with respect to a bipartite encoding of hyperedges) SL-NR grammar of similar size.

The difference in expressive power comes due to node replacement grammars using a different formalism for derivation. Instead of merging external nodes with the nodes in the nonterminals neighbourhood, every rule in an NR-grammar includes a *connection relation*, which specifies, which nodes in the rule are to be connected to which nodes in the nonterminal's neighbourhood. Consider as an example the grammar given in Figure 3.10: the initial graph S has two nonterminal nodes, both labelled A. The rule for A has two nonterminal nodes labelled B, and the connection relation $\{(A, B), (C, B), (D, B), (a, B)\}$. When deriving one of the A-nodes in S we note that the neighbourhood of this node consists of only one node with label A. The tuple (A, B) in the connection relation means that every A-node from the nonterminals neighbourhood is connected

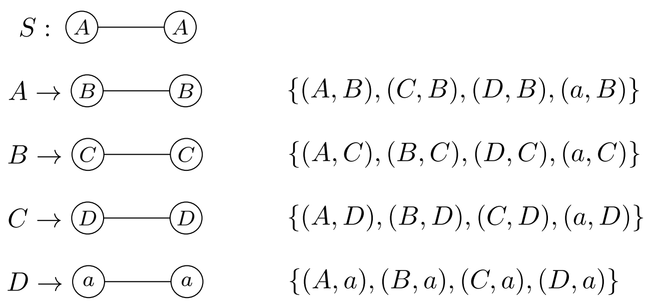

S :
A →
B →
C →
D →
{(A, B), (C, B), (D, B), (a, B)}
{(A, C), (B, C), (D, C), (a, C)}
{(A, D), (B, D), (C, D), (a, D)}
{(A, a), (B, a), (C, a), (D, a)}

Figure 3.10: NR grammar that generates a complete graph with 32 nodes.

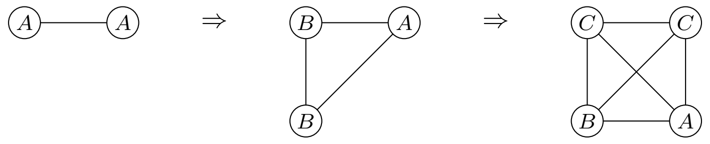


Figure 3.11: Two derivation steps of the grammar from Figure 3.10.

with every B-node in the rule. Figure 3.11 shows this, and one more derivation step as an example.

The natural choice to define a digram for use of RePair with NR-grammars would be two neighbouring nodes and the edge between them. However, due to the connection relation, we would also need to include the neighbourhood in this definition, as different occurrences of the same two nodes may need different connection relations. These may be incompatible with each other in some cases, or could be merged into one rule in others. This could make digrams too specific, and occurrences somewhat rare. It is therefore unclear, whether the use of node replacement yields a successful RePair variant, but we note that it is interesting future work, particularly as we believe, that complete graphs can be compressed much better using NR grammars.

Conjecture 3.9. *Let C_n be a complete graph with 2^n nodes. Then there exists an SL-NR grammar G_n such that $\mathsf{val}(G_n) = C_n$, and $|G_n| \in O(n)$, but there is no SL-HR grammar G'_n such that $\mathsf{val}(G'_n) = C_n$ and $|G_n| \in O(n)$.*

The previously mentioned grammar in Figure 3.10 is an example for an SL-NR grammar that compresses a complete graph as in the conjecture and can be extended to generate C_n for any n. The connection relation for this grammar could also use a shorthand (i.e., $\{(*, a)\}$ could be used for the connection relation of the D-rule instead of $\{(A, a), (B, a), (C, a), (D, a)\}$). Doing so yields a grammar with n rules, each having two nodes, one edge, and one tuple in the connection relation. We currently have no proof for the second part of the conjecture.

3.5.2 On the Additional Conditions for Hypergraphs

Recall, we also enforce two conditions on hypergraphs, which are not always found in the literature:

(C1) for every edge e: $\mathsf{att}(e)$ contains no node twice, and

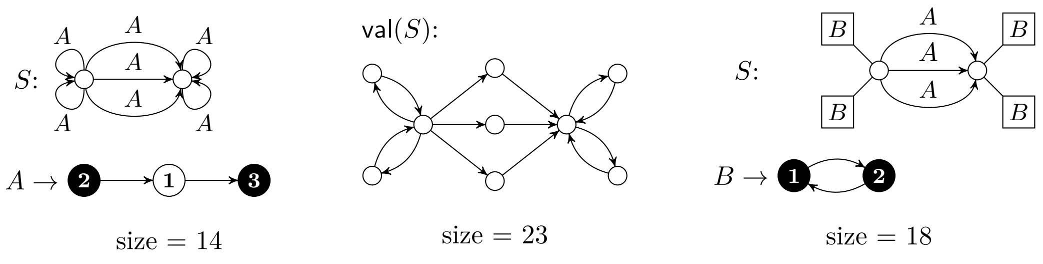


Figure 3.12: Example for a grammar that is not att-distinct (left), but smaller than the att-distinct grammar produced by gRePair (right) representing the same graph (middle).

(C2) the string ext of external nodes contains no node twice.

We refer to these as att-, and ext-distinctness, respectively. We also call a rule att-distinct (ext-distinct), if its right-hand side is att-distinct (ext-distinct). A grammar is att-distinct (ext-distinct), if every rule and the start graph are att-distinct (ext-distinct). These conditions have no effect on the graph language generating power of HR grammars (provided the graphs do not require edges/external nodes that are not att/ext-distinct) as shown in [Hab92, Chapter 1, Theorem 4.6]. With respect to compression however, this is not the case in both cases: For efficiency reasons gRePair does not consider or produce edges that are not att-distinct. This potentially weakens the compression, as nonterminal edges that are not att-distinct could be used to represent occurrences of different digrams using only one rule. For example, consider the digrams g) in Figure 3.2. Occurrences of them could also be replaced by a rule using the digrams a) from that same figure, by merging the centre and right-most external nodes. This is not necessarily always an improvement: in this case we are using a rank-3 nonterminal edge, where a rank-2 edge would have sufficed, thus storing larger nonterminals. It can be shown however, that there are grammars, which are not att-distinct, but smaller than the smallest att-distinct grammar for the same graph. Figure 3.12 is an example for such a graph. For ext-distinctness, on the other hand, we can show that the condition not only has no effect on compression, but enforcing it makes the grammar *smaller.*

Lemma 3.10. *Given a non-ext-distinct SL-HR grammar $G = (N, P, S)$, an ext-distinct SL-HR grammar G' can be constructed (in linear time), such that $L(G') = L(G)$ and $|G'| < |G|$.*

Proof. For a string w let $\mathsf{symb}(w)$ be the set of symbols appearing in w, and $\mathsf{dist}(w)$ the string v derived from w by only keeping the first occurrence of every distinct symbol in w. For example, for $w = acbac$, $\mathsf{symb}(w) = \{a, b, c\}$, and $\mathsf{dist}(w) = acb$. Let (A, g) be a rule that is not ext-distinct. We first add a new rule (A', g') with $\mathsf{rank}(A') = |\mathsf{dist}(\mathsf{ext}_g)|$, and g' is the same graph as g except with $\mathsf{ext}_{g'} = \mathsf{dist}(\mathsf{ext}_g)$. Then we replace the A-rule by (A, h) where $h = (\mathsf{symb}(\mathsf{ext}_g), \{e\}, \mathsf{lab}(e) = A', \mathsf{att}(e) = \mathsf{dist}(\mathsf{ext}_g), \mathsf{ext} = \mathsf{ext}_g)$. We can now derive every occurrence of A in the grammar and remove the A-rule altogether. Doing this merges some of the nodes in the grammar, as they were referenced by the same external node. Note that this process reduces the size of the grammar. Let u be a node that occurs $k \geq 2$ times in ext_g. Then, applying the A-rules decreases the node size by $k - 1$. The edge-size also decreases by $k - 1$, as A' has a smaller rank. ☐

3.5.3 Size of a Hypergraph

Of particular importance for compression is how the size of the object is defined. In our case it directly influences whether a nonterminal is considered contributing or not, and thus which rules are kept in the pruning stage. The size of simple graphs is usually counted as $|V| + |E|$. For hypergraphs as used here we know of no such convention. Using $|V| + |E|$ seems too reduced: clearly a hyperedge of rank $n >> 2$ should have a higher size than a simple edge, simply because much more information about attached nodes needs to be stored. Initially the size of an edge e was therefore defined as $\mathsf{rank}(e)$. This has the effect that a simple graph has a size of $|V| + 2|E|$ (because all the edges have rank 2). Early experiments showed that, under this size definition, complete graphs could be compressed to about 60%, however the output files were not much smaller than the input (if at all). The size definition presented here (size of an edge e is 1 if $\mathsf{rank}(e) \leq 2$ and $\mathsf{rank}(e)$ otherwise) is motivated from the way the graphs are encoded. Simple edges are encoded using adjacency matrices where only one "1"-entry exists per edge. Hyperedges of rank 3 and higher on the other hand use incidence matrices where $\mathsf{rank}(e)$ "1"-entries exist per edge. It much closer resembles the output sizes, however the amount of different labels (i.e., $|\Sigma| + |N|$) and the possible overhead generated by labels that only occur very few times is still disregarded. This is difficult to include in a size definition, as it depends on other edges and therefore cannot be used in the definition of the size for a single edge.

It is possible to encode a hypergraph into a bipartite simple graph. A bipartite graph is a simple graph g whose set of nodes V can be denoted as the union of two disjoint sets S and T such that $N(x) \subseteq T$ for any $x \in S$ and $N(y) \subseteq S$ for any $y \in T$. A hypergraph g can be encoded as such a bipartite graph, by using $S = V$ and $T = E$ and connecting every node $e \in T$ with $\mathsf{rank}(e)$ simple edges to the nodes $\mathsf{att}_g(e)$. Then the size of a hypergraph can be defined by the sum of nodes and edges of its bipartite encoding. This has similar effects to using the rank of an edge and thus does not correlate well with the file size of our encoding, but might be a good idea if such a bipartite encoding is used to encode SL-HR grammars.

3.6 Experimental Evaluation

We implemented a prototype in Scala (version 2.11.7) using the Graph for Scala library[1] (version 1.9.4). The experiments are conducted on a machine running Scientific Linux 6.6 (kernel version 2.6.32), with 2 Intel Xeon E5-2690 v2 processors at 3.00 GHz and 378 GB memory. As we are only evaluating a prototype, we do not mention runtime or peak memory performance, as these can be considerably improved by a more careful implementation. We compare to the following compressors:

- k^2-tree, where the first 5 levels of the tree use $k = 4$, and beyond that $k = 2$.
- The list merge (LM) algorithm by Grabowski and Bieniecki [GB14]. We use 64 for their chunk size parameter, as in their paper.

[1]http://www.scala-graph.org/

- The combination of dense substructure removal [BC08] and k^2-tree by Hernández and Navarro [HN14] (HN). For the parameters to the algorithm we use $T = 10$, $P = 2$, and $ES = 10$, which are the parameters their experiments show to provide the best compression. Note that this compressor uses a k^2-tree implementation with all the optimizations.

For all three of these, we use implementations provided by the authors. In the case of the k^2-tree-method we use their regular implementation for network and web graphs, but the interleaved method based on [ÁdBBN17] for RDF and version graphs with edge labels, as that yields better results in all but one case. We also experimented with RePair on adjacency lists by Claude and Navarro [CN10], but omit the results here, because on all graphs we tested, stronger compression was achieved by another compared compressor. For the results on network graphs (in Section 3.6.3 below) we also mention some results achieved with MP_k linearizations in [Mas12], a compressor designed to be used with social networks. As we do not have their implementation, we do not include a full comparison.

As common in graph compression, we present the compression ratios in *bpe* (bits per edge). Note that our compressor reorders the nodes. We omit the space required to retain the original node IDs, because we assume that they represent arbitrary data values and it is possible to update this mapping. This is particularly true for RDF graphs, as explained in Section 3.6.3.

Note that it is known that using different node orders will change the results for all of these compressors, including gRePair. We discussed the importance of the traversal order in Section 3.2.1, but another area where the node order will have an effect is the final encoding of the start graph using the k^2-tree-structure. To get comparable results, we use the natural order in all cases, when it comes to the encoding. We ran experiments, where the start graph of the grammar produced by gRePair was reordered according to the initially computed FP-order. This improves some of the results, but we consider it future work to test the impact of different orders on the encoding of the start graph (cf Section 3.8). Furthermore, we use our own implementation of the k^2-tree-method for the encoding of the start graph. It follows the description in [BLN14], using the same binary format, however it does not include the further optimization on compressing the leaf-level described there. The output files of gRePair do not seem to work very well with this optimization, as in most cases the results got slightly worse when using the k^2-tree implementation provided by its authors.

3.6.1 Datasets

We use three different types of graphs: network graphs, which are further split into web graphs (Table 3.1) and social networks (Table 3.2), RDF graphs (Table 3.3), and version graphs (Table 3.4). Each table lists the numbers of nodes and edges and the number of equivalence classes of $\cong_{\mathrm{FP}}$ (see Section 3.3.1) of each graph. For RDF graphs we also list the number of edge labels (i.e., predicates) of each graph. Two of the version graphs also have labelled edges.

We give a short description of each graph: the network graphs are from the Stanford Large Network Dataset Collection[2] and are unlabelled. They are communication networks (Email-

[2]http://snap.stanford.edu/data/index.html

Graph	$\lvert V \rvert$	$\lvert E \rvert$	$\lvert[\cong_{\mathrm{FP}}]\rvert$
NotreDame	325,729	1,497,134	118,264
Google	875,713	5,105,039	568,044
Stanford	281,903	2,312,497	182,349

Table 3.1: Web graphs

Graph	$\lvert V \rvert$	$\lvert E \rvert$	$\lvert[\cong_{\mathrm{FP}}]\rvert$
CA-AstroPh	18,772	396,160	14,742
CA-CondMat	23,133	186,936	17,135
CA-GrQc	5,242	28,980	3,394
Email-Enron	36,692	367,662	20,417
Email-EuAll	265,214	420,045	28,895
Wiki-Talk	2,394,385	5,021,410	566,846
Wiki-Vote	7,115	103,689	5,805

Table 3.2: Social network graphs

Graph	$\lvert V \rvert$	$\lvert E \rvert$	$\lvert\Sigma\rvert$	$\lvert[\cong_{\mathrm{FP}}]\rvert$
1 Specific properties en	609,014	819,764	71	236,235
2 Types ru	642,340	642,364	1	79
3 Types es	818,657	819,780	1	336
4 Types de with en	618,708	1,810,909	1	335
5 Identica	16,355	29,683	12	14,588
6 Jamendo	438,975	1,047,898	25	396,725

Table 3.3: RDF graphs

EuAll, Wiki-Vote, Wiki-Talk), web graphs (NotreDame, Google, Stanford) and Co-Authorship networks (CA-AstroPh, CA-CondMat, CA-GrQc). Even if they were advertised as undirected, we considered all of them to be lists of directed edges, to improve the comparability with the other compressors, as these would also assume the input graph to be directed.

The RDF graphs mostly come from the DBPedia project[3], which is an effort of representing ontology information from Wikipedia. We evaluate on *specific mapping-based properties* (English), which contains infobox data from the English Wikipedia and *mapping-based types*, which contains the rdf:types for the instances extracted from the infobox data. We use three different versions of the latter: types for instances extracted from the Spanish and Russian Wikipedia pages that do not have an equivalent English page, and types for instances extracted from the German Wikipedia pages that do have an equivalent English page. The Identica-dataset[4] represents messages from the public stream of the (no longer active) microblogging site identi.ca. Its triples

[3]`http://wiki.dbpedia.org/Downloads2015-04`
[4]`http://www.it.uc3m.es/berto/RDSZ/`

Graph	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\Sigma\rvert$	$\lvert[\cong_{\mathrm{FP}}]\rvert$
Tic-Tac-Toe	5,634	10,016	3	9
Chess	76,272	113,039	12	74,592
DBLP60-70	24,246	23,677	1	2,739
DBLP60-90	658,197	954,521	1	207,305

Table 3.4: Version graphs

map a notice or user with predicates such as creator (pointing to a user), date, content, or name. The Jamendo-dataset[5] is a linked-data representation of the Jamendo-repository for Creative Commons licensed music. Subjects are artists, records, tags, tracks, signals, or albums. The triples connect them with metadata such as names, birth date, biography, or date.

Version graphs are disjoint unions of multiple versions of the same graph. Here, Tic-Tac-Toe represents winning positions, and Chess represents legal moves[6]. The files contain node labels from a finite alphabet, which we ignore here. DBLP60-70 and DBLP60-90 are co-authorship networks from DBLP, created from the XML[7] file by using author IDs as nodes and creating an edge between two authors who appear as co-authors of some entry in the file. To make version graphs, we created graphs containing the disjoint union of yearly snapshots of the co-authorship network.

3.6.2 Influence of Parameters

We evaluate how the different parameters for our compressor affect compression. For these experiments every parameter except the one being evaluated is fixed for the runs. Note that this sometimes leads to situations where none of the results in a particular experiment represents the best compression our compressor is able to achieve for the given graph. The parameters evaluated are the maximum rank of a nonterminal and the node order.

Maximal Rank We test maxRank values from 2 up to 8. The average results for the three types of graphs tested are given in Table 3.5, as compression in bpe. We did some tests for higher values (up to 16) but only got worse results. The best results are marked in bold. For each class of graphs, a value of 4 achieved the best results on average. We therefore conclude that a value of 4 is a good compromise for our data sets.

Node Order Figure 3.13 shows the compression ratio of a selection of graphs under the different node orders detailed in Section 3.3.1 above. The selection aims to be representative for the graphs of the types we evaluated: CA-graphs behave similar to CA-AstroPh, version graphs similar to DBLP60-70, and the RDF graphs similar to Specific properties en. The other graphs in the figure are chosen because they are outliers in their respective category. Our FP-order

[5]`http://dbtune.org/jamendo/`

[6]Both from `http://ailab.wsu.edu/subdue/download.htm`

[7]`http://dblp.uni-trier.de/xml/` (release from August 1st, 2015)

	2	3	4	5	6	7	8
Network Graphs	11.74	11.59	**11.44**	11.94	12.51	13.67	12.92
RDF Graphs	5.17	5.40	**4.85**	4.92	5.60	6.12	6.25
Version Graphs	7.99	8.21	**5.57**	5.93	6.07	6.12	6.11

Table 3.5: Average results for different values of maxRank (compression in bpe).

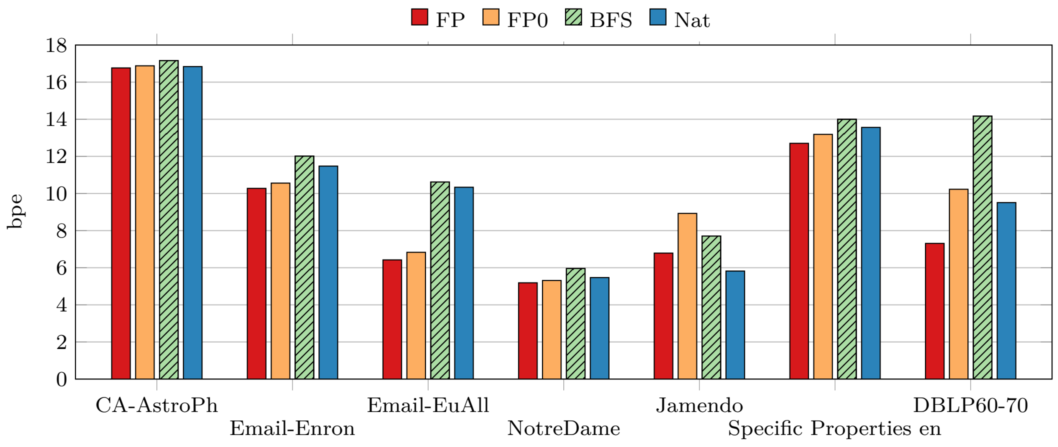


Figure 3.13: Performance of gRePair under different node orders.

achieves the best result on almost all of the graphs. On RDF graphs the order generally had only marginal impact: the best and worst results usually are within 1 bpe of each other. The Jamendo graph presents an exception here, with the natural order being about 1 bpe better than the closest other result and featuring the largest difference between FP0 and FP of all our graphs. Version graphs however benefit hugely from the FP-order, as evidenced by the results for DBLP60-70, and further discussed in Section 3.6.3. This shows that two or more versions of the same graph are similarly ordered in the FP-order, increasing the likelihood of the compressor of recognizing repeating structures.

There is another interesting observation about the FP-order, or in particular the equivalence relation $\cong_{\text{FP}}$. It is likely that nodes with a high similarity, i.e., the same neighbourhood up to a certain distance, are equivalent in this relation. This implies that graphs with a low number of equivalence classes should compress well, as they would have many repeating substructures. Figure 3.14 shows this correlation. There is no graph in the lower right corner, i.e., there is no graph with a low number of equivalence classes and low compression. Note that this is neither a necessary nor a sufficient condition for good compression, but merely an indication. For example, complete graphs always have only one equivalence class but do not compress well with gRePair.

3.6.3 Comparison with other compressors

We compare gRePair with the compressors k^2-tree, LM, and HN listed at the beginning of Section 3.6. Note that we compare RDF graph compression only against the k^2-tree-

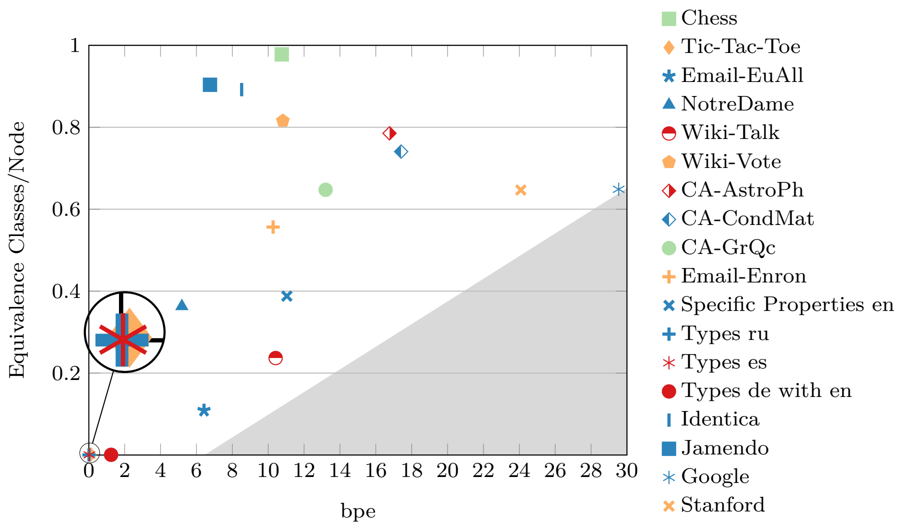


Figure 3.14: Scatter plot showing the dependency between equivalence classes of $\cong_{\mathrm{FP}}$ and compression.

method, as LM and HN have not been extended to RDF graphs (they only work with graphs without edge labels). While these algorithms all work as in-memory data structures, they produce outputs with file sizes comparable to the in-memory representations. We measure the compression performance based on these file sizes. Where applicable, we furthermore use the dense substructure removal done as a first step in the HN-method in combination with our compressor, marked as “gRePair+DSR”. A *dense substructure* in this case is defined by two sets of nodes U, S that induce a complete bipartite graph. Note that U and S need not be disjoint. The edges in these bicliques are replaced by a single “virtual node” with incoming edges from the nodes in U and outgoing edges to the nodes in S. Usually, the virtual nodes would be identified by their ID: ID’s higher than $|V|$ (with respect to the input graph) are virtual nodes. As our compressor renames the nodes, this is no longer feasible. Therefore, we added specially labelled rank-1 edges to the virtual nodes created by the dense substructure removal, to ensure that the original graph could be restored. This is the most space-efficient method we found.

Let us give an idea of the compression ratio for the graphs according to our size definition. Using gRePair (without DSR), we achieve, on average, a compression ratio ($\frac{|G|}{|g|}$) of

- 83% for web graphs,
- 68% for network graphs,
- 35% for RDF, and
- 24% for version graphs.

The parameters we choose for gRePair are maxRank $= 4$ and the FP-order, both being generally the best choice for our dataset. We note that in most results the majority of the file sizes of gRePair’s output ($> 90\%$) is for the representation of the start graph.

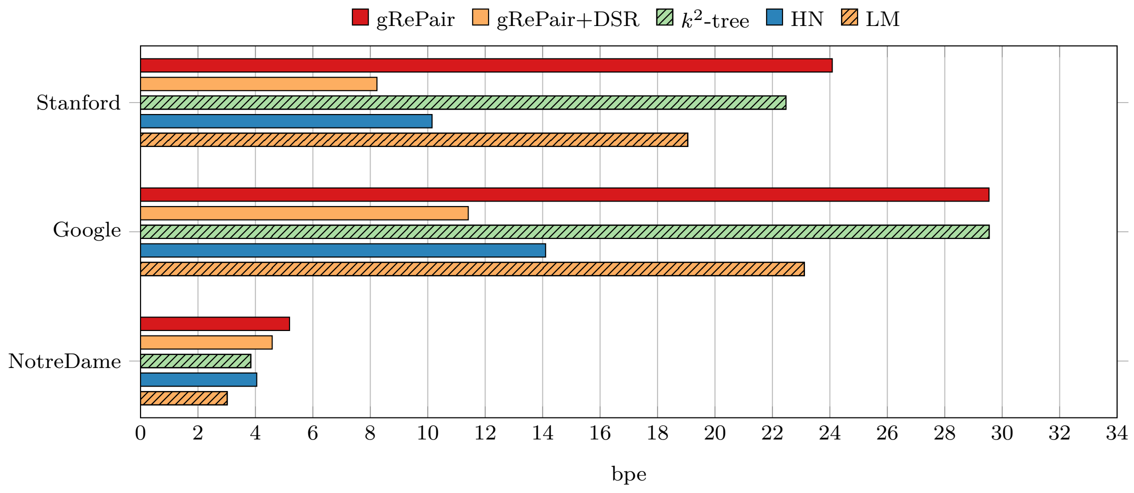


Figure 3.15: Web graph comparison of gRePair with three other compressors.

Network Graphs The methods we compare against are all designed with compression of web graphs in mind, though particularly the HN-compressor also was applied successfully to social graphs. Our results on web graphs compared to k^2-tree, LM, and HN are summarized in Figure 3.15. Clearly, by itself gRePair is not a good choice for the compression of web graphs. Only with the NotreDame graph are the results close and even then gRePair produces the least compression. However, combining the dense substructure removal with our compressor as detailed above improves the results significantly and leads to the best results on the Google and Stanford graphs. They are also better than the best results reported in [Mas12] for these two graphs (11.58 bpe and 9.88 bpe, respectively), which otherwise are the second-best results out of the compared compressors.

Our results on social network graphs compared to k^2-tree, LM, and HN are summarized in Figure 3.16. We improve on the plain k^2-tree-representation on all graphs. However, our results are often slightly worse than LM and HN, with Email-EuAll and CA-GrQc being exceptions. Our result for Email-EuAll is also significantly better than the 22.55 bpe reported in [Mas12]. That being said, combining dense substructure removal with our compressor, generally improves on our results and achieves the smallest bpe-values for two of the three CA-graphs. These are likely improved further by the method of [Mas12], however, as they report 6.69 bpe on CA-CondMat.

While not all the results are mentioned here, we conducted experiments on most of the graphs given in [Mas12] for their evaluation of MP_k linearizations. It is interesting to note that gRePair performs well on the graphs MP_k linearizations perform worst on. They attribute the worse performance on these graphs to a lack of locality. Specifically, these graphs have a very low "global clustering coefficient". Intuitively this is a measure for the likelihood of an edge existing between two nodes, if they have a common neighbour (cf. [Mas12, Section 4.1]). Our method performs better on graphs exhibiting less such interconnection.

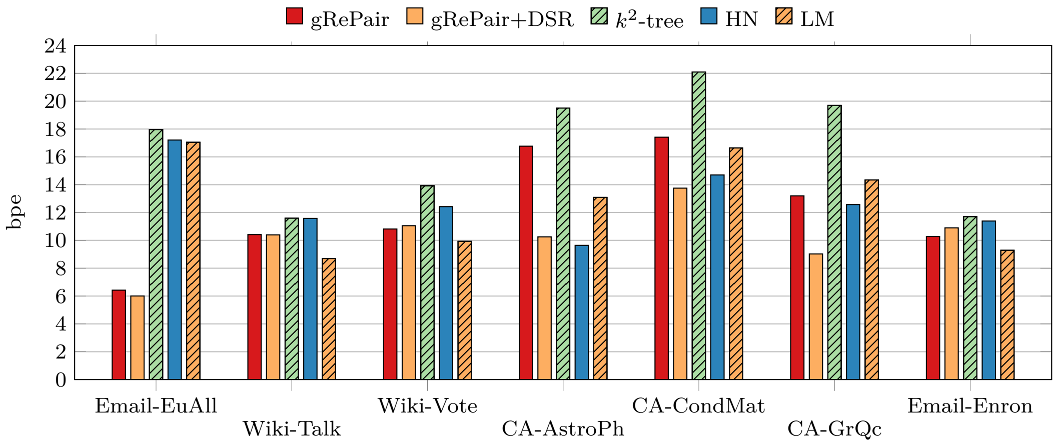


Figure 3.16: Social network graph comparison of gRePair with three other compressors.

RDF-Graph	1	2	3	4	5	6
gRePair	11.04	0.01	0.03	1.24	8.52	6.76
k^2-tree	25.59	10.62	7.83	4.53	14.61	5.32

Table 3.6: Results on RDF graphs (in bpe)

RDF Graphs Recall from Section 2.5 that the values for subject, predicate, and object of RDF triples are commonly mapped to integers using a dictionary to represent the original values. As in this way dictionary and graph are separate entities, we only focus on compressing the graph. Any method for dictionary compression can be used to additionally compress the dictionary (e.g. [MFC12]) and we omit the space necessary for the dictionary.

Our results in comparison to k^2-tree are given in Table 3.6. We greatly improve against this representation on all graphs except for the Jamendo graph. For the graphs 2 – 4 (in particular 2 and 3) we are able to produce a representation that is orders of magnitude smaller than the k^2-tree-representation (note that $|[\cong_{\text{FP}}]|$ is very low for these graphs). For these two graphs in particular, this is due to the majority of their nodes being laid out in a star pattern: a few hub nodes of very high degree are connected to nodes, most of which are only connected to the hub node. Furthermore, while not acyclic, the graphs are also very tree-like. Structures like these are compressed well by gRePair, because every iteration of the replacement-round of gRePair roughly halves the number of edges around the hub node.

Version Graphs We describe several experiments over version graphs. First we study how the compressor behaves given a high number of identical copies of the same simple graph. The graph in this case is a directed cycle with four nodes and one of the two possible diagonal edges. Figure 3.17a shows the results of this experiment for identical copies starting from 8 going in powers of 2 up to 4096. Clearly, gRePair is able to compress much better in this case ("exponential compression"), while the file size of other methods rises with roughly the same gradient as the size of the graph. Note that both axes in this graph use a logarithmic scale: in

	TTT	Chess	DBLP60-70	DBLP60-90
gRePair	0.12	9.06	7.31	12.10
k^2-tree	4.38	8.54	16.62	17.98
LM	-	-	15.71	17.75
HN	-	-	13.17	15.41

Table 3.7: Results on version graphs (compression in bpe)

this case, gRePair produces a representation that is orders of magnitude smaller than the other compressors.

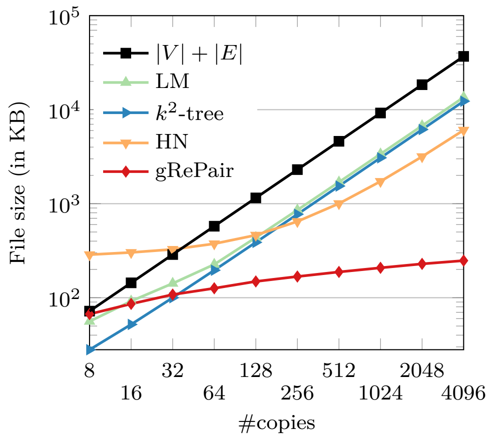
Except for identical copies of rather simple graphs, however, we cannot expect to achieve exponential compression on version graphs. Every version has changes and it is not easy to decide which parts of two versions remain the same and can thus be compressed using the same nonterminals. Even if we can guarantee that the same (or rather, isomorphic) substructures are consistently compressed in the same way, the changes between versions might be too big to allow for exponential compression. Our FP-order is inspired by the Weisfeiler-Lehman method [WL68] (see also [CFI92]), which approximates a test for isomorphism. The results on version graphs, when comparing different orders (see also Section 3.6.2 above), suggest that this is indeed exploited. Figure 3.17b shows a comparison on the compression of a version graph from the DBLP co-authorship network. We started with a co-authorship network including publications from 1960 and older. To this graph we then add versions with the publications from 1961, 1962,. . . until 1970 and compress the graphs obtained in this way. The comparison shows that using the FP-order our method achieves better compression than using other orders. Note that the results for BFS or natural order are much closer to k^2-trees. Our full results for version graphs are given in Table 3.7. Note that we compare Tic-Tac-Toe and Chess only against k^2-tree, because these graphs have edge labels. The results show that gRePair compresses version graphs well.

3.6.4 Results on Synthetic Graphs

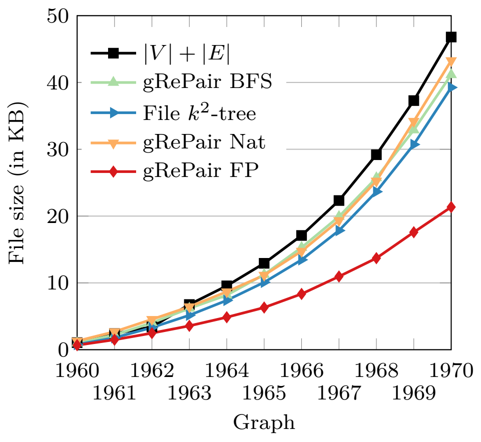
We finally describe experiments on synthetic graphs. These show some of the effects described earlier, namely that

1. some graphs can be compressed exponentially,
2. the maximal rank can have a big impact, and
3. different node orders can have a big influence on the compression.

We evaluate two different families of graphs: "grid" and "triangle fractal". Both have a parameter $n \in \mathbb{N}$ to achieve different sizes. Let grid_n be an $n \times 2^n$-grid graph, i.e., a graph with $n \cdot 2^n$ nodes where node i has edges to $i+1$ (unless $i \equiv 0 \mod 2^n$), and to $i + 2^n$ (unless $i + 2^n > n \cdot 2^n$). Following this construction, grid_n has $n \cdot 2^n$ nodes, and $(n-1) \cdot 2^n + n \cdot 2^{n-1}$ edges, yielding a size of $|\mathsf{grid}_n| = n(2^n + 2^{n-1}) + (n-1) \cdot 2^n$. Furthermore we define a triangle fractal in the



(a) Compression of disjoint unions of the same synthetic graph with 4 nodes and 5 edges.



(b) Using different node orders for the compression of a version graph.

Figure 3.17: Results on version graphs

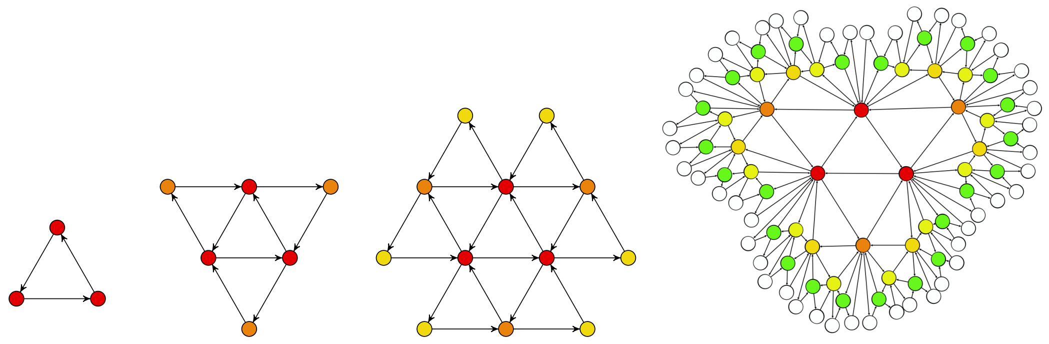


Figure 3.18: The tf_1, tf_2, tf_3, and tf_6 triangle fractals. The node colours indicate the layer of the node: red nodes are tf_1, dark orange is added in tf_2, and every step towards green means the nodes are added in the next iteration, with white being the nodes added in tf_6 from tf_5.

following way: initially tf_1 is a complete graph with 3 nodes. To define tf_i with $i > 1$, we start with tf_{i-1} and let X be the set of edges in tf_{i-1} that are incident to a node of degree 2. For each edge e in X we add a new node v and add edges from the target of e to v and from v to the source of e, creating another triangle. Intuitively, we add another triangle at every outer edge. Regarding the size, t_n has $|t_n|_V = 2^n + 2^{n-1}$ nodes, and $|t_n|_E = 2|t_n|_V - 3$ edges. The natural order (i.e., the order of the node-IDs) is a top-to-bottom, left-to-right order for grid_n, while the natural order for tf_n is to start with the three nodes of the innermost triangle and then go outward from there in the same way as generated (i.e., in "layers").

Table 3.8 lists the results for these graphs using various orders, graph sizes, and values of maxRank. Here we state the compression performance in relation to the size definition, i.e., the compression ratio is given by $\frac{|G|}{|g|}$ (in percent), where G is the compressed representation of g. Some notable results: triangle fractal compresses best at maxRank $= 2$. This is not surprising, as in this case the construction of the graph is exactly reversed (with respect to the recursive definition of tf_n). However, note that at maxRank $= 4$ the results are still very close using the

		Triangle fractal			Grid		
Order	MaxRank	$n = 4$	$n = 8$	$n = 12$	$n = 4$	$n = 8$	$n = 12$
FP	2	36.23%	4.61%	0.44%	98.26%	99.95%	100.00%
	4	46.38%	5.40%	0.50%	100.00%	95.95%	97.15%
	15	85.51%	23.59%	9.43%	100.00%	72.31%	46.38%
	∞	85.51%	24.80%	6.23%	100.00%	70.00%	33.99%
FP0	2	36.23%	4.61%	0.44%	98.26%	99.95%	100.00%
	4	46.38%	16.54%	5.70%	100.00%	92.23%	93.37%
	15	46.38%	61.10%	8.99%	100.00%	99.93%	96.99%
	∞	46.38%	61.10%	8.99%	100.00%	99.93%	96.99%
Nat	2	39.13%	4.79%	0.45%	98.26%	99.95%	100.00%
	4	100.00%	82.77%	80.69%	99.42%	95.88%	97.15%
	15	95.65%	28.72%	5.19%	94.77%	62.96%	75.48%
	∞	95.65%	28.72%	4.51%	94.77%	13.13%	1.23%
BFS	2	60.87%	18.28%	7.49%	98.26%	99.95%	100.00%
	4	81.16%	54.22%	56.68%	100.00%	95.88%	97.15%
	15	81.16%	57.18%	72.33%	100.00%	47.81%	37.89%
	∞	81.16%	52.13%	71.11%	100.00%	63.74%	(19.85%)

Table 3.8: Results on triangle fractal- and grid-graphs for various values of n, four orders, and 4 different values of maxRank.

FP-order, whereas they get noticeably worse for every other order. Indeed, with the BFS-order even for maxRank = 2 the optimal result is not found any more. Using a higher rank than 4 is generally detrimental for this graph, because digrams of higher rank occur more often than the ones of rank 2. This is not a problem in itself, as carefully chosen occurrences can still be reduced well, but there are many more sets of occurrences for rank-3 digrams, than there are for rank-2 digrams. Which one is found by gRePair depends on the node order. This graph family shows that

1. maxRank can have a huge impact (compare 2 vs. 4 for the natural order), and

2. the node order can have a huge impact (compare FP vs. BFS order).

Grids are the opposite, when it comes to maxRank: best compression is achieved for unbounded rank, while maxRank = 2 gives almost no compression (independent of the node order). For unbounded rank, FP0 gives close to no compression, FP compresses to 34%, and natural order compresses to 12% (for $n = 12$). It is possible to achieve "exponential compression" with grids, by treating the grid as n identical lines of length 2^n. Node orders that traverse the grid in such a way (like the natural order, which goes row by row, or the BFS-order which follows the rows "in parallel"), are likely to find digram occurrences corresponding to this structure. Accordingly, the BFS-order also achieves good compression. In particular, it has the best result for maxRank = 15 with $n = 8$ or $n = 12$. Note however, that for grid_{12} with BFS-order and unbounded maxRank the rank was actually bounded to 1500, because the computation takes too long otherwise.

3.7 Related Work

Context-free hyperedge replacement grammars are due Habel/Kreowski and Bauderon/Courcelle [HK86, BC87, Hab92], and also comprehensively discussed in, e.g. [Eng97, DKH97]. An approximation algorithm for finding a small HR grammar that generates a given graph is considered already by Peshkin [Pes07]. It is based on the SEQUITUR compression scheme [NW97]. However, the main aim of their work is to gain insight into the structure of a graph, rather than compression. Experiments are thus only presented for rather small protein graphs, and while the implementation is available[8], it does not consider succinct encoding of the output. As far as we know, no other compressor for straight-line graph grammars has been considered. Claude and Navarro [CN10] apply string RePair on the adjacency list of a graph. This works well, but is outperformed by newer compression schemes such as k^2-trees [BLN14]. More database oriented work is found for semi-structured data. For example the XMill-compressor [LS00] groups XML-data such that a subsequent use of general-purpose compression (e.g. gzip) is more effective. Schema information can improve its effectiveness, but is not required. Deriving schema information from existing data can be seen as a form of lossy compression. DataGuides [GW97] are a way of doing just that for XML data. As XML documents can be represented as trees, methods to compress trees are applicable. Grammar-based tree compression can be seen as a precursor to the work presented here. One of the first such algorithms was BPLEX [BLM08]. The results of BPLEX were later improved by applying the RePair compression scheme to trees [LMM13].

3.7.1 Succinct Graph Representations

A succinct data structure typically is an encoding that represents the data within the information theoretic lower bound using only $o(n)$ additional bits and often supporting a reasonable set of operations on the data. For graphs, these are typically adjacency or neighbourhood queries. This section summarizes some theoretic (to our knowledge no implementations of these algorithms exist) work on succinct graph representations. Most of these methods only work on subsets of graphs, we detail for each method which subset exactly it works for. Furthermore, “graphs” in the context of these algorithms are directed simple graphs without labels unless otherwise stated. For such a graph with n nodes and m edges the information theoretic lower bound (according to [FM13]) is $\mathcal{H}(n,m) = \lceil \log \binom{n^2}{m} \rceil$ bits.

Turán [Tur84] presents a succinct representation for planar graphs. Specifically it encodes undirected planar graphs without loops or multiple edges into a bitstring with at most $12n$ bits. Furthermore an undirected planar graph with node labels can be represented by $n\lceil \log_2 n \rceil + 12n$ bits. This is achieved by first choosing a spanning tree of the graph. The nodes of the graph are then arranged into a cyclic sequence according to a post-order traversal of the spanning tree

[8] `https://github.com/Peshkin/Graphitour`

(length: $2n-2$). The remaining edges (i.e. the ones not part of the spanning tree) are included as diagonals of the cycle. This is where planarity is necessary, as otherwise these diagonals might cross. The sequence is then encoded as a string consisting of the symbols $+, -, (,)$, which can be encoded by 2 bits each, leading to an encoding of at most $12n-24$ bits. In the labelled case the same encoding is used, padded to the maximum amount of bits, followed by a dictionary for the labels. Access to the compressed structure is not considered.

Jacobson's approach [Jac89] works on graphs of bounded pagenumber. This means that the graph can be represented by a bounded number of pages, which is a linear drawing of a subset of nodes with the edges connecting them drawn above so that they do not intersect. Such a page can be encoded as a string of parentheses, leading to an encoding for the full graph by combining the encodings of the single pages. He proves it to be optimal for trees and within a logarithmic factor of the information theoretic lower bound for graphs of bounded pagenumber. Jacobson also considers accessing the succinct structure and shows that an algorithm can test adjacency of two nodes in $O(\log n)$ bit inspections for graphs with one page. For graphs with k pages, Jacobson's approach uses $O(kn)$ bits and tests adjacency in $O(k \log n)$ time.

Deo and Litow [DL98] consider graphs of bounded genus g (a topological embedding similar to the pagenumber above) and present an encoding representing such graphs using $O(g+n)$ bits and $O(g+n)$ time. However, finding a minimal genus embedding for a graph G is NP-complete, but fixed-parameter tractable, therefore finding an embedding for a fixed genus g is possible in P. They furthermore show separator theorems for graphs defined by an excluded minor. These and similar separator techniques are later used for compression, as further mentioned below.

The approach by Farzan and Munro [FM13] works for arbitrary graphs. They first prove that it is not possible to achieve a representation within the information theoretic lower bound up to lower order terms unless the graph is either too sparse (i.e. $m = o(n^\delta)$ for any constant $\delta > 0$) or too dense (i.e. $m = \omega(n^{2-\delta})$ for any constant $\delta > 0$). They then present a succinct encoding supporting adjacency and neighbourhood queries in constant time. The space needed for the encoding is within a $(1+\varepsilon)$-factor of the information theoretic lower bound for any arbitrarily small constant $\varepsilon > 0$, and it tightly achieves the information theoretic lower bound within lower order terms for too sparse or too dense graphs.

Lu [Lu14] gives a compression scheme for hereditary classes of graphs. A class of graphs is hereditary, if it is closed under taking subgraphs. Let $\mathbb{G}$ be such a class of graphs, and let $\mathsf{num}(\mathbb{G}, n)$ be the amount of graphs with n nodes in $\mathbb{G}$. They prove that, if $\log \mathsf{num}(\mathbb{G}, n) = O(n)$, given a graph $G \in \mathbb{G}$ with n nodes and a genus-$o(\frac{n}{\log^2 n})$ embedding, it is possible to represent the graph using $\beta n + o(n)$ bits, where β is any positive constant such that $\log \mathsf{num}(\mathbb{G}, n) \leq \beta n + o(n)$. This is achieved by using graph separators as above.

3.7.2 Web Graph Compression

Several compression approaches have been developed particularly for web graphs. However, some have been successfully applied to other graphs as well, which we mention where appropriate.

In a web graph, nodes represent pages (i.e., URLs) and edges represent links from one page to another. Web graphs have two properties which are useful for compression:

Locality: most links lead to pages within the same host (i.e., URLs have the same prefix) and

Similarity: pages on the same host often share the same links.

Due to these properties, ordering the nodes lexicographically by their URL provides an order in which similar nodes are close to each other. The WebGraph framework [BV04] by Boldi and Vigna is originally based on this order, but was later improved with a different order [BSV09]. It represents the *adjacency list* of a graph using several layers of encodings, while retaining the ability to answer out-neighbourhood queries. A different approach is proposed by Grabowski and Bieniecki [GB14], where contiguous blocks of the adjacency list are merged into a single ordered list, and a list of flags which are used to recover the original lists. They then encode the ordered list and use the deflate-compressor to compress both lists. To our knowledge, their method is the current state-of-the-art in compression/query trade-off, when only out-neighbourhood queries are considered and the compression ratio is prioritized.

The methods above have in common that they encode the adjacency list of a graph and natively only support out-neighbourhood queries. The k^2-trees of Brisaboa et al. [BLN14] on the other hand compress the *adjacency matrix* of the graph. As briefly explained in Section 3.2.3, they do this by recursively partitioning it into k^2 many squares. If one of these includes only 0-values, then it is represented by a 0-leaf in the tree, and otherwise the square is partitioned further. This Quadtree-like representation is well known (see, e.g. [Ši09]), but their succinct binary encoding is a clever new approach. The method provides access to both, in- and out-neighbourhood queries, and can be applied to any binary relation. The k^2-tree-method was combined by Hernández and Navarro [HN14] with dense substructure removal, originally proposed by Buehrer and Chellapilla [BC08]. This also works well on general graphs, and to our knowledge, the method of [HN14] is the current state-of-the-art in compression/query trade-off, when in- and out-neighbourhood queries are considered and the compression ratio is prioritized. For graphs that are "tree-like" in the sense that there are not many edges in addition to those in a spanning tree, Fischer and Peters [FP16] propose a method called "GLOUDS", which is based on the LOUDS encoding [Jac89] to succinctly represent trees. Informally, the LOUDS encoding is used to represent a spanning tree and enhanced with additional information for the remaining non-tree edges.

3.7.3 Beyond Web Graphs

Web graph compression relies strongly on properties commonly found in web graphs, but not common to other graphs. These methods therefore tend to work less well on, for example, social networks. Some work on compression of general graphs exists, but it is less common than methods focussed on web graphs. One immediate problem is the node order. Outside of web graphs, a lexicographical order of the node-names is not necessarily possible or useful. Apostolico and Drovandi therefore propose an encoding of the adjacency list based on a BFS-order [AD09].

This method provides competitive compression while supporting out-neighbourhood queries, but their evaluation still focusses on web graphs.

Maserrat [Mas12] presents lossless and lossy compression schemes which support both, in- and out-neighbourhood queries, and also more efficient incremental updates than the above, based on MP_k-linearizations. These schemes are mainly evaluated on social networks achieving between 5 and 30 bpe. On two web graphs the methods achieves around 11 bpe, suggesting that it is indeed more suitable for general network graphs, as the WebGraph framework achieves better results (but only supporting out-neighbourhood queries).

Claude and Ladra [CL11] combine the k^2-tree representation with two other methods. First they use the method by Claude and Navarro [CN10] using RePair on the adjacency lists of a graph to compress web graphs. In addition to that, they also combine k^2-tree with the previous method of MP_k-linearizations to compress some social networks, but the results are not presented in detail. In both cases, this method retains the ability to query for both, in- and out-neighbours.

3.7.4 RDF Graph Compression

As previously (Section 2.5) mentioned, RDF is a specification used to represent linked data. As such there exists previous work on compressing this data. The splitting into graph structure and dictionary, now commonly used, was first proposed by Fernández et al. [FMG10] as a data structure called HDT (Header-Dictionary-Triples). Together with an encoding of the triples grouping them by subject as an adjacency list, this yields an in-memory representation of RDF data. They also investigate compressing this data structure using conventional general-purpose compression methods (gzip, bzip2, and ppmdi), beating these universal compressors used on the plain original file. Generally this leads to two approaches of RDF graph compression: one can either compress the dictionary (see, e.g. [MFC12, UMD+13]), or the underlying graph structure, which the present work focusses on.

It may be a natural idea to apply general graph compression methods to RDF graphs, but out of the methods mentioned in the previous sections, only the k^2-tree has been used for RDF graph compression in two ways [ÁBF+15, ÁdBBN17]. The first encodes a separate adjacency matrix for every predicate (i.e., edge label) in the graph in the same way we use the k^2-tree method for representing the start graphs of SL-HR grammars. The second method starts out this way as well, but then proceeds to store the resulting tree in one big "interleaved" k^2-tree, instead of $|\Sigma|$ separate ones. They also give and evaluate algorithms for common queries on RDF data. Another data structure mainly concerned with finding a good trade-off between compression and querying performance is due to Brisaboa et al. [BCFN15]. They propose the use of an index structure based on compressed suffix arrays, which does not always save space compared to other RDF representations, but is significantly faster.

A different approach to RDF graph compression is by using the semantics: some triples may be semantically redundant and can therefore be removed without losing any data. One such method by Iannone et al. [IPR05] focuses on finding blank nodes that represent redundant data and how to merge them with either named or other blank nodes. Meier [Mei08] and Pichler et al. [PPSW10] describe two rule-based approaches to remove redundant triples, but both methods require user input, and are therefore not appropriate for general purpose compression of large datasets. To our knowledge the first work investigating the automated removal of general triples is by Joshi et al. [JHD13] where a frequent itemset mining approach is used to split the dataset into two separate sets of triples. The so called "dormant" partition contains all the triples that could not be compressed, the "active" partition contains triples which can be used to recreate the deleted triples using rules created from the frequent itemsets. Jiang et al. [JZG$^+$13] introduce two compression schemes ("equivalent" and "dependent" compression) to compress annotated graphs called *Typed Object Graphs.* These are RDF-graphs with type information attached to each node. Thus the edges are quintuples in this model instead of the usual RDF-triples, as subject and object also get a type attached. The equivalent compression aims to combine nodes that have the same type and the same family (i.e. the same neighbours and the same relations to these) into one node. Dependent compression searches for nodes that have only one neighbour and combines the information stored in this node with the information stored in the node of their only neighbour. Their evaluation shows that the dependent compression achieves consistently better results. In a somewhat similar way, Pan et al. [PGR$^+$14] also focus on compressing the graph structure by mining for redundant graph patterns. They describe two methods of removing redundancies:

- By logical compression they mean the search for patterns that appear often and can be replaced by a generalized triple. For example, if the pattern

  $$<?x, a, \text{foaf: Person}>, <?x, a, \text{dbp: Person}>$$

  appears often, a type T could be introduced, along with a rule to expand T to foaf: Person and dbp: Person. Then the single triple $<?x, a, T>$ would represent the above pattern.
- RDF files are a textual representation of the graph structure. As there are many ways of serializing such a graph, some of these serializations are more concise than others. They therefore describe a way of using graph patterns to group triples in such a way that they can then be serialized more concisely, resulting in shorter files.

The two methods are joined by a pattern mining phase at the beginning to find the patterns which are used to achieve the compression. Then all three steps (pattern mining, logical compression, serialization) can be iterated to achieve compression.

Another approach of Swacha and Grabowski [SG15] combines techniques to compress graph and dictionary achieving a succinct representation of the RDF graph, which is subsequently compressed by general purpose compressors.

3.8 Discussion and Further Research

In this chapter a generalization of the RePair compression scheme to (hyper)graphs using hyperedge replacement graph grammars is presented. Some theoretical results about SL-HR grammars are proven, for instance that they cannot compress trees or strings (represented as graphs) better than the existing native grammars, despite being more expressive.

A prototype implementation of the algorithm is then studied and experimentally evaluated. It uses a greedy approximation to get a list of non-overlapping digrams, which is heavily influenced by the node order (as also experimentally verified). It is shown that a node order inspired by the Weisfeiler-Lehman method to test for isomorphism [WL68] works well in practice. The experimental evaluation is inconclusive in parts: on network graphs (without edge labels) the prototype sometimes outperforms the other compressors we compared to, but in most cases does not. On edge-labelled RDF graphs however, very good results are achieved by gRePair, sometimes even orders of magnitude better than other methods.

There are several directions for interesting future work on this topic. Firstly, investigating the use of NR graph grammars could be fruitful. As these are more expressive than HR grammars, particularly when it comes to complete graphs, it is possible that a RePair variant using SL-NR grammars yields better results. However, this would need to be balanced with more complex rules, which are more expensive to store due to the addition of connection relations.

For the HR grammar based compressor presented here, there are also some open questions. A further investigation into node orders would be interesting, as this could improve compression particularly on version graphs. Another area is the encoding of hypergraphs. The k^2-tree-based method used here is unlikely to be the best way encoding hypergraphs, as it generates many large but sparse matrices. However, even staying with it, there is a possibility that it could be improved further by reordering the nodes in the start graph again before encoding. Experiments have shown that k^2-tree tends to work better after ordering the nodes with a BFS ordering. Finally, the prototype implementation used to evaluate the algorithm is rather poor with regards to runtime and memory usage. A proper implementation is necessary to make the method viable, the work here shows that this may be a useful endeavour.

Chapter 4

Traversal of Compressed Graphs

In the previous chapter we discussed a method to compute compressed SL-HR grammar representations of a given graph. These representations can be exponentially smaller than the graph they represent. It is therefore not feasible to decompress the original graph, whenever a query occurs. Instead, we now focus on methods to directly query $\mathsf{val}(G)$ using only the grammar G (and potentially additional space polynomial in $|G|$). In this and the next chapter we present two types of algorithms to do so. First we look into neighbourhood and adjacency queries. Neighbourhood queries are queries where, given a node x of $\mathsf{val}(G)$, we wish to compute the neighbourhood $N(x)$ of x using the grammar. Adjacency queries were already mentioned in Section 2.6 and mean queries where, given two nodes x and y of $\mathsf{val}(G)$, we determine, whether there exists an edge between them. Adjacency queries can be reduced to neighbourhood queries by computing whether $y \in N(x)$. We therefore focus on neighbourhood queries. Both of these queries can be restricted to edges of a specific label, which we generally do, as lifting the restriction can be done by running the same algorithm $|\Sigma|$ times. These queries allow for any graph algorithm to be run on G, but this comes at a significant expense in runtime compared to a direct representation. Afterwards in the next chapter we present some speed-up algorithms, which allow to compute certain queries faster than on the uncompressed graph.

As they are used extensively in this chapter, recall from Section 2.3 that we refer to a node u of a derivation tree s as an s-context, denote $\mathsf{rhs}(s[u])$ by $g_{s,u}$, and omit the subscript s if $s = \mathsf{dt}(G)$. For the neighbourhood queries, we present two methods. Both use the same general approach. We navigate along the derivation tree and store the current context such that we can compute two queries from it:

1. for every internal node, its ID within $\mathsf{val}(G)$, and
2. for every external node, the internal node (and the corresponding context) it represents.

We first explain the general notions of this navigation within $\mathsf{val}(G)$ using the derivation tree $\mathsf{dt}(G)$ of G, without discussing data structures and computation. Afterwards, we outline direct algorithms to achieve this navigation in time proportional to the height of the grammar, for any SL-HR grammar. Finally we present a precomputed data structure, which allows to do the individual navigation steps in constant time, but only works on a restricted set of grammars.

Note that the algorithms presented in this chapter are always given only for outgoing simple edges/out-neighbourhood queries. The case for incoming edges/in-neighbourhood queries is algorithmically symmetric (however, runtimes might differ, depending on the underlying data structures). Graphs with terminal hyperedges of rank 3 or greater can also be considered, but the complexity then always rises by a factor r (where r is the largest terminal rank of $\mathsf{val}(G)$).

4.1 Traversing along the Derivation Tree

We first explain how to navigate $\mathsf{val}(G)$ by using the derivation tree $\mathsf{dt}(G)$ of G. Let $G = (N, P, S)$ be an SL-HR grammar. Consider the derivation tree in Figure 4.1 and take note of the red arrows. These show for some external nodes, which internal nodes they represent, i.e., where the graphs will be merged during the derivation steps. We make the following observation: for any derivation tree $s = \mathsf{dt}(A)$, s-context u, and external node x of $g_{s,u}$, there either exists a proper ancestor v of u such that $g_{s,v}$ contains the inner node that is merged with x, or this inner node does not exist in $\mathsf{dt}(A)$. Formally, for an s-context u we define a mapping $\mathsf{internal}_s(u, x)$ as $\mathsf{internal}_s(u, x) = (u, x)$ if x is internal. Otherwise x is the j-th external node of $g_{s,u}$ and there are two cases for u: if $u = \varepsilon$, $\mathsf{internal}_s(u, x)$ is undefined, otherwise u can be written as $v.i$ for some i. Then $\mathsf{internal}_s(u, x) = \mathsf{internal}_s(v, \mathsf{att}(e_i)[j])$, where e_i is the i-th edge in $\mathsf{sib}(E^{\mathsf{nt}}_{g_v})$. As before, if $s = \mathsf{dt}(S)$ we omit the subscript.
Remark. Should S have external nodes we pretend they are internal for the purpose of the $\mathsf{internal}$-mapping. Otherwise there are nodes for which the mapping is undefined even with respect to $\mathsf{dt}(S)$.

Given a context u, an internal node x within g_u, and an alphabet symbol σ, our goal is to compute every context v and node y within g_v, such that there is a σ-labelled edge from $\mathsf{node}(u, x)$ to $\mathsf{node}(v, y)$ in $\mathsf{val}(G)$. We need one more mapping to achieve this. We define $\mathsf{out}(\mathsf{lhs}(u), x, \sigma) = \{(v_1, y_1), \ldots, (v_k, y_k)\}$ where k is the number of outgoing σ-labelled edges from $\mathsf{node}(u, x)$ in $\mathsf{val}(G)$ and y_i (for $i \in [k]$) is a node in g_{uv_i} such that there exists a σ-labelled edge e in g_{uv_i} with $\mathsf{att}(e) = x'y_i$ and $\mathsf{internal}(uv_i, x') = x$. Note that, while we did use the context u for the definition, the result of $\mathsf{out}(A, x, \sigma)$ is independent from the particular context u with $\mathsf{lhs}(u) = A$ used. There may be two contexts u and u' with $u \neq u'$ such that x is a node in both g_u and $g_{u'}$ (they are just right-hand sides of the grammar G). The graph $g_{u'v_i}$ equally includes the node y_i, however, y_i represents two different nodes in $\mathsf{val}(G)$ depending on the two different contexts $u'v_i$ and uv_i. In summary, to compute $N^-_\sigma(\mathsf{node}(u, x))$[1] we

1. compute $\mathsf{out}(\mathsf{lhs}(u), x, \sigma) = \{(u_1, x_1), \ldots, (u_k, x_k)\}$,
2. for every $i \in [k]$ compute $\mathsf{internal}(uu_i, x_i) = (v_i, y_i)$, and
3. compute $\{\mathsf{node}(v_1, y_1), \ldots, \mathsf{node}(v_k, y_k)\}$.

[1] Recall that N^-_σ denotes the out-neighbourhood of a node, restricted to edges labelled σ.

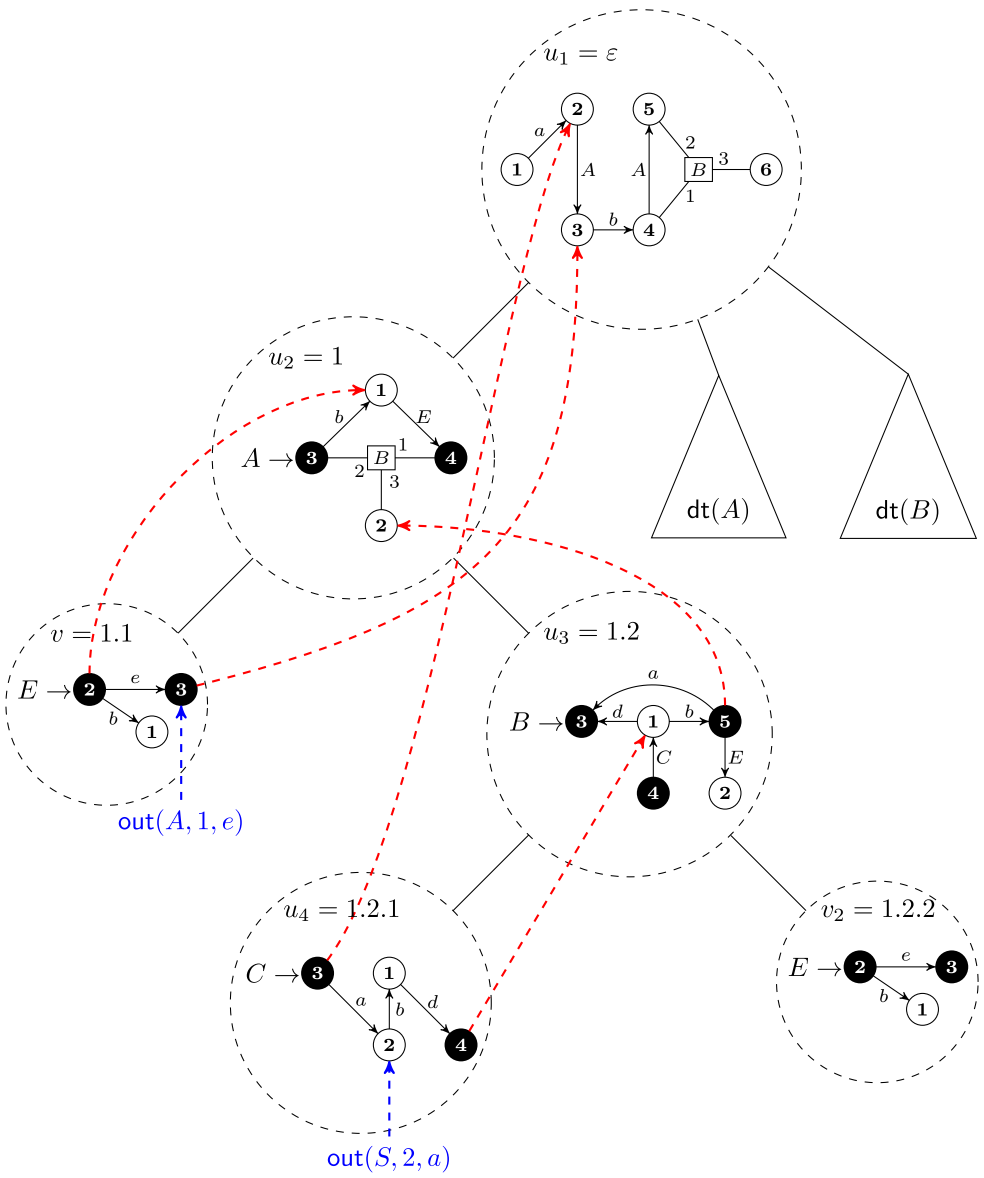


Figure 4.1: Derivation tree of a grammar. Matching internal nodes for some external nodes are marked by red arrows.

Figure 4.1 shows an example of this method. If the current node is 1 in the context u_2, then $\mathsf{out}(A, 1, e) = \{(1, 3)\}$ (note that $u_2.1 = v$). After that $\mathsf{internal}(v, 3) = (u_1, 3)$. Thus, there is an e-labelled edge from $\mathsf{node}(1, u_2) = 7$ to $\mathsf{node}(3, u_1) = 3$ in $\mathsf{val}(G)$.

Note that for any $A \in N$, $x \in \mathsf{rhs}(A)$, and $\sigma \in \Sigma$ the mapping $\mathsf{nodes}(A)$ (see Section 2.3) can be computed in one bottom-up parse of the grammar. This is because it can be defined as $\mathsf{nodes}(A) = \sum\{\mathsf{nodes}(\mathsf{lab}(e)) \mid e \in E^{\mathsf{nt}}_{\mathsf{rhs}(A)}\} + |V_{\mathsf{rhs}(A)}| - |\mathsf{ext}_{\mathsf{rhs}(A)}|$. Provided that the nonterminals are traversed in reverse $\leq_{\mathsf{NT}}$-order, this computation is always well defined. In the following we assume this is done and the result is stored in memory for every nonterminal (using $O(|N|)$ space).

Algorithm 3 Routine to compute a G-representation for a given node-ID x.

Precondition: Every right-hand side of G is normalised.

```
1: function getGRep($x$: node-ID (of val($G$)))
2:     if $x \leq |V_S|$ then
3:         return $(\varepsilon, x)$
4:     else
5:         $i \leftarrow$ smallest number such that first($i$) $\geq x$
6:         $j \leftarrow$ first($i - 1$)
7:         return getGRep($x, i - 1, j$)
8: function getGRep($x$: node-ID, $u$: context, $offSet$: Integer)
9:     $int \leftarrow$ number of internal nodes in $g_u$
10:    if $offSet + int \geq x$ then
11:        $n \leftarrow x - offSet$
12:        return $(u, n)$
13:    else
14:        $i \leftarrow$ smallest number such that first($i$) $\geq x$
15:        $j \leftarrow$ first($u.(i - 1)$)
16:        return getGRep($x, u.(i - 1), j$)
```

4.2 Direct Algorithms

We present direct algorithms to compute the neighbourhood of a node x of $\mathsf{val}(G)$. This is done in two steps:

1. Compute the G-representation (u, y) of x (recall that a G-representation of x is a tuple (u, y) where u is a context of G and y a node in g_u such that $\mathsf{node}(u, y) = x$).
2. Compute the mappings described in the previous section to get the neighbourhood.

Computing the G-representation of x is a straightforward top-down algorithm: starting in the start-graph the algorithm searches for the nonterminal edge that generates x (if $x > |V_S|$) and goes down into the rules from there, repeating the same process in the right-hand sides until y is found. The full algorithm is given as Algorithm 3. Using a binary search in lines 5 and 14 it has a runtime of $O(\log_2 |G|_E \cdot \mathsf{height}(G))$, assuming the uses of the first-mapping in lines 5 and 6 (and lines 14/15) are partially precomputed: first can be computed in $O(h)$ time by using $O(|G|)$ additional space. To do so, for every nonterminal A and every nonterminal edge e within $\mathsf{rhs}(A)$, let i be the position of e in $\mathsf{sib}(E^{\mathsf{nt}}_{\mathsf{rhs}(A)})$. We then let $\mathsf{firstOffset}(A, e) = \mathsf{first}_{\mathsf{dt}(A)}(i)$. Precomputing and storing $\mathsf{firstOffset}$ for every nonterminal edge in G needs $O(|G|)$ additional space, but now $\mathsf{first}(u)$ can be computed in $O(h)$ time, using the formula

$$\mathsf{first}(v.i) = \mathsf{first}(v) + \mathsf{firstOffset}(\mathsf{lhs}(\mathsf{dt}(G)[u]), e_i) + |V_{g_u}| - |\mathsf{ext}_{g_u}|.$$

Moreover, in Algorithm 3, the value of $\mathsf{first}(v)$ in this formula is carried along as a parameter ($offset$). Thus, within this algorithm, first is computed in constant time.

Algorithm 4 Algorithm to compute $\mathsf{out}(A, x, \sigma)$.

```
1: function out($A$: Nonterminal, $x$: node of $\mathsf{rhs}(A)$, $\sigma$: edge label)
2:     $N \leftarrow \{(\varepsilon, y) \mid y \in N^-_\sigma(x)\}$
3:     $E_{\mathsf{nt}} \leftarrow \{e \in E_{\mathsf{rhs}(A)} \mid e$ is nonterminal and $x \in \mathsf{att}(e)\}$
4:     for $e \in E_{\mathsf{nt}}$ do
5:         $i \leftarrow$ position of $x$ in $\mathsf{att}(e)$
6:         $j \leftarrow$ position of $e$ in $\mathsf{sib}(E^{\mathsf{nt}}_{\mathsf{rhs}(A)})$
7:         $N_e \leftarrow \{(j.u, y) \mid (u, y) \in \mathsf{out}(\mathsf{lab}(e), \mathsf{ext}_{\mathsf{rhs}(\mathsf{lab}(e))}[i], \sigma)\}$
8:     return $N \cup \bigcup_{e \in E_{\mathsf{nt}}} N_e$
```

Algorithm 5 Algorithm to compute $\mathsf{internal}(u, x)$.

```
1: function internal($(u, x)$: $G$-representation)
2:     if $u = \varepsilon$ or $x$ is internal then
3:         return $x$
4:     else                                              ▷ $x$ is external, and $u \neq \varepsilon$
5:         Let $u = v.i$, where $i \in [\mathsf{rank}(\mathsf{dt}(G)[v])]$
6:         Let $e$ be the $i$-th edge in $\mathsf{sib}(E^{\mathsf{nt}}_{g_v})$
7:         Let $x$ be the $j$-th external node of $g_u$
8:         return $\mathsf{internal}((v, \mathsf{att}(e)[j]))$
```

Using the G-representation we can now give algorithms for the out-, $\mathsf{internal}$-, and node-mappings. Starting with out: given a nonterminal A, a node x of $\mathsf{rhs}(A)$ and a label σ we compute the set of $\mathsf{dt}(A)$-contexts containing the outgoing neighbours of x using Algorithm 4. This recursive top-down algorithm first computes the direct σ-neighbours of x within $\mathsf{rhs}(A)$ and then continues its search within the nonterminal edges incident to x. It terminates, because the algorithm only moves downwards in the derivation tree, never up. Its runtime is in $O(n\mathsf{height}(G))$, where n is the number of σ-neighbours x has in $\mathsf{val}(A)$. Note that, as explained in Section 4.1 above, some of these nodes may be external. To get the full neighbourhood, it is necessary to combine this result with the $\mathsf{internal}$-mapping, which can be computed using Algorithm 5. Given a node (u, x)[2] it uses the context u to find the internal node merged with x. The runtime is therefore in $O(\mathsf{height}(G))$. The final ingredient to give a full neighbourhood-algorithm is the node-mapping. As $\mathsf{node}(u, x) = \mathsf{first}(u) + x$ its runtime is identical to that of first already discussed above (i.e., $O(\mathsf{height}(G))$).

Finally the algorithms can be combined to compute the (σ-labelled) out-neighbourhood of a node x within $\mathsf{val}(G)$. The algorithm is stated as Algorithm 6. Assuming x has n such neighbours the total runtime is $O(\log_2 |G|_E \cdot \mathsf{height}(G) + \mathsf{height}(G)^2 n)$.
Remark. Note that we define a G-representation using a context of the derivation tree and subsequently use these contexts in the algorithms. As constructing the derivation tree explicitly is infeasible, an implementation of these algorithms may better be done with the following equivalent definition of G-representations. Let (u, x) be a G-representation with $u = u_1 \dots u_n$.

[2]Note that, because x may be external, this is not necessarily a G-representation

Algorithm 6 Function getOutNeighborhood, which given a G-representation p returns the set of IDs that are out-neighbors of the node represented by p within $\mathsf{val}(G)$.

```
1: function getOutNeighborhood($x$: node ID, $\sigma$: edge label)
2:     $(u, y) \leftarrow \mathsf{getGRep}(x)$
3:     $N \leftarrow \mathsf{out}(\mathsf{lhs}(\mathsf{dt}(G)[u]), y, \sigma)$
4:     return $\{\mathsf{node}(\mathsf{internal}((u.v, z))) \mid (v, z) \in N\}$
```

For u_i ($i \in [n-1]$) let A_i be the nonterminal $\mathsf{lhs}(\mathsf{dt}(G)[u_1 \dots u_i])$, and e_i the u_{i+1}-th edge in $\mathsf{sib}(E^{\mathsf{nt}}_{\mathsf{rhs}(A_i)})$. Finally, let $A_n = \mathsf{lhs}(\mathsf{dt}(G)[u])$. Then a G-representation can also be defined as $(A_1, e_1) \cdots (A_{n-1}, e_{n-1})(A_n, x)$. This definition explicitly lists the nonterminals and edges in the grammar that make the context u. This does not affect the runtimes as stated, or the logic of the algorithms.

4.3 Constant-Delay Traversal of Specific Graphs

As an improvement of the previous section, we present a data structure that allows for a traversal step in constant time for grammars of bounded rank. The caveat is that it only works for a restricted class of grammars: we call a graph *unique-label*, if for every node x, any label $\sigma \in \Sigma$ and any $i \in \mathbb{N}$ there is at most one edge e with $\mathsf{att}(e)[i] = x$ and $\mathsf{lab}(e) = \sigma$. The algorithm presented in this chapter only works for SL-HR grammars G where $\mathsf{val}(G)$ is unique-label. This has one important consequence: $\mathsf{out}(A, x, \sigma)$ always has at most one element. For the rest of this section we will therefore refer to this element directly by $\mathsf{out}(A, x, \sigma)$. Should $\mathsf{out}(A, x, \sigma) = \emptyset$ we say the mapping is undefined. This also makes it possible to precompute the out-mapping in one bottom-up run of the grammar. The following recursive definition can be applied to compute the mapping $\mathsf{out}(A, x, \sigma)$:

- $\mathsf{out}(A, x, \sigma) = (\varepsilon, y)$, if there is an edge $e \in E_{\mathsf{rhs}(A)}$ with $\mathsf{att}(e) = xy$, and $\mathsf{lab}(e) = \sigma$.
- $\mathsf{out}(A, x, \sigma) = (v.i, y)$ if $\mathsf{out}(\mathsf{lab}(e_i), \mathsf{ext}_{\mathsf{rhs}(\mathsf{lab}(e_i))}[j], \sigma) = (v, y)$ for a $j \in [1, \mathsf{rank}(e_i)]$.

This can be used in a bottom-up computation, because the definition always only depends on already computed results for nonterminals smaller than A according to $\leq_{\mathsf{NT}}$. Note however, that $\mathsf{out}(A, x, \sigma)$ needs to be computed for external nodes x as well for this to work. As an example, the grammar with the derivation tree in Figure 4.1 has the unique-label restriction. We furthermore assume that any grammar G in this section has the rank κ, which is considered to be fixed, and we let η_G be the maximal appearing node-ID within G (i.e., $\eta_G = \max\{\bigcup_{A \to g \in P} V_g\}$).
Remark. If we are only interested in out-neighbourhood queries the unique-label restriction only needs to apply to the first node of every att-relation, making the method work on a larger set of grammars.

4.3.1 Tableaux and their Precomputation

We seek a data structure that allows to compute, in constant time, each of the three operations out, $\mathsf{internal}$, and node, and which can be precomputed in polynomial time and space. To support $\mathsf{out}(A, x, \sigma)$ for every nonterminal A, every node x in $\mathsf{rhs}(A)$, and every $\sigma \in \Sigma$, we compute a "tableau" $\mathsf{tab}(A, x, \sigma)$ for each such triple. A tableau represents the $\mathsf{internal}$-mapping for every right-hand side used in the shortest derivation starting with A and producing the σ-labelled edge starting at x.

Definition 4.1. Let $G = (N, P, S)$ be an SL-HR grammar. For some $A \in N$, $x \in \{1, \dots, \eta_G\}$, and $\sigma \in \Sigma$ let m be the height of $\mathsf{dt}(A)$. The tableau $\mathsf{tab}(A, x, \sigma)$ consists of an $m \times \kappa$-matrix t, an $(m+1)$-tuple $t^{\mathsf{node}} \in V_{\mathsf{val}(G)}^{m+1}$, and an m-tuple $t^{\mathsf{nt}} \in N^m$. For $i \in [0, m-1]$ and $j \in [0, \kappa - 1]$ a cell $t(i, j)$ contains a tuple $(\mathsf{nxt}, \mathsf{node})$ where $\mathsf{nxt} \in ([0, 2^\beta - 1] \times [0, \mathsf{height}(G) - 1] \times [0, k-1])$ is a triple consisting of a pointer to a matrix, a row number, and a column number (recall from Section 2.6 that β is the bit length) and $\mathsf{node} \in [0, \eta_G]$.

For a tableau t with m rows we let $\#\mathsf{row}(t) = m$. Every cell of t uses 4 registers (the triple nxt and one number for node). We refer to the entirety of nxt as a pointer to a cell, because the triple identifies one cell of a matrix. For a cell $t(i, j) = (\mathsf{nxt}, \mathsf{node})$ we refer by $\mathsf{nxt}(t(i, j))$ to the dereferenced contents (i.e., the cell it references) and by $\mathsf{node}(t(i, j))$ to the node-ID in the cell. Let $\mathsf{nxt}(t(i, j)) = t'(i', j')$. We call nxt a *short pointer* if $t' = t$ and a *long pointer* otherwise. If nxt is a short pointer we call it *downwards* if $i' \geq i$ and *upwards* otherwise. For a tableau t let $\pi(t)$ be the address of the register containing (the start of) t. Thus, if nxt is a triple $(\pi(t), i, j)$ it references the cell $t(i, j)$.

We define the empty tableau t_ε for a $\mathsf{dt}(A)$-context ε as the $1 \times \kappa$-tableau that has $t_\varepsilon(0, j) = ((\pi(t_\varepsilon), 0, j), 0)$ for $j \in [0, \mathsf{rank}(A) - 1]$ (that is, $\mathsf{nxt}(t_\varepsilon(0, j))$ points to itself), $t_\varepsilon(0, j) = (0, 0)$ for $j \geq \mathsf{rank}(A)$, $t_\varepsilon^{\mathsf{node}}(0) = t_\varepsilon^{\mathsf{node}}(1) = 0$, and $t_\varepsilon^{\mathsf{nt}}(0) = A$. For any $A \in N$, $x \in V_{\mathsf{rhs}(A)}$, and $\sigma \in \Sigma$ where $\mathsf{out}(A, x, \sigma) = (u, y)$ we compute the tableau $\mathsf{tab}(A, x, \sigma) = \mathsf{createTableau}(A, u)$ using Algorithm 7. Let $t = \mathsf{tab}(A, x, \sigma)$. Note that $\#\mathsf{row}(t) = |u| + 1$: the intention is that every row of t represents an ancestor of u. Let v be such an ancestor and let $g_{\mathsf{dt}(A), v}$ have n external nodes. Then, for every $j \in [0, n-1]$ we want that the nxt-pointer of $t(|v|, j)$ references a cell c such that this cell contains the node y with $\mathsf{internal}_{\mathsf{dt}(A)}(v, \mathsf{ext}_g[0, j]) = (w, y)$. This cell should also be in the $|w|$-th row of t. If this property holds for every ancestor of u, then t represents the $\mathsf{internal}$-mapping for the whole $\mathsf{dt}(A)$-context u. Furthermore, $t^{\mathsf{node}}(|v|)$ should be $\mathsf{first}_{\mathsf{dt}(A)}(v)$ so that we can compute the number of any internal node within $\mathsf{val}(A)$.

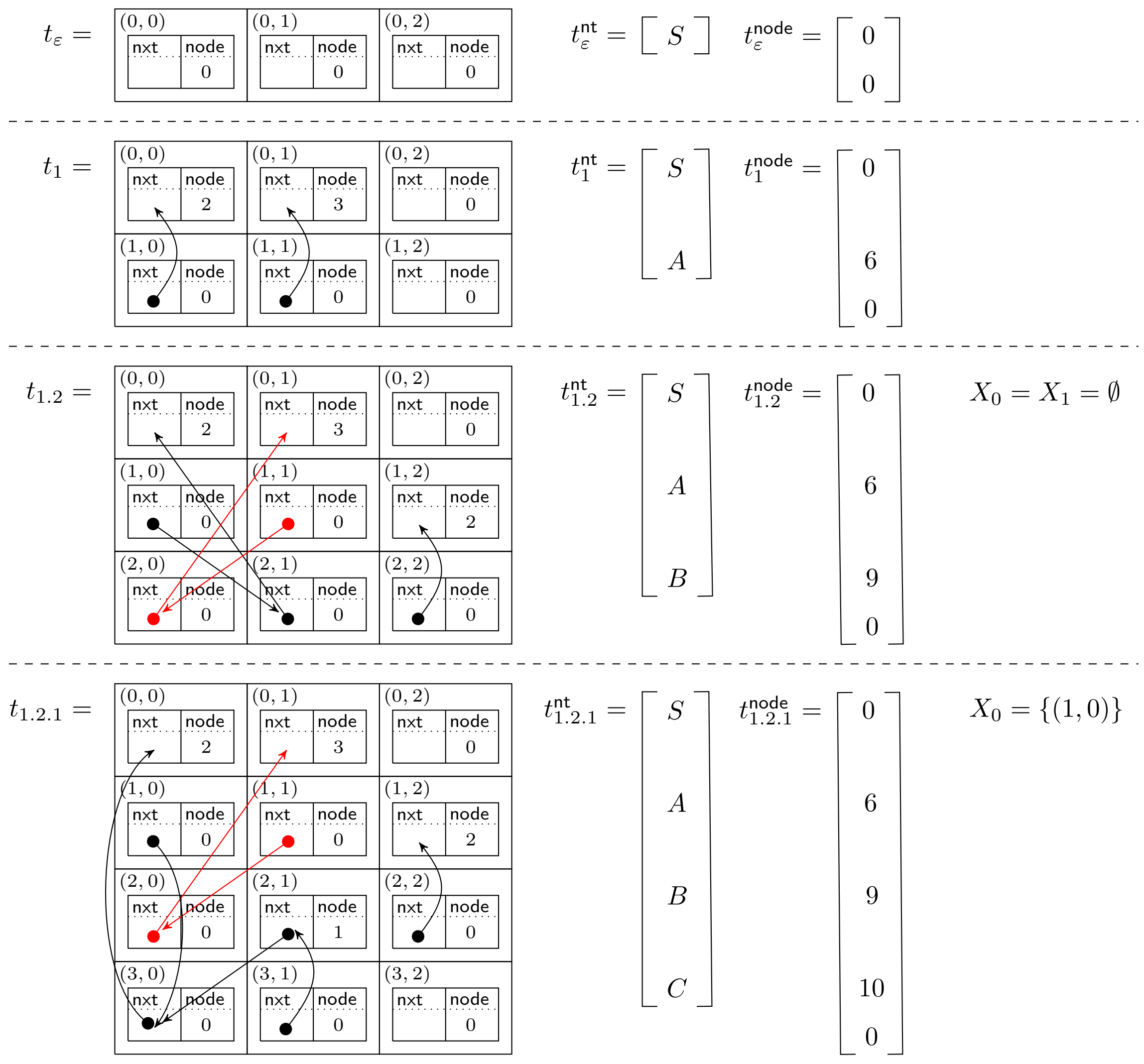
An example of this computation is shown in Figure 4.2. The Figure shows the results of every recursive call of $\mathsf{createTableau}$ from top to bottom, with $\mathsf{tab}(S, 2, a) = t_{1.2.1}$. We invite the reader to verify the figure by applying the algorithm on the grammar in Figure 4.1. Note that the construction creates downwards pointers which reference a cell containing an upwards pointer (e.g. cells $(1, 0)$ or $(2, 1)$). This seems unnecessary: these cells could instead directly reference the cell $(0, 0)$ containing the correct internal node at this position. However, there may be many cells that all reference the same internal node (in this small example there are already 3 for the node in cell $(0, 0)$). In the next section we will proceed to attach tableaux together, but we

Algorithm 7 Algorithm to create a tableau that s-models an s-context

```
1: function createTableau(Nonterminal $A$, $\mathsf{dt}(A)$-context $u$)
2:     if $u = \varepsilon$ then
3:         return $t_\varepsilon$
4:     else                                                    ▷ $u = v.l$ for some $l \in \mathbb{N}$
5:         $t_v$ =createTableau$(A, v)$
6:         Initialize $t_u$ as $|u| \times \kappa$-tableau
7:         for all $i \in [0, |v|], j \in [\kappa - 1]$ do
8:             $t_u(i,j) \leftarrow t_v(i,j)$
9:             $t_u^{\mathsf{nt}}(i) \leftarrow t_v^{\mathsf{nt}}(i)$
10:            $t_u^{\mathsf{node}}(i) \leftarrow t_v^{\mathsf{node}}(i)$
11:        Let $B = t_v^{\mathsf{nt}}(|v|)$ and $\mathsf{sib}(E^{\mathsf{nt}}_{\mathsf{rhs}(B)}) = (e_1, \dots, e_n)$
12:        for $j \in [0, \mathsf{rank}(e_l) - 1]$ do               ▷ $-1$ because column indices start with 0
13:            $x \leftarrow \mathsf{att}(e_l)[j+1]$
14:            if $x$ is internal then
15:                $\mathsf{nxt}(t_u(|u|, j)) \leftarrow (\pi(t_u), |v|, j)$
16:                $\mathsf{node}(t_u(|v|, j)) \leftarrow x$
17:                $\mathsf{node}(t_u(|u|, j)) \leftarrow 0$
18:            else                                   ▷ $x$ is the $j'$-th external Node of $\mathsf{rhs}(B)$
19:                $X_j \leftarrow \{(a,b) \mid \mathsf{nxt}(t_v(a,b)) = t_v(|v|, j'-1)\}$
20:                $\mathsf{nxt}(t_u(|u|, j)) \leftarrow \mathsf{nxt}(t_v(|v|, j'-1))$
21:                $\mathsf{nxt}(t_u(|v|, j'-1)) \leftarrow (\pi(t_u), |u|, j)$
22:                $\mathsf{node}(t_u(|u|, j)) \leftarrow 0$
23:                For all $(a,b) \in X_j$ : $\mathsf{nxt}(t_u(a,b)) \leftarrow (\pi(t_u), |u|, j)$
24:        $t_u^{\mathsf{nt}}(|u|) \leftarrow \mathsf{lab}(e_l)$
25:        $t_u^{\mathsf{node}}(|u|) \leftarrow t_v^{\mathsf{node}}(|v|) + \sum_{j<l} \mathsf{nodes}(\mathsf{lab}(e_j)) + |V_{g_{\mathsf{dt}(A),v}}| - |\mathsf{ext}_{g_{\mathsf{dt}(A),v}}|$
26:        return $t_u$
```

need to do so by only changing *one pointer* per column, to obtain $O(\kappa)$ time. This technique of concatenating pointers allows us to do so, because we only need to change the upwards pointer to have every relevant cell reference a new node. — For an example of the internal-mapping consider that $\mathsf{out}(S, 2, a) = (u_4, 2)$ in Figure 4.1. Every row of $\mathsf{tab}(S, 2, a)$ as shown in Figure 4.2 corresponds to an ancestor of u_4 in the derivation tree. As a concrete example, the third row represents u_3. Here we have $\mathsf{internal}(u_3, 4) = (u_1, 2)$, and if we follow the arrow in cell $(2, 1)$ of $\mathsf{tab}(S, 2, a)$ it first leads to cell $(3, 0)$ and then to cell $(0, 0)$, where the node-ID given is 2, and the corresponding nonterminal in $\mathsf{tab}(S, 2, a)^{\mathsf{nt}}(0) = S$.

We want to show that our construction is "correct" with respect to the intuition given above. We first define an auxiliary method $\mathsf{findNode}(t(i, j))$ for a cell (i, j) in the tableau t. This operation dereferences the nxt-pointer once or twice, depending on whether it first encounters a downwards pointer or not. The resulting cell is defined below to contain the internal node that is merged



$t_\varepsilon =$
$t^{\mathsf{nt}}_\varepsilon = [\, S \,]$
$t^{\mathsf{node}}_\varepsilon = [\, 0 \; 0 \,]$
$t_1 =$
$t^{\mathsf{nt}}_1 = [\, S \; A \,]$
$t^{\mathsf{node}}_1 = [\, 0 \; 6 \; 0 \,]$
$t_{1.2} =$
$t^{\mathsf{nt}}_{1.2} = [\, S \; A \; B \,]$
$t^{\mathsf{node}}_{1.2} = [\, 0 \; 6 \; 9 \; 0 \,]$
$X_0 = X_1 = \emptyset$
$t_{1.2.1} =$
$t^{\mathsf{nt}}_{1.2.1} = [\, S \; A \; B \; C \,]$
$t^{\mathsf{node}}_{1.2.1} = [\, 0 \; 6 \; 9 \; 10 \; 0 \,]$
$X_0 = \{(1,0)\}$
nxt
node

Figure 4.2: Every step of the computation of $\mathsf{tab}(S, 2, a)$ for the derivation tree in Figure 4.1. Some arrows are coloured to make it easier to read where the pointers are leading.

with the j-th external node. Note that the resulting cell may be in a different tableau. We distinguish two cases for the pointer $\mathsf{nxt}(t(i, j))$:

1. If $\mathsf{nxt}(t(i, j))$ is upwards or long, then $\mathsf{findNode}(t(i, j))) = \mathsf{nxt}(t(i, j)) = t'(i', j')$.

2. If $\mathsf{nxt}(t(i, j))$ is downwards, then $\mathsf{findNode}(t(i, j)) = \mathsf{nxt}(\mathsf{nxt}((t(i, j))) = t'(i', j')$.

Note, in both cases $t' \neq t$ if and only if one of the pointers was a long pointer. A tableau $\mathsf{tab}(A, x, \sigma)$ never includes long pointers, but they will be important in Section 4.3.2 where we attach tableaux. We now define formally how a tableau represents a $\mathsf{dt}(A)$-context.
Definition 4.2. Let $A \in N$, $s = \mathsf{dt}(A)$, and let u be an s-context. For any ancestor v of u we let $A_v = \mathsf{lhs}(s[v])$. A tableau t *s-models* u if for every ancestor v of u

1. $t^{\mathsf{nt}}(|v|) = A_v$

Algorithm 8 Algorithm to attach t_2 to the i-th row of t_1.

```
Input: $t_1^{\mathsf{nt}}(i) = B$ and $t_2$ models $\mathsf{dt}(B)$-context
1: function attach(Tableaux $t_2$, $t_1$, number $i$)
2:     for $j \in [0, \mathsf{rank}(B) - 1]$ do
3:         if $\mathsf{nxt}(t_2(1, j))$ is a downwards pointer then
4:             $c \leftarrow \mathsf{nxt}(t_2(1, j))$
5:         else                    ▷ $\mathsf{nxt}(t_2(1, j))$ is a long pointer from a previous attachment
6:             $c \leftarrow t_2(1, j)$
7:         Let $t(i', j') = \mathsf{findNode}(t_1(i, j))$
8:         $\mathsf{nxt}(c) \leftarrow (\pi(t), i', j')$
9:     $t_2^{\mathsf{node}}(\#\mathsf{row}(t_2) + 1) \leftarrow t_1^{\mathsf{node}}(\#\mathsf{row}(t_1) + 1) + t_1^{\mathsf{node}}(i)$
```

2. $t^{\mathsf{node}}(|v|) = \mathsf{first}_s(v)$
3. For $j \in [0, \mathsf{rank}(A_v) - 1]$ let $c = \mathsf{findNode}(t(|v|, j))$ and let $y = \mathsf{ext}_{g_{s,v}}[j]$. If $\mathsf{internal}(v, y) = (w, z)$ then $\mathsf{node}(c) = z$ and c is a cell of the $|w|$-th row of t. If $\mathsf{internal}(v, y)$ is undefined then $\mathsf{node}(c) = 0$ and c is a cell in the first row of t.

Lemma 4.3. *For $A \in N$, $x \in V_{\mathsf{rhs}(A)}$, and $\sigma \in \Sigma$ let $\mathsf{out}(A, x, \sigma) = (u, y)$. The tableau $\mathsf{tab}(A, x, \sigma)$ then $\mathsf{dt}(A)$-models u.*

Proof. We proof the claim inductively over u. For $u = \varepsilon$ the tableau t_ε obviously $\mathsf{dt}(A)$-models u. Let $u = v.l$ for some $l \in \mathsf{rank}(\mathsf{lhs}(\mathsf{dt}(A)[v]))$. By induction we obtain t_v, which $\mathsf{dt}(A)$-models v. Let t_u be computed using Algorithm 7. Condition 1 of Definition 4.2 obviously holds. Condition 2 is also fulfilled, as Line 25 of Algorithm 7 is just the definition of $\mathsf{first}(v)$. This leaves condition 3. Let $j \in [0, \mathsf{rank}(\mathsf{lhs}(\mathsf{dt}(A)[u]))]$ and let x be defined as in Line 13 of Algorithm 7. If x is internal, then $\mathsf{findNode}(t_u(|v|, j))$ fulfils condition 3 by induction and $\mathsf{findNode}(t_u(|u|, j))$ correctly references the cell containing x by Lines 15 to 17 of the Algorithm. If otherwise x is external then let $\mathsf{internal}_{\mathsf{dt}(A)}(v, x) = (w, y)$ and let x be the j'-th external node of $\mathsf{rhs}(\mathsf{dt}(A)[v])$. By induction $\mathsf{findNode}(t_v(|v|, j' - 1))$ references a cell c in the $|w|$-th row of t_v, which contains y. Note that it has to be an upwards pointer, because there is no row below $|v|$ in t_v. After Line 23 of the algorithm, $\mathsf{nxt}(t_u(|u|, j)) = c$ is an upwards pointer, and every downwards pointer previously pointing to $t_u(|v|, j)$ now instead points to $t_u(|u|, j)$ and thus has the same $\mathsf{findNode}$-result as before (i.e., c). The same argument holds if $\mathsf{internal}_{\mathsf{dt}(A)}(v, x)$ is undefined, where $t_v(|v|, j')$ is some cell in the first row of t_v. □

For $\mathsf{out}(A, x, \sigma) = (u, y)$ and an ancestor v of u we also say that the first $|v|$ rows of $\mathsf{tab}(A, x, \sigma)$ $\mathsf{dt}(A)$-model v.

4.3.2 Attaching Tableaux

The tableaux computed in the previous section do not represent every possible context u in $\mathsf{dt}(S)$, as there can be $O(2^{|G|})$ many contexts. They do, however, provide the information necessary to traverse along the edges. Consider again the example at the end of the introduction to Section 4.1

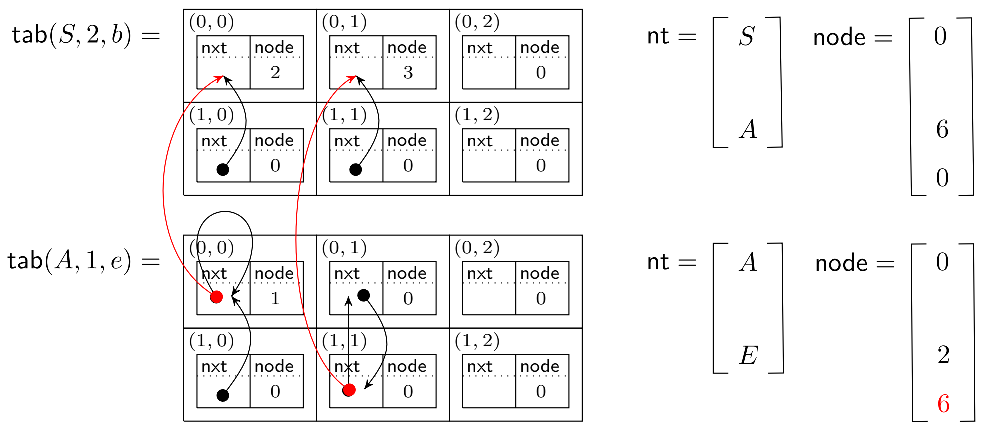


Figure 4.3: Example for the attachment of tableaux. The figure shows the result of the operation $\mathsf{attach}(\mathsf{tab}(A, 1, e), \mathsf{tab}(S, 2, b), 1)$ for the derivation tree shown in Figure 4.1.

and Figure 4.1: if we start in node 2 of S and follow the b-edge ($\mathsf{out}(S, 2, b) = (u_2, 1)$) we move to node 1 in $\mathsf{rhs}(A)$. From there let us follow the e-edge: $\mathsf{out}(A, 1, e) = (w, 3)$ where $w = 1$ and $v = u_2.w$. Thus the $\mathsf{dt}(A)$-context w and the $\mathsf{dt}(S)$-context u_2 combine into a new $\mathsf{dt}(S)$-context v. Our tableaux provide the same functionality. Given $t_1 = \mathsf{tab}(S, 2, b)$ and $t_2 = \mathsf{tab}(A, 1, e)$ we can combine them using an operation $\mathsf{attach}(t_2, t_1, i)$, where i is a row number of t_1 marking the point where we wish to attach t_2 at. This section details the attach operation. Let t_1 be a tableau $\mathsf{tab}(A, x, \sigma)$ and i a row number, such that the first i rows of t model a $\mathsf{dt}(A)$-context u with $\mathsf{lhs}(\mathsf{dt}(A)[u]) = B$. For some $y \in V_{\mathsf{rhs}(B)}$ and $\sigma' \in \Sigma$ let $t_2 = \mathsf{tab}(B, y, \sigma')$ be a tableau modelling a $\mathsf{dt}(B)$-context v. We combine them by linking from t_2 to t_1 using one long pointer per column. Therefore, $\mathsf{attach}(t_2, t_1, i)$ is defined as given in Algorithm 8. The main operation is in Line 8: here we create a long pointer from t_2 to the cell of t_1 containing the internal node merged with the j-th external node.

An example of this operation is shown in Figure 4.3. Here, the tableau $\mathsf{tab}(A, 1, e)$ is attached to the last row of the tableau $\mathsf{tab}(S, 2, b)$ (both tableaux are based on the derivation tree in Figure 4.1). Changes are entered in red: The two original nxt-pointers are *replaced* by the red long pointers, and the offset in $\mathsf{tab}(A, 1, e)^{\mathsf{node}}$ is adjusted.

Again, we want to show that this leads to a correct representation of the combined context. We define "correct representation" using the following definition, which extends the arguments of Definition 4.2 to include the possibility of long pointers and the usage of the offset element in $t^{\mathsf{node}}(\#\mathsf{row}(t) + 1)$.

Definition 4.4. Let $u = u_1 \ldots u_n$ be a $\mathsf{dt}(S)$-context. Let $A_1 = S$ and for every $i \in \{2, \ldots, n\}$ let $A_i = \mathsf{lhs}(\mathsf{dt}(S)[u_{i-1}])$. Let $t_1, \ldots, t_n$ be tableaux such that for $i \in \{1, \ldots, n\}$ the first $|u_i|$ rows of t_i $\mathsf{dt}(A_i)$-model u_i. Then we say the sequence of tableaux $t_1, \ldots, t_n$ *models* the context u if for every ancestor $v = u_1 \ldots u_l[1..i]$ (with $l \in [n]$ and $i \in [|u_l|]$) of u

1. $t_l^{\mathsf{nt}}(|v|) = \mathsf{lhs}(\mathsf{dt}(S)[v])$
2. $t_l^{\mathsf{node}}(|v|) + t_l^{\mathsf{node}}(\#\mathsf{row}(t_l) + 1) = \mathsf{first}(v)$

3. For $j \in [0, \mathsf{rank}(g_v)]$ let $c = \mathsf{findNode}(t_l(i,j))$ and let $y = \mathsf{ext}_{g_v}[j]$. Let $\mathsf{internal}(v,y) = (w,z)$ where $w = u_1 \dots u_{l'}[1..i']$ for some $l' \leq l$ and $i' \in [0, |u_{l'}|]$. Then $\mathsf{node}(c) = z$ and c is a cell in the i'-th row of $t_{l'}$.

Note that, as we are now only considering $\mathsf{dt}(S)$-contexts, $\mathsf{internal}$ is always defined.

Lemma 4.5. *Let $t_1, \dots, t_n$ be a sequence of tableaux modelling a context $u = u_1 \dots u_n$, and for $A = \mathsf{lhs}(\mathsf{dt}(S)[u])$, $x \in \mathsf{rhs}(A)$, and $\sigma \in \Sigma$ let $t = \mathsf{tab}(A, x, \sigma)$ be the tableau that $\mathsf{dt}(A)$-models a $\mathsf{dt}(A)$-context v. Then, after running $\mathsf{attach}(t_n, t, |u_n|)$ the sequence $t_1, \dots, t_n, t$ models $u.v$. Computing $\mathsf{attach}(t_n, t, |u_n|)$ takes $O(\kappa)$ time.*

Proof. Condition 1 of Definition 4.4 is trivially fulfilled. For Condition 2 consider first that t_n in row $|u_n|$ and t in the first row actually represent the same context: for any derivation tree s, the first row of a tableau s-modelling an s-context v models the s-context ε. Also, for any tableau t by definition $t^{\mathsf{node}}(1) = 0$ (because $\mathsf{first}(\varepsilon) = 0$). As $t_1, \dots, t_n$ model u, by Definition 4.4 $t_n^{\mathsf{node}}(|u_n|) + t_n^{\mathsf{node}}(\#\mathsf{row}(t_n) + 1) = \mathsf{first}(u) = t^{\mathsf{node}}(\#\mathsf{row}(t) + 1)$ and as $\mathsf{first}(u) = \mathsf{first}(u.\varepsilon)$ we also have $t^{\mathsf{node}}(1) + t^{\mathsf{node}}(\#\mathsf{row}(t) + 1) = \mathsf{first}(u.\varepsilon)$.

This leaves condition 3. Let $j \in [0, \kappa - 1]$. As $t_1, \dots, t_n$ models u, $t_n(|u_n|, j)$ satisfies condition 3. Let $r = \mathsf{findNode}(t_n(|u_n|, j))$. Now $t(1,j)$ needs to have the same property, which is true: if $\mathsf{nxt}(t(1,j))$ was a long pointer originally it now is a long pointer to c. If $\mathsf{nxt}(t(1,j))$ is a downwards pointer, then $\mathsf{nxt}(\mathsf{nxt}(t(1,j)))$ is now a long pointer to c. In both cases $\mathsf{findNode}(t(1,j)) = \mathsf{findNode}(t_n(|u_n|, j))$.

Finally, regarding the runtime, the attach-operation uses one call of $\mathsf{findNode}$ and follows up to one nxt-pointer per column of the tableau. As every tableau has κ columns, we obtain a runtime of $O(\kappa)$. □

4.3.3 Traversal Algorithm

To traverse a graph using the method outlined at the beginning of Section 4.1 we need to store the current context $u = u_1 \dots u_n$ within $\mathsf{dt}(S)$. As detailed in the previous section we represent this context using a sequence of tableaux $t_1, \dots, t_n$. However as these tableaux are all chained together using long pointers, we only need to store a pointer to $t_n = \mathsf{tab}(A, x, \sigma)$ for some $A \in N$, $x \in \mathsf{rhs}(A)$ and $\sigma \in \Sigma$. We call this pointer curTab. It is possible that u_n is actually modelled by the first i rows of t_n and not the entire tableau. We also need to store this number i of the current row of t_n, which we call pos. Finally, our current position is a node $y \in g_u$, which we call cur.

To find the matching internal node of a node $x \in g_u$ let $\mathsf{internal}(x, \mathsf{curTab}, \mathsf{pos}) = (x, \mathsf{curTab}, \mathsf{pos})$ if x is internal. Otherwise, let x be the j-th external node of g_u and let $c = \mathsf{findNode}(\mathsf{curTab}(\mathsf{pos}, j - 1))$ be a cell in the j'-th row of $\mathsf{tab}(A, y, \sigma)$ for some $A \in N$, $y \in \mathsf{rhs}(A)$, and $\sigma \in \Sigma$. Then $\mathsf{internal}(x, \mathsf{curTab}, \mathsf{pos}) = (\mathsf{node}(c), \mathsf{tab}(A, y, \sigma), j')$.

Theorem 4.6. *Given a unique-label SL-HR grammar $G = (N, P, S)$ of bounded rank κ and height h, we can traverse the edges of the graph $\mathsf{val}(G)$ starting from any node in the start* *graph, with constant time per traversal step. The precomputation uses $O(|G||\Sigma|\kappa h^2)$ time and $O(|G||\Sigma|\kappa h)$ space on a word-RAM with bit length $\beta \in O(\log_2 |G|)$.*

Proof. For any $A \in N$, $x \in [0, \eta_G]$, and $\sigma \in \Sigma$ we precompute the tableaux $\mathsf{tab}(A, x, \sigma)$ and for $\mathsf{out}(A, x, \sigma) = (v, y)$ we define (and store) $\mathsf{succ}(A, x, \sigma) = y$. This mapping can be computed simultaneously with the precomputation of the tableaux. There are $|G| \cdot |\Sigma|$ many such tableaux to compute and computing one takes $O(h^2\kappa)$ time. Therefore the precomputation takes $O(|G||\Sigma|\kappa h^2)$ time, but only $O(|G||\Sigma|\kappa h)$ space, because the intermediate tableaux constructed in Algorithm 7 are not stored, only the final ones. We then initialize the global state with $\mathsf{curTab} = t_\varepsilon$, $\mathsf{pos} = 1$, and $\mathsf{cur} = x$ for any $x \in V_S$. Now, assume we want to traverse to the σ'-successor of the current node, then we do the following:

1. Let $A = \mathsf{curTab}^{\mathsf{nt}}(\mathsf{pos})$ and $x = \mathsf{cur}$
2. If $\#\mathsf{row}(\mathsf{tab}(A, x, \sigma')) = 1$ let $\mathsf{cur} = \mathsf{succ}(A, x, \sigma')$
3. Otherwise $\mathsf{attach}(\mathsf{curTab}, \mathsf{tab}(A, x, \sigma'))$
4. $\mathsf{curTab} = \mathsf{tab}(A, x, \sigma')$, $\mathsf{pos} = |\mathsf{tab}(A, x, \sigma')|$, $\mathsf{cur} = \mathsf{succ}(A, x, \sigma')$
5. Let $\mathsf{internal}(\mathsf{cur}, \mathsf{curTab}, \mathsf{pos}) = (y, t, j)$ and let $\mathsf{cur} = y$, $\mathsf{curTab} = t$, and $\mathsf{pos} = j$.

All these steps take $O(1)$ time, with the exception of the attach-operation, which takes $O(\kappa)$ time. At any point we can determine our position in $\mathsf{val}(G)$ using $\mathsf{node}(\mathsf{cur}) = \mathsf{curTab}^{\mathsf{node}}(\mathsf{pos}) + \mathsf{curTab}^{\mathsf{node}}(\#\mathsf{row}(\mathsf{curTab}) + 1) + \mathsf{cur}$. $\square$

Regarding the space for the precomputation, specifically $O(|N| \cdot \eta_G \cdot |\Sigma| \cdot h \cdot \kappa)$ registers are needed with $\beta \in O(\log_2 |G|)$ for the tableaux. Furthermore, the current state during the traversal (curTab, pos, and cur) uses three registers.

For Theorem 4.6 it is assumed that the traversal starts from a node in the start graph. This can be expanded to any node of $\mathsf{val}(G)$ by using Algorithm 3. Given some node ID x of $\mathsf{val}(G)$, compute first its G-representation (u, y) using Algorithm 3. Then, initialize $\mathsf{curTab} = \mathsf{createTableau}(S, u)$ (Algorithm 7), $\mathsf{pos} = |u|$, and $\mathsf{cur} = y$. This adds an initial computation using $O(h \log_2 |G|_E + h^2\kappa)$ time.

It is straight-forward to apply this result also to SLT grammars of bounded rank κ. Let r be the largest *terminal* rank appearing in the grammar. The trees can now be viewed as directed graphs with the edges labelled by their Dewey-address, i.e., $\mathsf{out}(x, A, i)$ would lead to the i-th child of node x within $\mathsf{rhs}(A)$ and $\mathsf{internal}$ matches the parameter nodes with the root of the trees they represent. Thus we get for any SLT grammar H:

Corollary 4.7. *Assuming κ is constant, given a κ-bounded SLT grammar H over Σ of height h and with a maximal terminal rank r we can traverse the tree $\mathsf{val}(H)$ starting from the root-node with constant time per traversal step. The precomputation uses $O(|H|r\kappa h^2)$ time and $O(|H|r\kappa h)$ space.*

The previously known result on constant-delay traversal of SLT grammars [LMR16] only works for SLT grammars of rank 1. Any rank-κ SLT grammar (or κ-bounded SLT grammar) can be

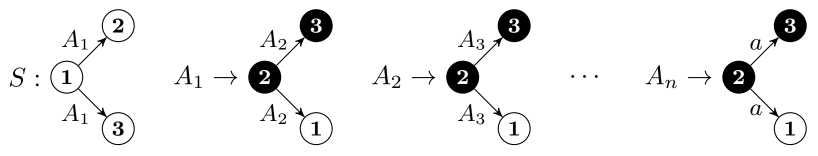


Figure 4.4: Grammar generating a Star with exponentially many tableaux.

converted into one of rank 1 [LMS12], but this takes $O(r\kappa|H|)$ time and $O(r|H|)$ space. Thus, the method presented here induces additional space overhead proportional to $\mathsf{height}(H)$, but the constant for a traversal step is potentially smaller as it only uses arrays and pointer-operations in contrast to the more complex data structures of [LMR16].

4.3.4 Limitations of the Method

The unique-label restriction is a strong one: it includes all tree-generating grammars and some other graphs, but of all the labelled graphs mentioned in Section 3.6, none have the property. Naturally this raises the question, whether it is necessary. Unfortunately, at least for the method as presented here, this seems to be the case. Consider the grammar given in Figure 4.4. It generates a star-graph with one centre node and 2^n a-neighbours. For the method given here to work, it would now be necessary to represent every context with only $O(n)$ precomputed tableaux. However, as evidenced by the derivation tree given in Figure 4.5, every leaf is modelled by a different tableaux. Furthermore, every leaf contains a node that is a neighbour of the 1-node in the start graph. Thus all of these tableaux need to be precomputed, but there are $O(2^n)$ many such tableaux. They do share some similarities, making it possible to generate all of them from “partial tableaux”, but there is no clear way to generalize this idea to arbitrary grammars.

We do note, however, that the grammar is a rather unnatural way of generating such a star-pattern. The more natural grammar in Figure 4.6 on the other hand has the property that every relevant context of the derivation tree indeed uses almost the same tableau. Only the node-vector would need to be adapted at runtime to the concrete context, which could likely be done using precomputed offsets. It is possible that there exists some normal form for SL-HR grammars, where our method would always work. Thus far we have not found one.

4.4 Related Work

The compression methods given in Sections 3.7.2 and 3.7.3 all support adjacency and neighbourhood queries. We already mention there, whether this only works for out-neighbourhood queries, or covers in-neighbourhood as well. While the methods of this section have not been implemented, it is likely that the access times will be slower than the compared methods, due to the additional overhead introduced by the grammar representation.

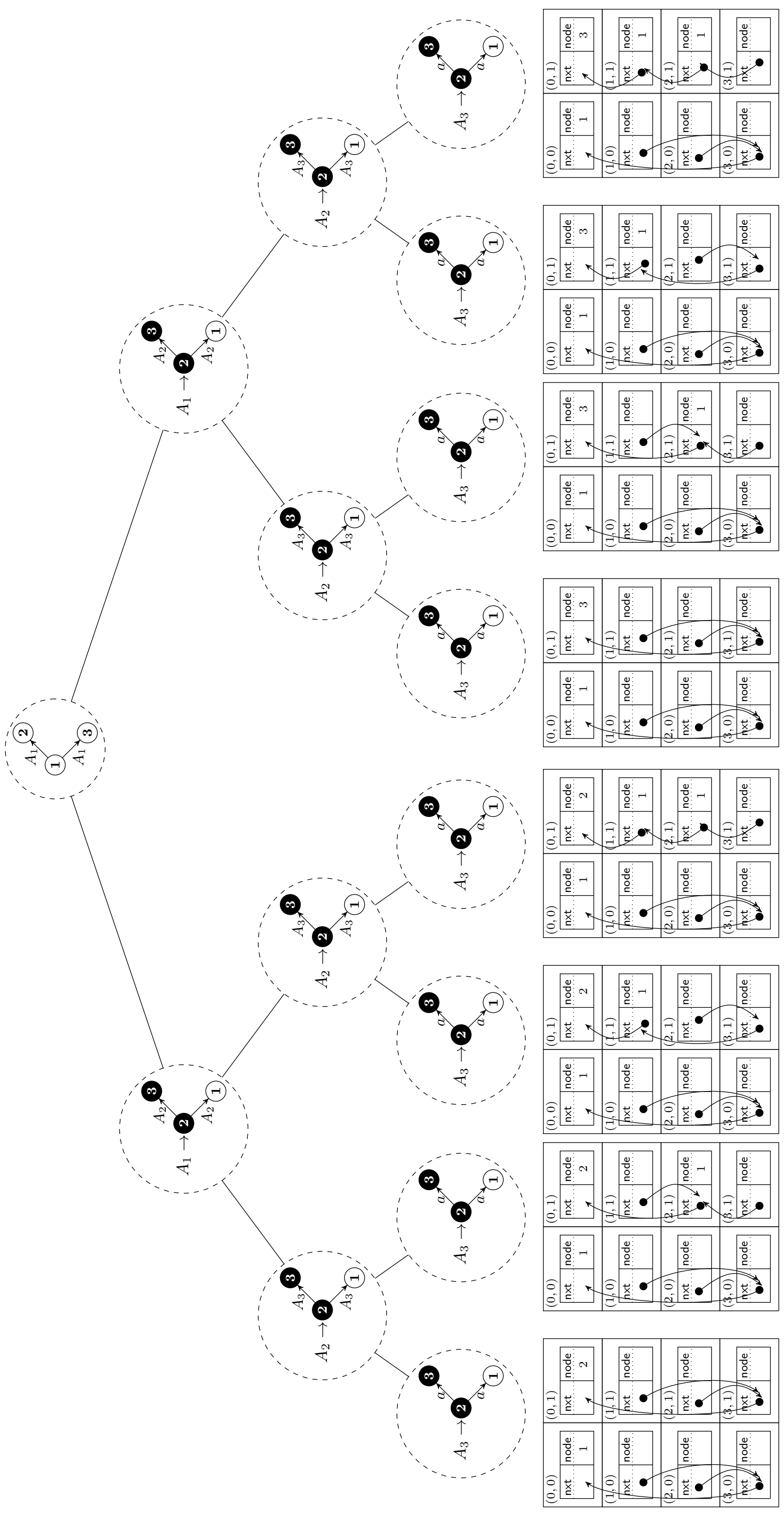


Figure 4.5: Derivation tree for the grammar in Figure 4.4 with $n = 3$. Every leaf is annotated with the tableau that dt(S)-models the leaf.

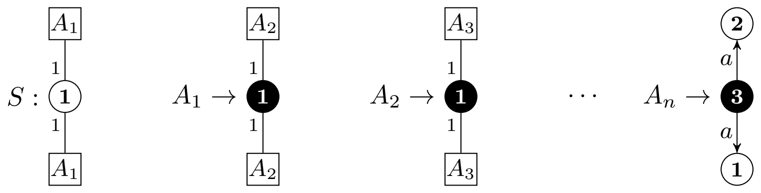


Figure 4.6: Alternative grammar equivalent to the grammar in Figure 4.4, but with polynomially many tableaux.

To the best of our knowledge, there are no other traversal methods for graph grammars. Constant-delay traversal of string and tree grammars is discussed in [GKPS05] and [LMR16], respectively. The latter method builds on the former. Both represent the position in the derivation tree using a stack, which contains the nodes where the direction was changed. They use an additional data structure to find the parent of such a node in constant time, making it in turn possible to update the stack in constant time to implement the traversal step. Both methods require the input grammar to be in a normal form. For strings, every rule needs to have exactly size 2, for trees the normal form using only one parameter from [LMS12] is required.

4.5 Discussion and Further Research

In this chapter the question of access to the compressed structure is discussed. We present algorithms to find a compressed representation of a node within the represented graph (a "G-representation"), and to traverse the graph from there. Both of these methods have disadvantages: while the first one works on arbitrary grammars, its runtime per traversal step depends on the height of the grammar and the size of the right-hand sides. The second one can do a traversal step in constant time, but requires preprocessing using quadratic time and space (with respect to the size of the grammar). More importantly, it only works on a restricted set of grammars and we also give indication that this restriction is likely inherent to the method.

This leaves two clear routes for future work. First, the given algorithm for constant-delay traversal should be experimentally evaluated. Even doing so only for tree grammars would already be interesting, as a previous implementation of the ideas in [LMR16] were slower than the naïve (and space hungry) traversal presented in [MS10]. While our method uses more space than the one given in [LMR16], the actual traversal is realized using rather simple pointer structures, thus we would hope for competitive speed. A second direction is to find a constant-delay traversal method that works for larger classes of SL-HR grammars. This method would likely need to use a new approach as both the one presented here, as well as the approach for trees in [LMR16] do not easily generalize to (arbitrary) SL-HR grammars.

Chapter 5

Speed-Up Algorithms

One attractive feature of straight-line context-free grammars is the ability to execute finite automata over them without prior decompression. This was first proved for strings (see e.g. [BMPT97, MS04, PR99]) and later extended to trees (and various models of tree automata, see [LMS12]). The idea is to run the automaton in one pass, bottom-up, through the grammar. As an example, consider the grammar $S \to AAA$ and $A \to ab$, and a deterministic automaton $\mathcal{A}$ that accepts strings (over $\{a, b\}$) with an odd number of a's. Thus, $\mathcal{A}$ has states q_0, q_1 (where q_0 is initial and q_1 is final) and the transitions (q_0, a, q_1), (q_1, a, q_0), and (q, b, q) for $q \in \{q_0, q_1\}$. Since the actual active states are not known during the bottom-up run through the grammar, we need to run the automaton *in every possible state* over a rule. For the nonterminal A we obtain (q_0, A, q_1) and (q_1, A, q_0), i.e., running in state q_0 over the string produced by A brings us to state q_1, and starting in q_1 brings us to q_0. Since S is the start nonterminal, we are only interested in starting the automaton in its initial state q_0. We obtain the run $(q_0, A, q_1)(q_1, A, q_0)(q_0, A, q_1)$, i.e., the automaton arrives in its final state q_1 and hence the grammar represents a string with odd number of a's. It should be clear that the running time of this process is $O(|Q||G|)$, where Q is the set of states of the automaton, and G is the grammar. As $|G|$ is smaller than $|\mathsf{val}(G)|$ this presents a speed-up proportional to the compression ratio. This is a major advantage over many other compressed input representations (e.g. boolean circuits), which induce an exponential blow-up in complexity (see also Section 5.4 below).

Unfortunately, for graphs there does not exist an accepted notion of finite-state automaton. Nevertheless, properties that can be checked in one pass through the derivation tree of a graph grammar have been studied under various names: “compatible”, “finite”, and “inductive”, and it was later shown that these notions are essentially equivalent [HKL93]. Courcelle and Mosbah [CM93] show that all properties definable in “counting monadic second-order logic” (CMSO) belong to this class, and by their Proposition 3.1, the complexity of evaluating a CMSO property ψ over a derivation tree t can be done in $O(|t|\eta)$, where η is an upper bound on the complexity of evaluation on each right-hand side of the rules in t. This is done by a bottom-up computation on the nodes of t, where every node u can use the results of its children to achieve a correct computation for the subtree rooted at u.

Proposition 5.1. *Let ψ be a fixed CMSO property. For a given SL-HR grammar G it can be decided in $O(|G|\eta)$ time whether or not ψ holds on $\mathsf{val}(G)$, where η is an upper bound on the time needed for the computation per right-hand side of the grammar.*

Note that we state the time complexity based on $|G|$ instead of $|t|$. This needs an adjustment of the proof, as the derivation tree t cannot be explicitly constructed, because it may be exponentially larger than G. However, a derivation DAG (directed acyclic graph) still retains all information necessary for the algorithm stated in [CM93] to work. It has a hierarchy, that makes the bottom-up computation possible, and while there is ambiguity in the parent relationship, the children for every node are well defined. A computation for one right-hand side depending on the results of that computation on the children in the derivation tree, can therefore be done in the derivation DAG as well.

Beware that η in Proposition 5.1 need not be linear in the size of a right-hand side, but is rather a generic upper bound. Courcelle and Mosbah [CM93] show (for their Proposition 3.1) linearity for evaluations using a certain Boolean algebra, and cubic complexity for sets of cardinalities. For universal evaluation they give an exponential upper bound. Lohrey [Loh12b] proves that a fixed MSO-property exists such that the problem in Proposition 5.1 is PSPACE-hard for grammars where both the rank and the number of nonterminals per right-hand side are bounded by a constant. Without these restrictions the problem is complete for EXPTIME. Note that PSPACE-completeness is already true for explicitly represented graphs.

The specific complexity thus depends on the problem. We show for two problems that they can be solved in linear time with grammar-compressed graphs as input. We then also consider the question of isomorphism for grammar-compressed tree-graphs.

5.1 Reachability

An important class of queries are *reachability queries*. For a given graph g and nodes x and y such a query asks if y is reachable from x, i.e., if there exists a path from x to y in g. It is well known that this problem can be solved in $O(|g|)$ time (e.g. by doing a BFS-traversal in $O(|V|+|E|)$ time). How can we solve this problem on an SL-HR grammar G? The following direct *linear time* algorithm essentially uses the method already applied by Lengauer and Wanke [LW88]. Their formalism is slightly different from ours however, as it uses an encoding of the hypergraphs using bipartite graphs (see also Section 3.5.3 for how that affects the size), and the algorithm as stated only decides reachability in undirected graphs. We therefore restate the algorithm using the following notion, which will again be used in the next section:

Definition 5.2. Let g be a graph with set V of external nodes. Then the *skeleton graph* $\mathsf{sk}(g) = (V, E)$ is the directed, unlabelled graph such that for all $v, v' \in V$, v' is reachable from v in $\mathsf{sk}(G)$ if and only if v' is reachable from v in $\mathsf{val}(g)$.

The edges of a skeleton graph are computed as follows. First, assume that g is a terminal graph. We determine the strongly connected components of g in linear time (e.g. using Tarjan's algorithm [Tar72]). Let g' be the corresponding graph which has as nodes the strongly connected components of g and an edge between two nodes A and B, if there is an edge from the strongly connected component A to B in g. We remove from g' each strongly connected component C that does not contain external nodes. This is done by inserting for every pair of edges e_1, e_2 where $e_1 = (D, C)$ and $e_2 = (C, E)$ (such that D, C, E are pairwise distinct) an edge $e = (D, E)$. Afterwards C and all edges incident with C are deleted. Finally, we replace each component by a cycle of the external nodes of that component, and, for an edge from a component D to a component E we add an edge from an arbitrary external node of D to one of E. If g is not a terminal graph (i.e., has nonterminal edges), let $e_1, \dots, e_n$ be the nonterminal edges labelled $A_1, \dots, A_n$. Then compute $g[e_1/\mathsf{sk}(\mathsf{rhs}(A_1))] \cdots [e_n/\mathsf{sk}(\mathsf{rhs}(A_n))]$. The resulting graph is terminal, and we can use the above procedure to compute the skeleton graph of g.

Theorem 5.3. *Let g be a graph and $G = (N, S, P)$ an SL-HR grammar with $\mathsf{val}(G) = g$. Given nodes $x, y \in V_g$, it can be determined in $O(|G|)$ time whether or not y is reachable from x in g.*

Proof. We first compute G-representations (u, x'), (v, y') of x and y in $O(\log_2 |G|_E + h)$ time, as described in Section 4.2. We traverse G bottom-up with respect to $\leq_{\mathsf{NT}}$ in one pass and compute for each nonterminal A its skeleton graph $\mathsf{sk}(A)$. After having computed in $O(|G|)$ time the skeleta for all nonterminals, we can solve a reachability query as follows. Let S' be the graph obtained from S by replacing each nonterminal edge by its skeleton graph; clearly, it can be obtained in $O(|G|)$ time.

Case 1: Assume that u and v both equal ε, i.e., x' and y' are both in the start graph. It should be clear that y is reachable from x in $\mathsf{val}(G)$ if and only if y' is reachable from x' in S'. The latter is checked in $O(|S'|)$ time.

Case 2: Let $u = u_1 \cdots u_n$ and $v = v_1 \cdots v_m$. Let $A_i = \mathsf{lhs}(\mathsf{dt}(S)[u_1 \dots u_i])$ for $i \in \{1, \dots, n\}$ and let $B_j = \mathsf{lhs}(\mathsf{dt}(S)[v_1 \dots v_j]$ for $j \in \{1, \dots, m\}$. For the right-hand side h_n of A_n we determine the set E_n of external nodes that are reachable from x' in h_n. This is done by replacing the nonterminal edges in h_n by their skeleton graphs, and then running a standard reachability test. We now move up the derivation tree (viz. to the left in u), at each step computing a subset E_i of the external nodes of $\mathsf{sk}(A_i)$: we locate the nodes corresponding E_{i+1} in $\mathsf{sk}(A_i)$ and determine the set E_i of external nodes reachable from these. Finally, we obtain a set E_0 of nodes in S' (all incident with the edge e_0). In a similar way we compute a set F_0 of nodes in S that are incident with f_0 (and can reach y'). Finally, we check if a node in F_0 is reachable from a node in E_0. This is done by adding edges over F_0 that form a cycle, and edges over E_0 that form a cycle. We now pick arbitrary nodes f in F_0 and e in E_0 and check if f is reachable from e in S'. □

Furthermore, by Lengauer and Wagner [LW92] the reachability problem on grammar-compressed graphs is P-complete.

5.2 Regular Path Queries

Regular path queries (RPQ) are a well-known (see, e.g. [Woo12]) way to query graph data: the query is given as a regular expression α over the edge alphabet of the graph g. Given two nodes

x, y in g the problem is to decide whether there exists a path from x to y such that its edge labels, taken as a string, match the regular expression α. Such queries are of relevance in modern applications: version 1.1 of the SPARQL query-language[1], for RDF data, introduces *property paths*, which are a variant of regular path queries. It is well-known that the problem whether a regular path query holds for two given nodes in a graph is decidable in time $O(|g| \cdot |\mathcal{A}_\alpha|)$ where $\mathcal{A}_\alpha$ is an NFA deciding the language $L(\alpha)$ of α. Such an automaton $\mathcal{A}_\alpha$ can be constructed in linear time and with $|\mathcal{A}_\alpha| \in O(|\alpha|)$, e.g. using Thompson's construction [Tho68]. The algorithm works as follows:

- Consider g to be an NFA $\mathcal{A}_g$ with initial state x and final state y.
- Construct the product automaton of $\mathcal{A}_g$ and $\mathcal{A}_\alpha$ deciding $L(\mathcal{A}_g) \cap L(\alpha)$.
- Test the product automaton for emptiness.

If the final test is true, then there is no such path. Otherwise it exists and we say x and y *satisfy* α. We generalize this method to grammar-compressed graphs. Given a grammar G and a regular expression α, we first compute an SL-HR grammar G_α which generates the graph-structure (ignoring initial and final states for now) of the product automaton of $\mathcal{A}_{\mathsf{val}(G)}$ and $\mathcal{A}_\alpha$. Then, using the result from the previous section, we decide reachability from the initial to the final state. To make this construction easier to read, we will relax one condition we enforced on hypergraphs before: nodes in the grammar are named not just by IDs, but by a tuple of ID and state. In fact, for a node-ID i and a set of states Q, we generate nodes (i, q) for every $q \in Q$. We refer to i as the node's ID, and q as the node's state. Consequently, this affects how nodes are named during a derivation step. The renaming ρ for nodes (i, q) always only affects the ID portion of the node, i.e., for any $q \in Q$ only renamings of the form $\rho((i, q)) = (j, q)$ are allowed. The rules by which the ID is renamed still follow the method laid out in Section 2.3. However, as there may now be distinct nodes (i, q) and (i, p) for two different states $q, p \in Q$, we also require that $\rho((i, q)) = (j, q)$ and $\rho((i, p)) = (j, p)$, i.e., nodes with the same ID will keep this property.

We recall some necessary definitions. A *nondeterministic finite automaton (NFA)* over an alphabet Σ is a tuple $\mathcal{A} = (Q, \delta, q_i, q_f)$, where Q is a finite set of states, $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the transition relation, $q_i \in Q$ is the initial state, and $q_f \in Q$ is the final state. A tuple $(q, w) \in Q \times \Sigma^*$ is called a *configuration* of $\mathcal{A}$. A configuration (p, v) can follow a configuration (q, w), denoted $(q, w) \vdash (p, v)$, if

1. $w = \sigma v$ for some $\sigma \in \Sigma$ and $(q, \sigma, p) \in \delta$, or
2. $w = v$ and $(q, \varepsilon, p) \in \delta$.

We denote the transitive and reflexive closure of $\vdash$ by $\vdash^*$. The language decided by $\mathcal{A}$ is denoted as $L(\mathcal{A}) = \{w \mid (q_i, w) \vdash^* (q_f, \varepsilon)\}$. Let $\mathcal{A} = (Q, \delta, q_i, q_f)$ and $\mathcal{A}' = (Q', \delta', q_i', q_f')$ be two

[1]`https://www.w3.org/TR/sparql11-query/`

NFA's. By $\mathcal{A} \otimes \mathcal{A}' = (Q_\otimes, \delta_\otimes)$ we denote the *product construction* defined by $Q_\otimes = Q \times Q'$ and $\delta_\otimes = \delta_\Sigma \cup \delta_\varepsilon$, where

$$\begin{aligned}
\delta_\Sigma \quad = \{ \quad ((q,q'),\sigma,(p,p')) \mid \quad & q,p \in Q, q',p' \in Q', \sigma \in \Sigma, \\
& (q,\sigma,p) \in \delta, \text{ and } (q',\sigma,p') \in \delta'\}, \text{ and} \\
\delta_\varepsilon \quad = \{ \quad ((q,q'),\varepsilon,(p,p')) \mid \quad & q,p \in Q, q',p' \in Q', \\
& (q,\varepsilon,p) \in \delta \text{ and } (q',\varepsilon,p') \in \delta', \text{ or} \\
& (q,\varepsilon,p) \in \delta \text{ and } q' = p', \text{ or} \\
& q = p \text{ and } (q',\varepsilon,p') \in \delta'\}.
\end{aligned}$$

Note that, while the product construction defines a system of state transitions, it does not set initial and final states by default. However, the NFA $(Q_\otimes, \delta_\otimes, (q_i, q_i'), (q_f, q_f'))$ does decide the language $L(\mathcal{A}) \cap L(\mathcal{A}')$. This property is used to decide whether there exists a path satisfying a regular path query between two given nodes. Such automata without initial/final states are also just called transition systems. The product construction still works if either or both the automata are transition systems. A simple graph $g = (V, E, \mathsf{att}, \mathsf{lab}, \mathsf{ext})$ with edge labels from Σ defines a transition system $\mathsf{trans}(g) = (Q_g, \delta_g)$ with $Q_g = V$ and $\delta_g = \{(x, \sigma, y) \mid \exists e \in E : \mathsf{lab}(e) = \sigma$ and $\mathsf{att}(e) = xy\}$.

Consider the product construction $\mathsf{trans}(g) \otimes \mathcal{A}$ for some simple graph g and an NFA $\mathcal{A}$. It represents every possible run of $\mathcal{A}$ on any path within g, only by setting an initial and final state we choose specific start and end-nodes within g, and initial/final states of $\mathcal{A}$. Thus, for any two nodes x, y of g we can check whether there is a path between them that $\mathcal{A}$ accepts with the same product construction. We next extend this notion to SL-HR grammars representing simple graphs, by describing a construction which, given an SL-HR grammar G and an NFA $\mathcal{A}$, generates an SL-HR grammar representing the product construction of $\mathsf{trans}(\mathsf{val}(G))$ and $\mathcal{A}$. Thus, we achieve a *compressed representation* of all runs of $\mathcal{A}$ on paths of $\mathsf{val}(G)$.

Lemma 5.4. *Given an SL-HR grammar $G = (N, P, S)$ that generates a simple graph and an NFA $\mathcal{A} = (Q, \delta, q_i, q_f)$, we can compute in $O(|G||\mathcal{A}|)$ time an SL-HR grammar $G_\mathcal{A} = (N_\mathcal{A}, P_\mathcal{A}, S_\mathcal{A})$, such that $\mathsf{trans}(\mathsf{val}(G_\mathcal{A})) = \mathsf{trans}(\mathsf{val}(G)) \otimes \mathcal{A}$.*

Proof. The construction is straightforward: for every rule $p = (A, g)$ of G, the product construction of g and $\mathcal{A}$ is computed. Due to this, the rank increases: if $\mathsf{rank}(g) = k$, then $\mathsf{rank}(g \otimes \mathcal{A}) = k|Q|$. Accordingly, some care has to be taken with nonterminal edges, as these also increase in rank. They need to be attached to the same node-IDs as before, but once for every state in Q. To make sure that nodes attached to nonterminal edges and external nodes match up correctly, we enforce some (arbitrary) order on Q.

Let $Q = \{q_1, \ldots, q_n\}$ and let $<_Q$ be a strict total order $q_1 <_Q q_2 <_Q \cdots <_Q q_n$ on Q. For a string $w = w_1 \cdots w_m$ let $w \times Q$ be the string $(w_1, q_1)(w_2, q_1) \cdots (w_m, q_1)(w_1, q_2) \cdots (w_m, q_n)$. For every $(A, g) \in P$, $P_{\mathcal{A}}$ contains a rule $(A, g_\otimes)$ defined as follows. Let $V_\otimes = V \times Q$ and

$$\begin{aligned} E_\otimes = &\{e_{i,q,\sigma,p} \mid e_i \in E_g, (q, \sigma, p) \in \delta, \mathsf{lab}_g(e_i) = \sigma\} \cup \\ &\{e_{i,q,\varepsilon,p} \mid i \in V_g, (q, \varepsilon, p) \in \delta\} \cup \\ &\{e_i \mid e_i \in E_g, \mathsf{lab}_g(e_i) \in N\} \end{aligned}$$

such that

$$\begin{aligned} \mathsf{att}_\otimes(e_{i,q,\sigma,p}) &= (x, q) \cdot (y, p) \qquad \text{(with } \mathsf{att}_g(e_i) = xy) & \mathsf{lab}_\otimes(e_{i,q,\sigma,p}) &= \sigma \\ \mathsf{att}_\otimes(e_{i,q,\varepsilon,p}) &= (i, q) \cdot (i, p) & \mathsf{lab}_\otimes(e_{i,q,\varepsilon,p}) &= \varepsilon \\ \mathsf{att}_\otimes(e_i) &= \mathsf{att}_g(e_i) \times Q & \mathsf{lab}_\otimes(e_i) &= \mathsf{lab}_g(e_i). \end{aligned}$$

For the external nodes we set $\mathsf{ext}_\otimes = \mathsf{ext} \times Q$. This defines the rules of $G_{\mathcal{A}}$. Clearly, we do not add any nonterminals, and thus have $N_{\mathcal{A}} = N$, where $\mathsf{rank}_{N_{\mathcal{A}}}(A) = |Q|\mathsf{rank}_N(A)$ for any $A \in N$. Finally $S_{\mathcal{A}}$ is constructed from S and $\mathcal{A}$ using the same construction as described above for the right-hand sides of the rules (i.e., $S_{\mathcal{A}} = S_\otimes$). □

It should be clear that $|G_{\mathcal{A}}| \in O(|G||\mathcal{A}|)$. It can now be used to decide regular path queries:

Theorem 5.5. *Given an SL-HR grammar G, where* $\mathsf{val}(G)$ *is a simple graph, an RPQ α, and a pair of nodes x, y from* $\mathsf{val}(G)$*, it can be decided in time $O(|\alpha||G|)$ whether x and y satisfy α.*

Proof. First compute an NFA $\mathcal{A}$ from α by using Thompson's well known construction [Tho68]. Note that $|\mathcal{A}| \in O(|\alpha|)$. Using Lemma 5.4 compute the grammar $G_{\mathcal{A}}$ generating the product construction of G and $\mathcal{A}$. Now, using Theorem 5.3, decide reachability from (x, q_i) to (y, q_f) in $\mathsf{val}(G_{\mathcal{A}})$. Note that x, y are node-IDs in $\mathsf{val}(G)$, but a G-representation (u, x') of x can be converted into a $G_{\mathcal{A}}$-representation of (x, q_i) by using $(u, (x', q_i))$. If a path exists, we can conclude that there is a path from x to y satisfying the RPQ α. Otherwise, it does not exist. □

As an example showing how powerful this construction is, consider the string a^{1040}, which can be represented by a grammar as $a^{1024}a^{16}$ and the obvious SL-HR grammar G representing the string-graph encoding this string. The minimal NFA $\mathcal{A}$ deciding $L = \{a^x \mid x \bmod 5 = 0\}$ has 5 states. The product construction of $\mathsf{s\text{-}graph}(a^{1040})$ and $\mathcal{A}$ would therefore have 5202 States, whereas the grammar $G_{\mathcal{A}}$ only has 155 states, and just replacing the nonterminals with their skeleta in the small start graph (20 states), e.g. when testing for reachability between $(0, q_i)$ and $(1040, q_f)$, immediately shows that there is an accepting run from the first to the last node (i.e., nodes 0 and 1040, respectively) of the graph.

We can also use this "product grammar" to decide a more general problem. In a way, the product construction encodes every pair of nodes that fulfils the query α. Thus, the product grammar $G_{\mathcal{A}}$ is a compressed representation of every such pair. We can therefore decide, whether there exists a pair of nodes at all, that satisfies the query. Intuitively, this is done by computing, bottom-up, for every rule, which external nodes can be reached from any initial state, and from which external nodes a final state can be reached. If an external node appears in both these lists at some point, we know that there is a path between some initial and final state.

Theorem 5.6. *Given an SL-HR grammar G, where* $\mathsf{val}(G)$ *is a simple graph, and an RPQ α, it can be decided in time $O(|\alpha||G|)$, if there exist nodes x, y in* $\mathsf{val}(G)$ *satisfying α.*

Proof. As before, construct $\mathcal{A}$ from α, and $G_{\mathcal{A}}$ from G and $\mathcal{A}$ according to Lemma 5.4. We now compute some helpful information in a bottom-up pass over the grammar, i.e., iterating over the rules in the reverse $\leq_{\mathsf{NT}}$-order. For a rule (A, g) we first compute whether any node (y, q_f) can be reached from any node (x, q_i) (where (x, q_i) and (y, q_f) are nodes in g) within $\mathsf{val}(g)$, using the methods from Theorem 5.3. If this is the case, a pair satisfying α exists. Otherwise, we compute two sets of nodes P_A and Q_A. Intuitively, P_A contains the external nodes of g that can be reached from some (x, q_i) (an initial state) while Q_A contains the external nodes of g from which a node (y, q_f) (a final state) is reachable.

P_A and Q_A for a rule (A, g) are computed in the following way: first we compute two sets of nodes, X and Y. The set X contains every node $x \in V_g$ with the following properties:

- there exists a nonterminal edge $e \in E_g$ such that $x \in \mathsf{att}_g(e)$, and
- if $x = \mathsf{att}_g(e)[i]$, then the i-th external node of $\mathsf{rhs}(\mathsf{lab}(e))$ is in $P_{\mathsf{lab}(e)}$.

The set Y is defined analogously, but using $Q_{\mathsf{lab}(e)}$ instead of $P_{\mathsf{lab}(e)}$. Now we replace every nonterminal edge of g by its skeleton (see Definition 5.2) and let this graph be g'. To g' we add a new node s and an edge from s to every node in $\{(x, q_i) \in V_g \mid x \in \mathbb{N}\} \cup X$. Every external node now reachable from s belongs to the set P_A. This set can be computed by a single BFS-traversal starting in s. To compute Q_A we add a new node t to g' that has an edge *from* every node in $\{(x, q_f) \in V_g \mid x \in \mathbb{N}\} \cup Y$. Every external node that can reach t is part of Q_A. This can be computed by complementing all the edges and again starting a BFS traversal from t.

If, for any $A \in N$, $P_A \cap Q_A \neq \emptyset$, then a pair of nodes x, y exists within $\mathsf{val}(A)$ such that they satisfy α. Otherwise, we compute X and Y for the start graph $S_{\mathcal{A}}$, replace the nonterminals in $S_{\mathcal{A}}$ by their skeleta, and add both s and t to $S_{\mathcal{A}}$ as above. If t is now reachable from s, then there exist nodes x, y within $\mathsf{val}(G)$ satisfying α. Otherwise no such nodes exist. ☐

Note that the constructions in this section all generalize to hypergraphs (instead of simple graphs) in a straightforward way. There is some ambiguity on how to define transition systems using hyperedges. We suggest to use a similar approach as for the definition of paths within hypergraphs (cf. Section 2.1). A hyperedge in a transition system would thus be considered as directed with one source node (the first one it is attached to) and multiple target nodes. Semantically, a rank k ($k > 2$) hyperedge labelled σ and attached to nodes $x_1 \cdots x_k$ would then be the same as $k - 1$ simple edges all labelled σ, starting in x_1, and using $x_2, \ldots, x_k$ as target nodes.

5.3 Isomorphism of Grammar-Compressed Tree-Graphs

Checking structures for isomorphism is a problem occurring in many applications and therefore of high interest. Indeed, the search for digram occurrences gRePair does is essentially a search for isomorphic subgraphs, with a very limited scope. In general, testing isomorphism for graphs is rather famous for having no known deterministic polynomial time algorithm, but also not being known to be NP-hard. For trees on the other hand, polynomial time algorithms exist (e.g. [Lin92]). A natural question is, if it is possible to test whether $\mathsf{val}(G) \cong \mathsf{val}(H)$ for two tree-generating SL-HR grammars G, H in time polynomial in $|G| + |H|$. In this section we show that this is indeed the case, using equivalent SLT grammars.

In Section 2.4 we discussed how trees can be encoded as graphs. For a tree-graph g we let $\mathsf{tree}(g)$ be the tree encoded by the graph (i.e., the reversal of $\mathsf{t\text{-}graph}$). By Theorem 3.7 we can convert a tree-generating SL-HR grammar G into a grammar G' where every right-hand side is a tree-graph. Applying $\mathsf{tree}(g)$ to every right-hand side g in the grammar thus yields an SLT grammar G_t with $\mathsf{t\text{-}graph}(\mathsf{val}(G_t)) \cong \mathsf{val}(G)$. Therefore, given two tree-generating grammars G, H testing whether $\mathsf{val}(G) \cong \mathsf{val}(H)$ according to the definition of isomorphism in Section 2.1 can be done in the following way. First, convert them to equivalent SLT grammars G_t and H_t. Then test whether $\mathsf{val}(G_t) = \mathsf{val}(H_t)$, which is possible in polynomial time (with respect to $|G_t| + |H_t|$) as shown in [BLM08].

However, this definition of isomorphism assumes the trees to be ordered, which often does not quite capture the intuition for isomorphic trees. In many situations, two trees should be considered isomorphic regardless of the order of the children. For example, consider the following two XML documents:

```
<person>                                <person>
  <first name>John</first name>           <surname>Doe</surname>
  <surname>Doe</surname>                  <first name>John</first name>
</person>                               </person>
```

They encode the same information, yet their document trees are not equal, because the nodes appear in a different order. The isomorphism definition thus should be one that respects the parent/child relationship, but disregards the specific order of the children. We next present an algorithm that checks, in polynomial time with respect to the grammar sizes, whether two SLT grammars produce isomorphic *unordered* trees by restructuring the grammars such that they generate the trees in a canonic order. The remainder of this section uses SLT grammars, because, as mentioned above, tree-generating SL-HR grammars can be converted into equivalent SLT grammars and it massively simplifies the notation.

We start with some known results about SLT grammars and notation used below. Let Δ be an alphabet. A tree $t \in T_\Delta(\{y\})$ is called a *frame* if it has *exactly one* occurrence of y. The set of all frames is referred to as $\mathcal{F}_\Delta$ (and $\mathcal{F}_{\Delta \cup N}$ for frames that include nonterminals). For a frame $t(y)$ and a tree s we write $t[s]$ for $t[y \leftarrow s]$. The following lemma is already mentioned in

Section 4.3.3 above and proved in [LMS12] for SLT grammars over ranked terminal alphabets, but carries over to unranked alphabets.

Lemma 5.7. *Any SLT grammar $G = (N, P, S)$ over Δ can be transformed, in polynomial time, into an equivalent SLT grammar, where each rule has one of the following four types (where $\delta \in \Delta$ and $A, B, C, A_1, \dots, A_k \in N$):*

1. $A \to \delta(A_1, \dots, A_k)$,
2. $A \to B(C)$,
3. $A(y) \to \delta(A_1, \dots, A_i, y, A_{i+1}, \dots, A_k)$, *or*
4. $A(y) \to B(C(y))$.

We assume that every SLT grammar below has this form, and let $N^{(0)} = \{A \in N \mid \mathsf{rank}(A) = 0\}$ and $N^{(1)} = \{A \in N \mid \mathsf{rank}(A) = 1\}$. For an SLT grammar G in the normal form from Lemma 5.7, we let $G(i)$ with $i \in [4]$ be the SLT grammar (without start nonterminal) consisting of all rules of G of type i from the Lemma. We denote by $t \Rightarrow_{G(i)} s$ a derivation using a rule of type i. Occasionally, we also consider SLT grammars, where the start nonterminal belongs to $N^{(1)}$, i.e., has rank 1. We call such a grammar a 1-*SLT grammar*. Note that $\mathsf{val}(G)$ for a 1-SLT grammar G is a frame.

A straight-line program over Δ (SLP, i.e., a grammar-compressed string) can be seen as a 1-SLT grammar $G = (N, P, S)$ containing only rules of the form $A(y) \to B(C(y))$ and $A(y) \to \delta(y)$ with $B, C \in N$ and $\delta \in \Delta$ and vice-versa. Thus, if G produces $\mathsf{val}(G) = a_1(\cdots a_n(y) \cdots)$ then it represents the string $a_1 \cdots a_n$. We also write $\mathsf{val}(G) = a_1 \cdots a_n$ in this case. The following result follows from [Hag00, Algorithm 8.1.4], where it is shown that an "interval grammar" (an SLP with right-hand sides of the form $A[l, r]$ for positions $l \leq r$) can be transformed into regular SLPs (in Chomsky normal form) of quadratic size.

Lemma 5.8. *Let G be an SLP and $l, r \in [|\mathsf{val}(G)|]$ two binary encoded numbers with $l \leq r$. One can compute in polynomial time an SLP G' such that $\mathsf{val}(G') = \mathsf{val}(G)[l, r]$.*

5.3.1 Length-lexicographical Ordering

For a tree $t \in T_\Delta$ we denote by $\mathsf{dflr}(t)$ its depth-first left-to-right (or pre-order) traversal string in Δ^*. It is defined as $\mathsf{dflr}(\delta(t_1, \dots, t_k)) = \delta \mathsf{dflr}(t_1) \cdots \mathsf{dflr}(t_k)$ for every $\delta \in \Delta$, $k \geq 0$, and $t_1, \dots, t_k \in T_\Delta$.

The *length-lexicographical ordering* $<_{\mathsf{llex}}$ on Δ^* is defined by $u <_{\mathsf{llex}} w$ if and only if (i) $|u| < |w|$ or (ii) $|u| = |w|$ and $u <_{\mathsf{lex}} w$. We extend the definition of $<_{\mathsf{llex}}$ to trees s, t over Δ by $s <_{\mathsf{llex}} t$ if and only if $\mathsf{dflr}(s) <_{\mathsf{llex}} \mathsf{dflr}(t)$.

Lemma 5.9. *Given SLT grammars G, H, it is decidable in polynomial time, whether or not*

1. $\mathsf{val}(G) <_{\mathsf{llex}} \mathsf{val}(H)$, *and*
2. $\mathsf{val}(G) = \mathsf{val}(H)$.

Proof. As mentioned above, point 2 is shown in [BLM08] by computing from G, H in polynomial time SLPs G', H' with $\mathsf{val}(G') = \mathsf{dflr}(\mathsf{val}(G))$ and $\mathsf{val}(H') = \mathsf{dflr}(\mathsf{val}(H))$. Equivalence of SLPs can be decided in polynomial time; this was proved independently in [HJM96, MSU97, Pla94], cf. [Loh12a].

To show point 1, we compute in two single bottom-up runs the numbers $n_1 = |\mathsf{val}(G')|$ and $n_2 = |\mathsf{val}(H')|$. If $n_1 \neq n_2$ we are done; so assume that $n = n_1 = n_2$. Next, we compute the first position for which the strings $\mathsf{val}(G')$ and $\mathsf{val}(H')$ differ. This is done via binary search and polynomially many equivalence tests: We compute $m = \lceil n/2 \rceil$ and, using Lemma 5.8, construct SLPs G_1 and G_2 for $\mathsf{val}(G')[1, m]$ and $\mathsf{val}(G')[m + 1, n]$, respectively, and SLPs H_1 and H_2 for $\mathsf{val}(H')[1, m]$ and $\mathsf{val}(H')[m + 1, n]$, respectively. We proceed with G_1 and H_1 if $\mathsf{val}(G_1) \neq \mathsf{val}(H_1)$, otherwise we proceed with G_2 and H_2. After $c \leq \lceil \log(n) \rceil$ many steps we obtain SLPs G_c, H_c representing the first position for which $\mathsf{val}(G')$ and $\mathsf{val}(H')$ differ. We compute the terminal symbols g, h with $\mathsf{val}(G_c) = \{g\}$ and $\mathsf{val}(H_c) = \{h\}$ and determine whether or not $g <_\Delta h$. □

5.3.2 Canonizing SLT grammars

For a tree t we denote with $\mathsf{uo}(t)$ the unordered rooted version of t. It is the node-labelled directed graph (V, E, λ) where $V = D(t)$ is the set of nodes, $E = \{(u, u.i) \mid i \in \mathbb{N}, u \in \mathbb{N}^*, u.i \in D(t)\}$ is the edge relation, and λ is the node-labelling function with $\lambda(u) = t[u]$. For an SLT grammar G, we also write $\mathsf{val}_{\mathsf{uo}}(G)$ for $\mathsf{uo}(\mathsf{val}(G))$.

In this section we fix the terminal alphabet Δ and the order $<_\Delta$ on Δ. As defined in Section 5.3.1, this induces the order $<_{\mathsf{llex}}$ on trees in T_Δ. For a tree $t \in T_\Delta$ we define its *canon* $\mathsf{canon}(t)$ as follows: Let $t = \delta(t_1, \dots, t_k)$ with $\delta \in \Delta$ and $k \geq 0$ and let $c_i = \mathsf{canon}(t_i)$ for $i \in [k]$. Let $c_{i_1} \leq_{\mathsf{llex}} c_{i_2} \leq_{\mathsf{llex}} \dots \leq_{\mathsf{llex}} c_{i_k}$ be the length-lexicographically ordered list of the canons $c_1, \dots, c_k$. Then $\mathsf{canon}(t) = \delta(c_{i_1}, \dots, c_{i_n})$. The following lemma can be shown by an induction on the tree structure:

Lemma 5.10. *Let $s, t \in T_\Delta$. Then $\mathsf{uo}(s)$ and $\mathsf{uo}(t)$ are isomorphic if and only if $\mathsf{canon}(s) = \mathsf{canon}(t)$.*

In the following, we denote a tree $A_1(A_2(\cdots A_n(t) \cdots))$, where $A_1, A_2, \dots, A_n$ are unary nonterminals with $A_1 A_2 \cdots A_n(t)$.

Theorem 5.11. *From a given SLT grammar G one can construct in polynomial time an SLT grammar G' such that $\mathsf{val}(G') = \mathsf{canon}(\mathsf{val}(G))$.*

Proof. Let $G = (N, P, S)$ be an SLT grammar over Δ. We assume that G contains no distinct nonterminals $A_1, A_2 \in N^{(0)}$ such that $\mathsf{val}_G(A_1) = \mathsf{val}_G(A_2)$. This is justified because we can test $\mathsf{val}_G(A_1) = \mathsf{val}_G(A_2)$ in polynomial time by Lemma 5.9 (and replace A_2 by A_1 in G in such a case). We will add polynomially many new nonterminals to G and change the rules for nonterminals from $N^{(0)}$ such that for the resulting SLT grammar G' we have $\mathsf{val}_{G'}(Z) = \mathsf{canon}(\mathsf{val}_G(Z))$ for every $Z \in N^{(0)}$.

Consider a nonterminal $Z \in N^{(0)}$ and let M be the set of all nonterminals in G that can be reached from Z. By induction, we can assume that G already satisfies $\mathsf{val}_G(A) = \mathsf{canon}(\mathsf{val}_G(A))$ for every $A \in M^{(0)} \setminus \{Z\}$. We distinguish two cases.

Case (i). Z is of type (1) from Lemma 5.7, i.e., has a rule $Z \to \delta(A_1, \dots, A_k)$. Using Lemma 5.9 we construct an ordering $i_1, \dots, i_k$ of $[k]$ such that $\mathsf{val}_G(A_{i_1}) \leq_{\mathsf{llex}} \mathsf{val}_G(A_{i_2}) \leq_{\mathsf{llex}} \cdots \leq_{\mathsf{llex}} \mathsf{val}_G(A_{i_k})$. We obtain G' by replacing the rule $Z \to \delta(A_1, \dots, A_k)$ by $Z \to \delta(A_{i_1}, \dots, A_{i_k})$ and get $\mathsf{val}_{G'}(Z) = \mathsf{canon}(\mathsf{val}_G(Z))$.

Case (ii). Z is of type (2), i.e., has a rule $Z \to B(A)$. Let $\{S_1, \dots, S_m\} = M^{(0)} \setminus \{Z\}$ be an ordering such that

$$\mathsf{val}_G(S_1) <_{\mathsf{llex}} \mathsf{val}_G(S_2) <_{\mathsf{llex}} \cdots <_{\mathsf{llex}} \mathsf{val}_G(S_m).$$

Note that A is one of these S_i. The sequence $S_1, S_2, \dots, S_m$ partitions the set of all trees t in T_Δ into intervals $\mathcal{I}_0, \mathcal{I}_1, \dots, \mathcal{I}_m$ with

- $\mathcal{I}_0 = \{t \in T_\Delta \mid t <_{\mathsf{llex}} \mathsf{val}_H(S_1)\}$,
- $\mathcal{I}_i = \{t \in T_\Delta \mid \mathsf{val}_H(S_i) \leq_{\mathsf{llex}} t <_{\mathsf{llex}} \mathsf{val}_H(S_{i+1})\}$ for $1 \leq i < m$, and
- $\mathcal{I}_m = \{t \in T_\Delta \mid \mathsf{val}_H(S_m) \leq_{\mathsf{llex}} t\}$.

Consider the maximal $G(4)$-derivation starting from $B(A)$, i.e.,

$$B(A) \Rightarrow^*_{G(4)} B_1 B_2 \cdots B_N(A),$$

where B_i is a type-(3) nonterminal. Clearly, the number N might be of exponential size, but the set $\{B_1, \dots, B_N\}$ can be easily constructed. In order to construct an SLT for $\mathsf{canon}(\mathsf{val}_G(Z))$, it remains to reorder the arguments in right-hand sides of the type-(3) nonterminals B_i. The problem is that different occurrences of a type-(3) nonterminal in the sequence $B_1 B_2 \cdots B_N$ have to be reordered in a different way. But we will show that the sequence $B_1 B_2 \cdots B_N$ can be split into $m+1$ blocks such that all occurrences of a type-(3) nonterminal in one of these blocks have to be reordered in the same way.

Let $t_k = \mathsf{val}_G(B_k B_{k+1} \cdots B_N(A))$ for $k \in [N]$ and $t_{N+1} = \mathsf{val}_G(A)$. Note that $t_1 = \mathsf{val}_G(Z) >_{\mathsf{llex}} \mathsf{val}_G(S_m)$ and that $t_{k+1} <_{\mathsf{llex}} t_k$ for all k. For $i \in [m]$ let k_i be the maximal position $k \leq N+1$ such that $t_k \geq_{\mathsf{llex}} \mathsf{val}_G(S_i)$. Since $t_1 \geq_{\mathsf{llex}} \mathsf{val}_G(S_m) \geq_{\mathsf{llex}} \mathsf{val}_G(S_i)$ this position is well defined. Also note that if $A = S_i$, then we have $k_i = k_{i-1} = \cdots = k_1 = N+1$. For every $0 \leq i \leq m$, the interval $[k_{i+1}+1, k_i]$ is the set of all positions k such that $\mathsf{val}_G(t_k) \in \mathcal{I}_i$. Here we set $k_{m+1} = 0$ and $k_0 = N+1$. Clearly, the interval $[k_{i+1}+1, k_i]$ might be empty. The positions $k_0, \dots, k_m$ can be computed in polynomial time, using binary search combined with Lemma 5.9. To apply the latter, note that for a given position k we can compute in polynomial time an SLT grammar for the tree t_k using Lemma 5.8 for the SLP consisting of all type-(4) rules that are used to derive $B_1 B_2 \cdots B_N$.

We now factorize the string $B_1 B_2 \cdots B_N$ as $B_1 B_2 \cdots B_N = u_m u_{m-1} \cdots u_0$, where $u_m = B_1 \cdots B_{k_m - 1}$ and $u_i = B_{k_{i+1}} \cdots B_{k_i - 1}$ for $0 \leq i \leq m-1$. By Lemma 5.8 we can compute in polynomial time an SLP G_i for the string u_i. For the further consideration, we view

G_i as a 1-SLT grammar consisting only of type-(4) rules. Note that $\mathsf{val}(G_i)$ is a linear tree, where every node is labelled with a type-(3) nonterminal. We now add reordered versions of type-(3) rules to G_i. Consider a type-(3) rule $(C(y) \to \delta(A_1, \ldots, A_j, y, A_{j+1}, \ldots, A_k)) \in P$ where $C \in \{B_1, \ldots, B_N\}$. Then we add to G_i the type-(3) rule

$$C(y) \to \delta(A_{j_1}, \ldots, A_{j_\nu}, y, A_{j_{\nu+1}}, \ldots, A_{j_k}),$$

where $\{j_1, \ldots, j_k\} = [k]$ and $0 \leq \nu \leq k$ are chosen such that

1. $\mathsf{val}_G(A_{j_1}) \leq_{\mathsf{llex}} \mathsf{val}_G(A_{j_2}) \leq_{\mathsf{llex}} \cdots \leq_{\mathsf{llex}} \mathsf{val}_G(A_{j_k})$ and
2. $\mathsf{val}_G(A_{j_\nu}) \leq_{\mathsf{llex}} \mathsf{val}_G(S_i) <_{\mathsf{llex}} \mathsf{val}_G(A_{j_{\nu+1}})$.

Note that if $\nu = k$ then condition (2) states that $\mathsf{val}_G(A_{j_k}) \leq_{\mathsf{llex}} \mathsf{val}_G(S_i)$, and if $\nu = 0$ then it states that $\mathsf{val}_G(S_i) <_{\mathsf{llex}} \mathsf{val}_G(A_{j_1})$. Also note that condition (2) ensures that for every tree $t \in \mathcal{I}_i$ we have $\mathsf{val}_G(A_{j_\nu}) \leq_{\mathsf{llex}} t <_{\mathsf{llex}} \mathsf{val}_G(A_{j_{\nu+1}})$. Hence, $\mathsf{val}_G(\delta(A_{j_1}, \ldots, A_{j_\nu}, t, A_{j_{\nu+1}}, \ldots, A_{j_k}))$ is a canon. The crucial observation now is that the above factorization $u_m u_{m-1} \cdots u_0$ of $B_1 B_2 \cdots B_N$ was defined in such a way that for every occurrence of a type-(3) nonterminal $C(y)$ in u_i, the parameter y will be substituted by a tree from $\mathcal{I}_i$ during the derivation from Z to $\mathsf{val}_G(Z)$. Hence, we reorder the arguments in the right-hand sides of nonterminal occurrences in u_i in the correct way to obtain a canon.

We now rename the nonterminals in the SLT grammars G_i (which are now of type (3) and type (4)) so that the nonterminal sets of $G, G_0, \ldots, G_m$ are pairwise disjoint. Let $X_i(y)$ be the start nonterminal of G_i after the renaming. Then we add to the current SLT grammar G the union of all the G_i, and replace the rule $Z \to B(A)$ by $Z \to X_m X_{m-1} \cdots X_0(A)$. The construction implies that $\mathsf{val}_{G'}(Z) = \mathsf{canon}(\mathsf{val}_G(Z))$ for the resulting grammar G'.

It remains to argue that the above construction can be carried out in polynomial time. All steps only need polynomial time in the size of the current SLT grammar. Hence, it suffices to show that the size of the SLT grammar is polynomially bounded. The algorithm is divided into $|N^{(0)}|$ many phases, where in each phase it enforces $\mathsf{val}_{G'}(Z) = \mathsf{canon}(\mathsf{val}_G(Z))$ for a single nonterminal Z. Consider a single phase, where $\mathsf{val}_{G'}(Z) = \mathsf{canon}(\mathsf{val}_G(Z))$ is enforced for a nonterminal Z. In this phase, we (i) change the rule for Z and (ii) add new type-(3) and type-(4) rules to G (the union of the G_i above). But the number of these new rules is polynomially bounded in the size of the initial SLT grammar (the one before the first phase), because the nonterminals introduced in earlier phases are not relevant for the current phase. This implies that the additive size increase in each phase is bounded polynomially in the size of the initial grammar. □

Corollary 5.12. *The problem of deciding whether* $\mathsf{val}_{\mathsf{uo}}(G_1)$ *and* $\mathsf{val}_{\mathsf{uo}}(G_2)$ *are isomorphic for given SLT grammars* G_1 *and* G_2 *is* P*-complete.*

Proof. Membership in P follows immediately from Lemma 5.9, Lemma 5.10, and Theorem 5.11. Moreover, P-hardness already holds for dags, i.e., SLT grammars where all nonterminals have rank 0, as shown in [LM13]. □

5.3.3 Isomorphism of Unrooted, Unordered SLT-Compressed Trees

We can extend this result further. For two tree-generating grammars we can also test in polynomial time, whether the (undirected) graph structures they generate are isomorphic regardless of the root-node. An unrooted unordered tree t over Δ can be seen as a node-labelled (undirected) graph $t = (V, E, \lambda)$, where $E \subseteq V \times V$ is symmetric and $\lambda : V \to \Delta$. For two nodes u, v of t we denote the distance between them (i.e., the number of edges on the path from u to v in t) by $\mathsf{distance}_t(u, v)$. We further define the eccentricity of a node v as $\mathsf{ecc}_t(v) = \max_{u \in V} \mathsf{distance}_t(u, v)$ and the diameter $\phi(t) = \max_{v \in V} \mathsf{ecc}_t(v)$.

Let $t \in T_\Delta$ be a rooted ordered tree over Δ and let $t' = \mathsf{uo}(t) = (V, E, \lambda)$ be the rooted unordered tree corresponding to t. The tree $\mathsf{ur}(t') = (V, E \cup E^{-1}, \lambda)$ over Δ is the unrooted version of t'. An unrooted unordered tree t can be represented by an SLT grammar G by forgetting the order and root information present in G. Let $\mathsf{val}_{\mathsf{ur},\mathsf{uo}}(G) = \mathsf{ur}(\mathsf{uo}(\mathsf{val}(G)))$.

In this section it is proved that isomorphism for unrooted unordered trees t_1, t_2 represented by SLT grammars G_1, G_2, respectively, can be solved in polynomial time with respect to $|G_1| + |G_2|$. We reduce the problem to the (rooted) unordered case solved in Corollary 5.12.

Let $t = (V, E, \lambda)$ be an unordered unrooted tree. A node u of t is called *centre node of t* if for all leaves v of t:

$$\mathsf{distance}_t(u, v) \leq (\phi(s) + 1)/2.$$

Let $\mathsf{centre}(t)$ be the set of all centre nodes of t. One can compute the centre nodes by deleting all leaves of the tree and iterating this step, until the current tree consists of at most two nodes. These are the centre nodes of t. In particular, t has either one or two centre nodes. Another characterization of centre nodes that is important for our algorithm is via longest paths. Let $p = (v_0, v_1, \ldots, v_n)$ be a longest simple path in t, i.e., $n = \phi(t)$. Then the middle points $v_{\lfloor n/2 \rfloor}$ and $v_{\lceil n/2 \rceil}$ (which are identical if n is even) are the centre nodes of t. These nodes are independent of the concrete longest path p.

Note that there are two centre nodes if and only if $\phi(t)$ is odd. Since our constructions are simpler if a unique centre node exists, we first make sure that $\phi(t)$ is even. Let $\#$ be a new symbol not in Δ. For an unrooted unordered tree t we denote by $\mathsf{even}(t)$ the tree where every pair of edges $(u, v), (v, u)$ is replaced by the edges $(u, v'), (v', v), (v, v'), (v', u)$, where v' is a new node labelled $\#$. Then for an SLT grammar $G = (N, P, S)$ over Δ we let $\mathsf{even}(G) = (N, P', S)$ be the SLT grammar over $\Delta \cup \{\#\}$ where P' is obtained from P by replacing every subtree $\delta(t_1, \ldots, t_k)$ with $\delta \in \Delta$, $k \geq 1$, in a right-hand side by the subtree $\delta(\#(t_1), \ldots, \#(t_k))$. Observe that

- $\mathsf{val}_{\mathsf{ur},\mathsf{uo}}(\mathsf{even}(G)) = \mathsf{even}(\mathsf{val}_{\mathsf{ur},\mathsf{uo}}(G))$,
- $\phi(\mathsf{even}(t)) = 2 \cdot \phi(t)$ is even, i.e., $\mathsf{even}(t)$ has only one centre node, and
- trees t and s are isomorphic if and only if $\mathsf{even}(t)$ and $\mathsf{even}(s)$ are isomorphic.

Since $\mathsf{even}(G)$ can be constructed in polynomial time, we assume in the following that every SLT grammar produces a tree of even diameter and therefore has only one centre node. For a tree t of even diameter, we denote with $\mathsf{centre}(t)$ its unique centre node.

Let $u \in V$. We construct a rooted version $\mathsf{root}(t, u)$ of t, with root node u. We set $\mathsf{root}(t, u) = (V, E', \lambda)$, where $E' = \{(v, v') \in E \mid \mathsf{distance}_t(u, v) < \mathsf{distance}_t(u, v')\}$.

Two unrooted unordered trees t_1, t_2 of even diameter are isomorphic if and only if $\mathsf{root}(t_1, \mathsf{centre}(t_1))$ is isomorphic to $\mathsf{root}(t_2, \mathsf{centre}(t_2))$. Thus, we can solve in polynomial time the isomorphism problem for unrooted unordered trees represented by SLT grammars G, G' by

1. determining in polynomial time compressed representations $\tilde{u}_1$ and $\tilde{u}_2$ of $u_1 = \mathsf{centre}(\mathsf{val}_{\mathsf{ur,uo}}(G))$ and $u_2 = \mathsf{centre}(\mathsf{val}_{\mathsf{ur,uo}}(G'))$, respectively (Section 5.3.4),
2. constructing in polynomial time SLT grammars G_1, G_2 such that $\mathsf{val}_{\mathsf{uo}}(G_1) = \mathsf{root}(\mathsf{val}_{\mathsf{ur,uo}}(G), u_1)$ and $\mathsf{val}_{\mathsf{uo}}(G_2) = \mathsf{root}(\mathsf{val}_{\mathsf{ur,uo}}(G'), u_2)$ (Section 5.3.5), and
3. testing in polynomial time if $\mathsf{val}_{\mathsf{uo}}(G_1)$ is isomorphic to $\mathsf{val}_{\mathsf{uo}}(G_2)$ (Corollary 5.12).

5.3.4 Finding a Root-Node

Let $G = (N, P, S)$ be an SLT grammar over Δ. A *G-compressed path* p is a string of pairs $p = (A_1, u_1) \cdots (A_n, u_n)$ such that for all $i \in [n]$, $A_i \in N$, $A_1 = S$, $u_i \in D(t_i)$ is a Dewey address in t_i where $(A_i \to t_i) \in P$, $t_i[u_i] = A_{i+1}$ for $i < n$, and $t_i[u_n] \in \Delta$. If we omit the condition $t_i[u_n] \in \Delta$, then p is a partial G-compressed path. Note that by definition, $n \leq |N|$. A partial G-compressed path uniquely represents one particular node in the derivation tree of G, and a G-compressed path represents a leaf of the derivation tree and hence a node of $\mathsf{val}(G)$. We denote this node by $\mathsf{val}_G(p)$. The concatenation $u_1, u_2, \ldots, u_n$ of the Dewey addresses is denoted by $u(p)$.

Recall that the height of a tree $s \in T_\Delta$ is defined as $\mathsf{height}(s) = \max\{|u| + 1 \mid u \in D(t)\}$. For a frame $t(y) \in \mathcal{F}_\Delta$ we define $\mathsf{ecc}(t) = \mathsf{ecc}_t(y)$ (recall that in a frame there is a unique occurrence of the parameter y) and $\mathsf{rty}(t) = \mathsf{distance}_t(\varepsilon, y)$ (the distance from the root to the parameter y). We extend these notions to frames $t \in \mathcal{F}_{\Delta \cup N}$ and trees $s \in T_{\Delta \cup N}$ by $\mathsf{ecc}(t) = \mathsf{ecc}(\mathsf{val}_G(t))$, $\mathsf{rty}(t) = \mathsf{rty}(\mathsf{val}_G(t))$, and $\mathsf{height}(s) = \mathsf{height}(\mathsf{val}_G(s))$.

Eccentricity, distance from root to y, and height can be computed in polynomial time for all nonterminals bottom-up. To do so, observe that for two frames $t(y), t'(y) \in \mathcal{F}_{\Delta \cup N}$ and a tree $s \in T_{\Delta \cup N}$ we have

- $\mathsf{rty}(t[t']) = \mathsf{rty}(t) + \mathsf{rty}(t')$,
- $\mathsf{ecc}(t[t']) = \max\{\mathsf{ecc}(t'), \mathsf{ecc}(t) + \mathsf{rty}(t')\}$, and
- $\mathsf{height}(t[s]) = \max\{\mathsf{height}(s), \mathsf{rty}(t) + \mathsf{height}(s)\}$.

Similarly, for a frame $t(y) = \delta(s_1, \ldots s_i, y, s_{i+1}, \ldots, s_k)$ and a tree $s = \delta(s_1, \ldots, s_k)$ we have:

- $\mathsf{rty}(t) = 1$,

- $\mathsf{ecc}(t) = 2 + \max\{\mathsf{height}(s_i) \mid 1 \leq i \leq k\}$, and
- $\mathsf{height}(s) = 1 + \max\{\mathsf{height}(s_i) \mid 1 \leq i \leq k\}$.

Finally, note that for the tree $t[s]$ ($t(y) \in \mathcal{F}_\Delta, s \in T_\Delta$) we have

$$\phi(t[s]) = \max\{\phi(t), \phi(s), \mathsf{ecc}(t) + \mathsf{height}(t)\}. \tag{5.1}$$

Our search for the centre node of an SLT-compressed tree is based on the following lemma. For a frame $t(y) \in \mathcal{F}_\Delta$, where u is the Dewey address of the parameter y, and a tree $s \in T_\Delta$ we say that a node v of $t[s]$ belongs to t if the Dewey address of v is in $D(t) \setminus \{u\}$. Otherwise, we say that v belongs to s, which means that u is a prefix of the Dewey address of v.

Lemma 5.13. *Let $t(y) \in \mathcal{F}_\Delta$ be a frame and $s \in T_\Delta$ a tree such that $\phi(t[s])$ is even. Let $c = \mathsf{centre}(t[s])$. Then we have the following:*

- *If $\mathsf{ecc}(t) \leq \mathsf{height}(s)$ then c belongs to s.*
- *If $\mathsf{ecc}(t) > \mathsf{height}(s)$ then c belongs to t.*

Proof. Let us first assume that $\mathsf{ecc}(t) \leq \mathsf{height}(s)$, Then we have $\phi(t) \leq 2 \cdot \mathsf{ecc}(t) \leq \mathsf{ecc}(t) + \mathsf{height}(s)$, i.e., $\phi(t[s]) = \max\{\phi(s), \mathsf{ecc}(t) + \mathsf{height}(s)\}$ by (5.1). Together with $\mathsf{ecc}(t) \leq \mathsf{height}(s)$ this implies that the middle point of a longest path in $s[t]$ (which is c) belongs to the tree s.

Next, assume that $\mathsf{ecc}(t) = \mathsf{height}(s) + 1$. Then we have $\phi(s) \leq 2 \cdot \mathsf{height}(s) < \mathsf{ecc}(t) + \mathsf{height}(s)$, i.e., $\phi(t[s]) = \max\{\phi(t), \mathsf{ecc}(t) + \mathsf{height}(s)\}$. Moreover, we claim that $\mathsf{ecc}(t) + \mathsf{height}(s) \geq \phi(t)$. In case $\phi(t) = \mathsf{ecc}(t)$, this is clear. Otherwise, $\phi(t) > \mathsf{ecc}(t)$ and a longest path in t does not end in the parameter node y. It follows that $\phi(t) \leq 2 \cdot (\mathsf{ecc}(t) - 1) < \mathsf{ecc}(t) + \mathsf{height}(s)$. Thus, we have $\phi(t[s]) = \mathsf{ecc}(t) + \mathsf{height}(s) = 2 \cdot \mathsf{height}(s) + 1$, which is odd, a contradiction. Hence, this case cannot occur.

Finally, assume that $\mathsf{ecc}(t) > \mathsf{height}(s) + 1$. Again, we get $\phi(t[s]) = \max\{\phi(t), \mathsf{ecc}(t) + \mathsf{height}(s)\}$. Moreover, since $\mathsf{ecc}(t) > \mathsf{height}(s) + 1$ the centre nodes c must belong to t. □

Lemma 5.14. *For a given SLT grammar G such that $\mathsf{val}_{\mathsf{ur},\mathsf{uo}}(G)$ has even diameter, one can construct a G-compressed path for $\mathsf{centre}(\mathsf{val}_{\mathsf{ur},\mathsf{uo}}(G))$.*

Proof. Consider the recursive Algorithm 9. It is started with $t_l = t_r = p = \varepsilon$ and $A = S$ and computes the node $\mathsf{centre}(\mathsf{val}_{\mathsf{ur},\mathsf{uo}}(G))$. The following invariants are preserved by the algorithm: If $\mathsf{centre}(t_l, A, t_r, p)$ is called, then we have:

- If A has rank 0 then $t_r = \varepsilon$
- $\mathsf{val}(G) = \mathsf{val}(t_l[A[t_r]])$ (here we set $\varepsilon[t] = t = t[\varepsilon]$).
- The tree $t_l[A[t_r]]$ can be derived from the start variable S.
- p is the partial G-compressed path to the distinguished A in $t_l[A[t_r]]$.
- $\mathsf{centre}(\mathsf{val}_{\mathsf{ur},\mathsf{uo}}(G))$ belongs to the subframe $\mathsf{val}(A)$ in $\mathsf{val}(t_l)[\mathsf{val}(A)[\mathsf{val}(t_r)]]$.

Algorithm 9 Recursive procedure to find the G-compressed path for the centre node

```
procedure centre($t_l, A, t_r, p$)
    if $A \to B(C)$ (and thus $t_r = \varepsilon$) or $A(y) \to B(C(y))$ then
        if $\mathsf{ecc}(t_l[B(y)]) \leq \mathsf{height}(C[t_r])$ then
            return($t_l[B(y)], C, t_r, p \cdot (A, 1)$)
        else
            return($t_l, B, C[t_r], p \cdot (A, \varepsilon)$)
    if $A \to \delta(A_1, \dots, A_k)$ (and thus $t_r = \varepsilon$) then
        $t_i \leftarrow t_l(\delta(A_1, \dots, A_{i-1}, y, A_{i+1}, \dots, A_k)$ for all $i \in [k]$
        if there is an $i \in [k]$ such that $\mathsf{ecc}(t_i) \leq \mathsf{height}(A_i)$ then
            return($t_i, A_i, \varepsilon, p \cdot (A, i)$)
        else
            return ($p \cdot (A, \varepsilon)$)
    if $A(y) \to \delta(A_1, \dots A_{s-1}, y, A_{s+1}, \dots, A_k)$ then
        $t_i \leftarrow t_l(\delta(A_1, \dots, A_{i-1}, y, A_{i+1}, \dots, A_{s-1}, t_r, A_{s+1}, \dots, A_k)$ if $i < s$
        $t_i \leftarrow t_l(\delta(A_1, \dots, A_{s-1}, t_r, A_{s+1}, \dots, A_{i-1}, y, A_{i+1}, \dots, A_k)$ if $s < i$
        if there is an $i \in [k] \setminus \{s\}$ such that $\mathsf{ecc}(t_i) \leq \mathsf{height}(A_i)$ then
            return($t_i, A_i, \varepsilon, p \cdot (A_i, i)$)
        else
            return ($p \cdot (A, \varepsilon)$)
```

For a call $\mathsf{centre}(t_l, A, t_r, p)$, the algorithm distinguishes on the right-hand side of A. If this right-hand side has the form $A(B)$ or $A(B(y))$, then, by comparing $\mathsf{ecc}(t_l[B(y)])$and $\mathsf{height}(C[t_r])$, we determine, whether the search for the centre node has to continue in B or C, see Lemma 5.13.

The case that the right-hand side of A has the form $\delta(A_1, \dots, A_k)$ is slightly more involved. Let $s_l = \mathsf{val}(t_l)$ and $s_i = \mathsf{val}(A_i)$ (by the first invariant we know that $t_r = \varepsilon$). We have to find the centre node of $t := s_l(\delta(s_1, \dots, s_k)$ and by the last invariant we know that it is contained in $\delta(s_1, \dots, s_k)$. We now consider all k many cuts of t along one of the edges between the δ-node and one of the s_i, i.e., we cut t into $s_l(\delta(s_1, \dots, s_{i-1}, y, s_{i+1}, \dots, s_k)$ and s_i. Using again Lemma 5.13, it suffices to compare $\mathsf{ecc}(s_l(\delta(s_1, \dots, s_{i-1}, y, s_{i+1}, \dots, s_k)))$ and $\mathsf{height}(s_i)$ in order to determine whether the centre node belongs to $s_l(\delta(s_1, \dots, s_{i-1}, y, s_{i+1}, \dots, s_k)$ or s_i. If for some i, it turns out that the centre node is in s_i, then we continue the search with A_i. Finally, assume that for all i, it turns out that the centre node is in $s_l(\delta(s_1, \dots, s_{i-1}, y, s_{i+1}, \dots, s_k)$. Since by the last invariant, the centre node is in $\delta(s_1, \dots, s_k)$, the δ-labelled node must be the centre node. The case of a rule $A(y) \to \delta(A_1, \dots A_{s-1}, y, A_{s+1}, \dots, A_k)$ can be dealt with similarly.

Finally, note that whenever $\mathsf{ecc}(t)$ and $\mathsf{height}(t)$ have to be determined by the algorithm, then t is a polynomial size tree built from terminal and nonterminal symbols. By the previous remarks, $\mathsf{ecc}(t)$ and $\mathsf{height}(t)$ can be computed in polynomial time. □

5.3.5 Moving the Root Node

Let $G = (N, P, S)$ be an SLT grammar over Δ (as usual, having the normal form from Lemma 5.7) and p a G-compressed path. Let $s(p) \in T_{\Delta \cup N}$ be the tree defined inductively as follows: Let $(A \to t) \in P$ and $u \in D(t)$. Then $s((A, u)) = t$. If $p = (A, t)p'$ with p' non-empty, then either (i) $u = \varepsilon$ and $t = B(C)$ or (ii) $u = i \in \mathbb{N}$ and $t[i] \in N^{(0)}$. In case (i) we set $s(p) = s(p')[C]$, in case (ii) we set $s(p) = t'[s(p')]$, where $t'(y)$ is obtained from t by replacing the i-th argument of the root by y. Note that $s(p') \in \mathcal{F}_{\Delta \cup N}$ if p' starts with a nonterminal of rank 1. Let $s = s(p)$; its size is bounded by the size of G. Note that $s[u(p)]$ is a terminal symbol (recall that $u(p)$ denotes the concatenation of the Dewey addresses in p). Assume that $s[u(p)] = \delta \in \Delta$. Let # be a fresh symbol and let s' be obtained from s by changing the label at $u(p)$ from δ to #. Let $s' \Rightarrow_G^* s''$ be the shortest derivation such that $s''[\varepsilon] = \gamma \in \Delta$ (it consists of at most $|N|$ derivation steps). We denote the #-labelled node in s'' by u. Finally, let t be obtained from s'' by changing the unique # into δ. We define the *p-expansion* of G, denoted $\mathsf{ex}_G(p)$, as the tuple (t, u, δ, γ). Note that $\mathsf{val}_G(p)$ is the unique #-labelled node in $\mathsf{val}_G(s'')$. Moreover, the p-expansion can be computed in polynomial time from G and p.

The p-expansion (t, u, δ, γ) has all information needed to construct a grammar G' representing the rooted version at p of $\mathsf{val}(G)$. If $u = \varepsilon$ then also $\mathsf{val}_G(p) = \varepsilon$. Since G is already rooted at ε nothing has to be done in this case and we return $G' = G$. If $u \neq \varepsilon$ then $\mathsf{val}_G(p) \neq \varepsilon$ and hence t contains two terminal nodes which uniquely represent the root node and the node $\mathsf{val}_G(p)$ of the tree $\mathsf{val}(G)$.

Let $s_1 \in T_\Delta$ be a rooted ordered tree representing the unrooted unordered tree $\tilde{s}_1 = \mathsf{ur}(\mathsf{uo}(s_1))$. Let $u \neq \varepsilon$ be a node of s_1. Let $s_1[\varepsilon] = \gamma \in \Delta$ and $s_1[u] = \delta \in \Delta$. A rooted ordered tree s_2 that represents the rooted unordered tree $\tilde{s}_2 = \mathsf{root}(\tilde{s}_1, u)$ can be defined as follows: Since $u \neq \varepsilon$, we can write

$$s_1 = \gamma(\zeta_1, \ldots, \zeta_{i-1}, t'[\delta(\xi_1, \ldots, \xi_m)], \zeta_{i+1}, \ldots, \zeta_k),$$

where t' is a frame, and $u = iu'$, where u' is the Dewey address of the parameter y in t'. We can define s_2 as

$$s_2 = \delta(\xi_1, \ldots, \xi_m, \mathsf{rooty}(t')[\gamma(\zeta_1, \ldots, \zeta_{i-1}, \zeta_{i+1}, \ldots, \zeta_k)]), \tag{5.2}$$

where rooty is a function mapping frames to frames defined recursively as follows, where $f \in \Delta$, $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_\ell \in T_\Delta$, and $t(y), t'(y) \in \mathcal{F}_\Delta$:

$$\mathsf{rooty}(y) = y \tag{5.3}$$

$$\mathsf{rooty}(f(t_1, \ldots, t_{i-1}, y, t_{i+1}, \ldots, t_\ell)) = f(t_1, \ldots, t_{i-1}, y, t_{i+1}, \ldots, t_\ell) \tag{5.4}$$

$$\mathsf{rooty}(t[t'(y)]) = \mathsf{rooty}(t')[\mathsf{rooty}(t(y))] \tag{5.5}$$

Intuitively, the mapping rooty unroots a frame $t(y)$ towards its y-node u, i.e., it reverses the path from the root to u. Thus, for instance, $\mathsf{rooty}(f(a, y, b)) = f(a, y, b)$ and $\mathsf{rooty}(f(a, g(c, y, d), b)) = g(c, f(a, y, b), d)$.

Lemma 5.15. *From a given SLT grammar G and a G-compressed path p one can construct in polynomial time an SLT grammar G' such that* $\mathsf{val}_{\mathsf{uo}}(G')$ *is isomorphic to* $\mathsf{root}(\mathsf{val}_{\mathsf{ur,uo}}(G), \mathsf{val}_G(p))$.

Proof. Let $G = (N, P, S)$ over Δ and $\mathsf{ex}_G(p) = (t, u, \delta, \gamma)$. If $u = \varepsilon$ then define $G' = G$. If $u \neq \varepsilon$ then we can write

$$t = \gamma(B_1, \ldots, B_{i-1}, t'[\delta(\xi_1, \ldots, \xi_m)], B_{i+1}, \ldots, B_k), \tag{5.6}$$

where $B_j \in N^{(0)}$, $\xi_j \in T_N$, t' is a frame composed of nonterminals $A \in N^{(1)}$ and frames $f(\zeta_1, \ldots, \zeta_{j-1}, y, \zeta_{j+1}, \ldots, \zeta_l)$ ($\zeta_j \in T_N$), and $u = iu'$, where u' is the Dewey address of the parameter y in t'.

We define $G' = (N \uplus N', P', S)$ where $N' = \{A' \mid A \in N^{(1)}\}$. To define the rule set P', we extend the definition of rooty to frames from $\mathcal{F}_{\Delta \cup N}$ by (i) allowing in the trees t_j from Equation (5.4) also nonterminals, and (ii) defining for every $B \in N^{(1)}$, $\mathsf{rooty}(B(y)) = B'(y)$. We now define the set of rules P' of P as follows: We put all rules from P except for the start rule $(S \to s) \in P$ into P'. For the start variable S we add to P' the rule

$$S \to \delta(\xi_1, \ldots, \xi_m, \mathsf{rooty}(t')[\gamma(B_1, \ldots, B_{i-1}, B_{i+1}, \ldots, B_k)]).$$

Moreover, let $A \in N^{(1)}$ and $(A(y) \to \zeta) \in P$. If this is a type-(3) rule, then we add $A'(y) \to \zeta$ to P'. If $\zeta = B(C(y))$ then add $A'(y) \to C'(B'(y))$ to P'.

Claim. Let $A \in N^{(1)}$. Then $\mathsf{val}_{G'}(A') = \mathsf{rooty}(\mathsf{val}_G(A))$.

The claim is easily shown by induction on the reverse hierarchical structure of G: Let $(A \to t_A) \in P$. If $t_A = f(A_1, \ldots, A_j, y, A_{j+1}, \ldots, A_l)$ then $\mathsf{rooty}(\mathsf{val}_G(A)) = \mathsf{val}_G(A)$. Since $(A' \to t_A) \in P'$ and G' contains all rules of G except for the start rule, we obtain $\mathsf{val}_{G'}(A') = \mathsf{rooty}(\mathsf{val}_G(A))$. If $t_A = B(C(y))$ then, by Equation (5.5), $\mathsf{rooty}(\mathsf{val}_G(B(C(y)))) = \mathsf{rooty}(\mathsf{val}_G(C))[\mathsf{val}_G(B)]$. By induction the latter is equal to $\mathsf{val}_{G'}(C')[\mathsf{val}_{G'}(B')]$ which equals $\mathsf{val}(A')$ by the definition of the right-hand side of A'. This proves the claim.

The above claim implies that $\mathsf{val}_{G'}(\mathsf{rooty}(c(y))) = \mathsf{rooty}(\mathsf{val}_G(c(y)))$ for every frame $c(y)$ that is composed of frames $f(\zeta_1, \ldots, \zeta_{j-1}, y, \zeta_{j+1}, \ldots, \zeta_l)$ ($\zeta_j \in T_N$) and nonterminals $A \in N^{(1)}$. In particular, $\mathsf{val}_{G'}(\mathsf{rooty}(t')) = \mathsf{rooty}(\mathsf{val}_G(t'(y)))$ for the frame t' from Equation (5.6). Hence, with $s_j = \mathsf{val}_{G'}(\xi_j) = \mathsf{val}_G(\xi_j)$ and $t_j = \mathsf{val}_{G'}(B_j) = \mathsf{val}_G(B_j)$ we obtain

$$\begin{aligned}
\mathsf{val}(G') &= \mathsf{val}_{G'}(\delta(\xi_1, \ldots, \xi_m, \mathsf{rooty}(t')[\gamma(B_1, \ldots, B_{i-1}, B_{i+1}, \ldots, B_k)])) \\
&= \delta(s_1, \ldots, s_m, \mathsf{val}_{G'}(\mathsf{rooty}(t'))[\gamma(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_k)]) \\
&= \delta(s_1, \ldots, s_m, \mathsf{rooty}(\mathsf{val}_G(t'))[\gamma(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_k)]).
\end{aligned}$$

Since $\mathsf{val}(G) = \gamma(t_1, \ldots, t_{i-1}, \mathsf{val}_G(t')[\delta(s_1, \ldots, s_m)], t_{i+1}, \ldots, t_k)$, it follows that $\mathsf{val}_{\mathsf{uo}}(G')$ is isomorphic to $\mathsf{root}(\mathsf{val}_{\mathsf{ur,uo}}(G), \mathsf{val}_G(p))$. □

Corollary 5.16. *Given SLT grammars G_1 and G_2, deciding whether or not $\mathsf{val}_{\mathsf{ur,uo}}(G_1)$ and $\mathsf{val}_{\mathsf{ur,uo}}(G_2)$ are isomorphic is* P*-complete.*

Proof. Membership in P follows from Lemma 5.14, Lemma 5.15, and Corollary 5.12. Hardness for P follows from the P-hardness for dags [LM13] and the fact that isomorphism of rooted unordered trees can be reduced to isomorphism of unrooted unordered trees by labelling the roots with a fresh symbol. □

5.4 Related Work

As mentioned at the beginning of Chapter 5 above, compressed input often induces an exponential blow-up in complexity when deciding queries on the represented data. Such "upgrading-theorems" have been shown for example for the representation of graphs as boolean circuits, which can be exponentially smaller than an explicit representation, but in turn make queries exponentially harder to answer (see e.g. [GW83, PY86]). Fortunately, upgrading theorems do not hold for grammar-compressed data. Therefore it is possible, to achieve a speed-up proportional to the compression ratio for certain queries. Various such algorithms are known for SLPs and SLT grammars. For recent surveys see [Loh12a] and [Loh15], respectively.

SL-HR grammars are also known in literature as "hierarchical graph definitions". Some algorithmic properties of problems on graphs represented by such hierarchical graph definitions are studied by Lengauer and Wanke [LW88]. They show that connectivity of grammar-compressed graphs can be decided in linear time with respect to the size of the compressed representation. They also show this for biconnectivity and (undirected) reachability. For strong connectivity however, they only get a quadratic upper bound. It should be noted that there is still an increase in complexity here, but a smaller one. Lengauer and Wagner [LW92] study several problems and their complexity on explicit versus hierarchical representations. They show, for example, that reachability is complete for P on hierarchical graphs, but the problem is NL-complete for explicit representations. The latter class is widely believed to be a proper subset of the former, but both classes only include problems which can be solved deterministically in polynomial time, so there is no exponential blow-up in computation time. As the hierarchical representation can be exponentially smaller than the represented graph, this makes a speed-up possible. This smaller increase in complexity is also stated by Marathe et al. [MRHR97, MHR94, MHSR98], who show that some NP-complete problems on graphs (e.g., 3-colourization, Max Cut, and Vertex Cover) become PSPACE-hard on hierarchically defined graphs, which again does not imply an exponential blow-up in computation time (assuming widely-held assumptions in complexity theory). They also present polynomial approximation algorithms for some of these problems with hierarchical input. Note that we cannot conclude that hierarchical versions of classical problems always lead to a speed-up: in [LW92] they also show that the P-complete threshold network flow problem and the circuit value problem become PSPACE-complete for hierarchical input graphs.

5.5 Discussion and Further Research

This chapter focussed on queries, where the answer can be provided faster on a grammar-compressed representation than on the explicit graph. Input compression usually slows computation down, but as already known for SLPs and SLT grammars, grammar-based compression has the somewhat unique property of allowing for speed-up queries. The reason for this is that the hierarchical representation allows for precomputation and reuse of partial results on the rules of the grammar. We start by mentioning a general result that MSO model checking is

PSPACE-complete. However, specific formulae can be checked in polynomial time. We then recap that this is possible for (s,t)-reachability and present algorithms to solve regular path queries in a single bottom-up pass of the grammar, reusing the reachability result in the process. Finally we look into the isomorphism problem and show that deciding whether two tree-generating grammars produce isomorphic trees (regardless of the order of the hyperedges attachment relations) is possible in time polynomial in the size of the input grammars.

For future work investigations into several more graph algorithms are of interest. To give some examples: graph pattern matching is not, in general, an easy problem but is often used in graph databases as it essentially represents what conjunctive queries are within a graph database (see e.g. [Woo12]). Investigating its complexity for a grammar-compressed representation could be a useful step in making this a viable choice for a graph data storage model, provided the result works well in practice. Somewhat easier might be to work on queries used in network analysis, like average clustering coefficient, density, or PageRank. Finally, deciding isomorphism for general graph grammars, or at least larger classes than tree-generating, would be very interesting. We note however that this is a difficult problem for explicitly represented graphs as well: at the time of writing, no polynomial time algorithm is known to decide the isomorphism of two graphs. Therefore, even showing that isomorphism of grammar-compressed graphs is in NP, or even PSPACE is a non-trivial achievement.

Chapter 6

Conclusions

The main result of the thesis is the generalization of the RePair compression scheme to graphs, using hyperedge-replacement graph grammars. We see that an exact application of the RePair method is not feasible, as currently no practically viable method to the problem of finding a maximal set of non-overlapping digram occurrences is known. Instead a working greedy approximation based on a node order is presented. Using the right order, this approximation will yield the same result on a string-graph, as RePair would otherwise give on a string. For tree-graphs this is also true (given the right order) for the initial replacement of digrams. Due to the differences in size definition, however, gRePair will remove more rules during the pruning step than TreeRePair does. Experimentally it can be verified that tree-graphs still compress to a similar level using gRePair, as the represented trees are compressed by TreeRePair. We also give examples on how the maximal rank can affect the compression and experimentally verify that these effects can indeed occur. The design decisions and choice of formalism are explained, and finally an experimental evaluation is presented based on a prototype. This evaluation shows competitive performance on some graphs, particularly on RDF and version graphs. Also note that the compression of web graphs is promising, if a dense substructure removal is applied first, even though gRePair by itself does not compress web graphs particularly well.

The second half of the thesis considers algorithmic questions. First it presents two ways of dealing with neighbourhood queries. The first one is a set of direct algorithms, which work on any given SL-HR grammar, but for a grammar with a total of m edges and height h the algorithms need $O(h^2)$ time per neighbour and an additional preprocessing using $O(\log_2 m \cdot h)$ time for a neighbourhood query. This is followed by a data structure, which can be used to retrieve a neighbour in constant time. It works by precomputing the possible paths that can be taken within one step in the derivation tree, which induces $O(|G||\Sigma|kh^2)$ additional precomputation time and space requirements in the order of $O(|G||\Sigma|kh)$, where k is the maximal rank of the grammar and G is the grammar. However, the method only works on a restricted set of grammars, representing graphs where every node has at most one incoming and outgoing edge per label. After neighbourhood queries, speed-up algorithms are considered. There are already a few known results for hierarchical graph definitions, which are essentially the same as the

SL-HR grammars used here. We extend this with a new result computing regular path queries. Doing so, we see that an SL-HR grammar G can be combined with a finite automaton $\mathcal{A}$ to compute a new SL-HR grammar which represents the product automaton of $\mathsf{val}(G)$ and $\mathcal{A}$ in $O(|G||\mathcal{A}|)$ space. Finally, we show that it is possible to decide isomorphism of tree-generating grammars in time polynomial in the size of the two grammars.

6.1 Discussion of the Results

Grammar-based compression is well understood for strings and trees. Some work exists on algorithmic aspects of hierarchical graph definitions, thus there clearly is interest in grammar-based representations of graphs. Achieving a small such representation, however was barely considered. The work presented here closes part of this gap. We successfully show that it is possible to compress the more complex graph data structure using known approximation methods for small grammars. We experimentally verify that some real-world graphs compress very well using this method, as expected particularly on graphs exhibiting a lot of repetition. However, the best results come from rather atypical graphs: the compressor works best with graphs having low amounts of locality and the two RDF graphs compressing best have only slightly more edges than nodes. In general, the method struggles with dense subgraphs, possibly the strongest indicator for the Conjecture 3.9 to be true. This makes it a somewhat specialised solution, but it does excel in these cases.

The second half focusses on algorithms on graphs represented by grammars and first presents traversal methods and a speed-up algorithm for regular path queries. These are interesting auxiliary results, as they present ways to use the achieved grammars. More work can be done here, particularly with the constant-delay traversal. The method presented here is still rather limited in application, due to the strong restriction it imposes on the grammar. However, if the number of edges per node for every label can be bounded by a constant c then the method can be extended, at the cost of an additional factor c in precomputation time/space. The thesis closes with addressing the question of isomorphism of two straight-line tree grammars. The complexity of this problem is settled with it being P-complete, by showing that it is possible in polynomial time (with respect to the size of the input grammars) to decide, whether two SLT grammars generate isomorphic trees. As an extension, it is also shown that this still holds for unrooted trees (i.e., tree-graphs without a dedicated root-node).

6.2 Directions for Future Work

Finally we mention some interesting questions and issues that are left open for now and could be approached in future work. First is the possibility of using node replacement grammars. As discussed in Section 3.5.1, a good digram definition may be more involved for NR grammars. But once found, the compressor likely works better: the encoding of the grammar should be simpler, because it is not necessary to deal with hyperedges. Furthermore, counting digram occurrences might be easier as there are generally fewer pairs of nodes than pairs of edges in a graph, and dense substructures may be less of a problem.

The method as stated, based on hyperedge-replacement grammars, also has potential for improvement. The currently used encoding of the start graph may not be the best choice. Particularly with incidence matrices of hyperedges, very large matrices are compressed that consist almost exclusively of zeroes. This makes the k^2-trees used quite unbalanced. A new method designed with hypergraphs in mind, might give better results. A further investigation into more node orders might also lead to an improvement. Finally the prototype implementation is very slow, particularly in decompression (and thus, the pruning step). A technical direction for future work would therefore be to provide a more thoughtful implementation, which would definitely improve the practical viability of the method. RePair as a method generally uses a fair amount of space during computation and has originally been proposed as an offline-method. Graph compression often works online or on streaming data to deal with graphs too large to fit into memory. A very interesting direction would be to look into possibilities to improve gRePair's memory performance. A technique based on recompression [Jez16] could be a way to achieve an algorithm that works for streaming data, considering the incoming edges as updates. A similar method was proposed for grammar-based compression of trees [BHJM16] and works quite well. For a general idea on more space-efficient RePair compression we further point to recent work by Bille et al. [BGP17] as a start.

As a final direction more algorithms could be investigated for viability of speed-up queries. PageRank might be a good choice. Graph pattern matching on grammar-compressed graphs is also very interesting and could provide some insight into the question of isomorphism, as pattern matching on graphs is essentially subgraph isomorphism. One thing noticeable when considering speed-up algorithms for grammar-based compression of trees is that these often work only by computing a normal-form for the grammar first. This normal form is particularly well suited to many computational approaches, due to it structuring the tree into a set of spine-trees. Working on similar normal forms for SL-HR grammars might therefore be a valuable approach.

Bibliography

[ABF96] A. Amir, G. Benson, and M. Farach, *Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files*, J. Comput. Syst. Sci. (1996), 299–307.

[ÁBF+15] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, M. A. Martínez-Prieto, and G. Navarro, *Compressed vertical partitioning for efficient RDF management*, Knowl. Inf. Syst. **44** (2015), no. 2, 439–474.

[AD09] A. Apostolico and G. Drovandi, *Graph Compression by BFS*, Algorithms **2** (2009), no. 3, 1031–1044.

[ÁdBBN17] S. Álvarez-García, G. de Bernardo, N. R. Brisaboa, and G. Navarro, *A succinct data structure for self-indexing ternary relations*, J. Discrete Algorithms **43** (2017), 38–53.

[BC87] M. Bauderon and B. Courcelle, *Graph expressions and graph rewritings*, Mathematical Systems Theory **20** (1987), no. 2-3, 83–127.

[BC08] G. Buehrer and K. Chellapilla, *A Scalable Pattern Mining Approach to Web Graph Compression with Communities*, WSDM, 2008, pp. 95–106.

[BCFN15] N. R. Brisaboa, A. Cerdeira-Pena, A. Fariña, and G. Navarro, *A compact RDF store using suffix arrays*, SPIRE, 2015, pp. 103–115.

[BGP17] P. Bille, I. L. Gørtz, and N. Prezza, *Space-efficient re-pair compression*, DCC, 2017, pp. 171–180.

[BHJM16] S. Böttcher, R. Hartel, T. Jacobs, and S. Maneth, *Incremental updates on compressed XML*, ICDE, 2016, pp. 1026–1037.

[BLM08] G. Busatto, M. Lohrey, and S. Maneth, *Efficient memory representation of XML document trees*, Inf. Syst. **33** (2008), no. 4-5, 456–474.

[BLN14] N. R. Brisaboa, S. Ladra, and G. Navarro, *Compact representation of web graphs with extended functionality*, Inf. Syst. **39** (2014), 152–174.

[BMPT97] M. Beaudry, P. McKenzie, P. Péladeau, and D. Thérien, *Finite moniods: From word to circuit evaluation*, SIAM J. Comput. **26** (1997), no. 1, 138–152.

[BP98] S. Brin and L. Page, *The anatomy of a large-scale hypertextual web search engine*, Computer Networks **30** (1998), no. 1-7, 107–117.

[BSV09] P. Boldi, M. Santini, and S. Vigna, *Permuting web graphs*, Algorithms and Models for the Web-Graph, 2009, pp. 116–126.

[BV04] P. Boldi and S. Vigna, *The webgraph framework I: compression techniques*, WWW, 2004, pp. 595–602.

[CFI92] J. Cai, M. Fürer, and N. Immerman, *An optimal lower bound on the number of variables for graph identifications*, Combinatorica **12** (1992), no. 4, 389–410.

[CL11] F. Claude and S. Ladra, *Practical representations for web and social graphs*, ACM CIKM, 2011, pp. 1185–1190.

[CLL$^+$05] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, *The smallest grammar problem*, IEEE Transactions on Information Theory **51** (2005), no. 7, 2554–2576.

[CM93] B. Courcelle and M. Mosbah, *Monadic Second-Order Evaluations on Tree-Decomposable Graphs*, Theor. Comput. Sci. **109** (1993), no. 1&2, 49–82.

[CN10] F. Claude and G. Navarro, *Fast and Compact Web Graph Representations*, TWEB **4** (2010), no. 4, 16:1–16:31.

[DKH97] F. Drewes, H.-J. Kreowski, and A. Habel, *Hyperedge Replacement Graph Grammars*, Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, 1997, pp. 95–162.

[DL98] N. Deo and B. Litow, *A structural approach to graph compression*, MFCS Workshop on Communications, 1998, pp. 91–101.

[Dre99] F. Drewes, *A characterization of the sets of hypertrees generated by hyperedge-replacement graph grammars*, Theory Comput. Syst. **32** (1999), no. 2, 159–208.

[Edm65] J. Edmonds, *Paths, trees, and flowers*, Canad. J. Math. **17** (1965), 449–467.

[EH92] J. Engelfriet and L. Heyker, *Context-Free Hypergraph Grammars have the Same Term-Generating Power as Attribute Grammars*, Acta Inf. **29** (1992), no. 2, 161–210.

[Eli75] P. Elias, *Universal codeword sets and representations of the integers*, IEEE Transactions on Information Theory **21** (1975), no. 2, 194–203.

[Eng97] Joost Engelfriet, *Context-free graph grammars*, Handbook of Formal Languages, Vol. 3 (Grzegorz Rozenberg and Arto Salomaa, eds.), 1997, pp. 125–213.

[FGM10] J. D. Fernández, C. Gutierrez, and M. A. Martínez-Prieto, *RDF compression: basic approaches*, WWW, 2010, pp. 1091–1092.

[FM13] A. Farzan and J. I. Munro, *Succinct encoding of arbitrary graphs*, Theor. Comput. Sci. (2013), 38–52.

[FMG10] J. D. Fernández, M. A. Martínez-Prieto, and C. Gutierrez, *Compact representation of large RDF data sets for publishing and exchange*, ISWC, 2010, pp. 193–208.

[FP16] J. Fischer and D. Peters, *GLOUDS: representing tree-like graphs*, J. Discrete Algorithms **36** (2016), 39–49.

[FT98] M. Farach and M. Thorup, *String matching in lempel-ziv compressed strings*, Algorithmica **20** (1998), no. 4, 388–404.

[GB14] S. Grabowski and W. Bieniecki, *Tight and simple web graph compression for forward and reverse neighbor queries*, Discrete Applied Mathematics **163** (2014), 298–306.

[GKPS05] L. Gasieniec, R. M. Kolpakov, I. Potapov, and P. Sant, *Real-time traversal in grammar-based compressed files*, DCC, 2005, p. 458.

[GW83] H. Galperin and A. Wigderson, *Succinct representations of graphs*, Information and Control **56** (1983), no. 3, 183–198.

[GW97] R. Goldman and J. Widom, *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*, VLDB, 1997, pp. 436–445.

[Hab92] A. Habel, *Hyperedge Replacement: Grammars and Languages*, Lecture Notes in Computer Science, 1992.

[Hag00] C. Hagenah, *Gleichungen mit regulären randbedingungen über freien gruppen*, Ph.D. thesis, University of Stuttgart, Institut für Informatik, 2000, (in German).

[HJM96] Y. Hirshfeld, M. Jerrum, and F. Moller, *A polynomial algorithm for deciding bisimilarity of normed context-free processes*, Theor. Comput. Sci. **158** (1996), no. 1&2, 143–159.

[HK86] A. Habel and H.-J. Kreowski, *May we introduce to you: hyperedge replacement*, Graph-Grammars and Their Application to Computer Science, 3rd International Workshop, 1986, pp. 15–26.

[HKL93] A. Habel, H. Kreowski, and C. Lautemann, *A Comparison of Compatible, Finite, and Inductive Graph Properties*, Theor. Comput. Sci. **110** (1993), no. 1, 145–168.

[HN14] C. Hernández and G. Navarro, *Compressed representations for web and social graphs*, Knowl. Inf. Syst. **40** (2014), no. 2, 279–313.

[IPR05] L. Iannone, I. Palmisano, and D. Redavid, *Optimizing RDF storage removing redundancies: An algorithm*, IEA/AIE, 2005, pp. 732–742.

[Jac89] G. J. Jacobson, *Space-efficient static trees and graphs*, FOCS, 1989, pp. 549–554.

[Jez16] A. Jez, *Recompression: A simple and powerful technique for word equations*, J. ACM **63** (2016), no. 1, 4:1–4:51.

[JHD13] A. K. Joshi, P. Hitzler, and G. Dong, *Logical linked data compression*, ESWC, 2013, pp. 170–184.

[JZG$^+$13] X. Jiang, X. Zhang, F. Gao, C. Pu, and P. Wang, *Graph compression strategies for instance-focused semantic mining*, Linked Data and Knowledge Graph, Communications in Computer and Information Science, 2013, pp. 50–61.

[Lin92] S. Lindell, *A logspace algorithm for tree canonization (extended abstract)*, STOC, 1992, pp. 400–404.

[LM00] N. J. Larsson and A. Moffat, *Off-line dictionary-based compression*, Proceedings of the IEEE (2000), 1722–1732.

[LM06] M. Lohrey and S. Maneth, *The complexity of tree automata and XPath on grammar-compressed trees*, Theor. Comp. Sci. **363** (2006), no. 2, 196–210.

[LM13] M. Lohrey and C. Mathissen, *Isomorphism of regular trees and words*, Inf. Comput. **224** (2013), 71–105.

[LMM13] M. Lohrey, S. Maneth, and R. Mennicke, *XML tree structure compression using RePair*, Information Systems **38** (2013), no. 8, 1150–1167.

[LMP15] M. Lohrey, S. Maneth, and F. Peternek, *Compressed tree canonization*, ICALP, 2015, pp. 337–349.

[LMR16] M. Lohrey, S. Maneth, and C. P. Reh, *Traversing grammar-compressed trees with constant delay*, DCC, 2016, pp. 546–555.

[LMS12] M. Lohrey, S. Maneth, and M. Schmidt-Schauß, *Parameter reduction and automata evaluation for grammar-compressed trees*, J. Comput. Syst. Sci. **78** (2012), no. 5, 1651–1669.

[Loh12a] M. Lohrey, *Algorithmics on SLP-compressed strings: A survey*, Groups Complexity Cryptology **4** (2012), no. 2, 241–299.

[Loh12b] M. Lohrey, *Model-checking hierarchical structures*, JCSS **78** (2012), no. 2, 461–490.

[Loh15] ———, *Grammar-Based Tree Compression*, DLT, 2015, pp. 46–57.

[LS00] H. Liefke and D. Suciu, *XMILL: an efficient compressor for XML data*, SIGMOD, 2000, pp. 153–164.

[Lu14] H. Lu, *Linear-time compression of bounded-genus graphs into information-theoretically optimal number of bits*, SIAM J. Comput. (2014), 477–496.

[LW88] T. Lengauer and E. Wanke, *Efficient solution of connectivity problems on hierarchically defined graphs*, SIAM J. Comput. **17** (1988), no. 6, 1063–1080.

[LW92] T. Lengauer and K. W. Wagner, *The correlation between the complexities of the nonhierarchical and hierarchical versions of graph problems*, JCSS **44** (1992), no. 1, 63–93.

[Mas12] H. Maserrat, *Compression of social networks*, Ph.D. thesis, Simon Fraser University, 2012.

[Mei08] M. Meier, *Towards rule-based minimization of RDF graphs under constraints*, RR, 2008, pp. 89–103.

[MFC12] M. A. Martínez-Prieto, J. D. Fernández, and R. Cánovas, *Compression of RDF dictionaries*, SAC, 2012, pp. 340–347.

[MHR94] M. V. Marathe, H. B. Hunt III, and S. S. Ravi, *The Complexity of Approximation PSPACE-Complete Problems for Hierarchical Specifications*, Nord. J. Comput. **1** (1994), no. 3, 275–316.

[MHSR98] M. V. Marathe, H. B. Hunt III, R. E. Stearns, and V. Radhakrishnan, *Approximation Algorithms for pspace-Hard Hierarchically and Periodically Specified Problems*, SIAM J. Comput. **27** (1998), no. 5, 1237–1261.

[MP16] S. Maneth and F. Peternek, *Compressing graphs by grammars*, ICDE, 2016, pp. 109–120.

[MP17] ______, *Grammar-based graph compression*, CoRR **abs/1704.05254** (2017).

[MRHR97] M. V. Marathe, V. Radhakrishnan, H. B. Hunt III, and S. S. Ravi, *Hierarchically Specified Unit Disk Graphs*, Theor. Comput. Sci. **174** (1997), no. 1-2, 23–65.

[MS04] N. Markey and P. Schnoebelen, *A PTIME-complete matching problem for SLP-compressed words*, Inf. Process. Lett. **90** (2004), no. 1, 3–6.

[MS10] S. Maneth and T. Sebastian, *Fast and tiny structural self-indexes for XML*, CoRR **abs/1012.5696** (2010).

[MSU97] K. Mehlhorn, R. Sundar, and C. Uhrig, *Maintaining dynamic sequences under equality tests in polylogarithmic time*, Algorithmica **17** (1997), no. 2, 183–198.

[NW97] C. G. Nevill-Manning and I. H. Witten, *Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm*, JAIR **7** (1997), 67–82.

[OR02] E. Otte and R. Rousseau, *Social network analysis: a powerful strategy, also for the information sciences*, J. Information Science **28** (2002), no. 6, 441–453.

[Pes07] L. Peshkin, *Structure induction by lossless graph compression*, DCC, 2007, pp. 53–62.

[PGR$^+$14] J. Z. Pan, J. M. Gómez-Pérez, Y. Ren, H. Wu, H. Wang, and M. Zhu, *Graph pattern based RDF data compression*, JIST, 2014, pp. 239–256.

[Pla94] W. Plandowski, *Testing equivalence of morphisms on context-free languages*, ESA, 1994, pp. 460–470.

[PPSW10] R. Pichler, A. Polleres, S. Skritek, and S. Woltran, *Redundancy elimination on RDF graphs in the presence of rules, constraints, and queries*, RR, 2010, pp. 133–148.

[PR99] W. Plandowski and W. Rytter, *Complexity of language recognition problems for compressed words*, Jewels are Forever, 1999, pp. 262–272.

[PY86] C. H. Papadimitriou and M. Yannakakis, *A Note on Succinct Representations of Graphs*, Information and Control **71** (1986), no. 3, 181–185.

[SG15] J. Swacha and S. Grabowski, *OFR: An Efficient Representation of RDF Datasets*, SLATE, 2015, pp. 224–235.

[Tar72] R. E. Tarjan, *Depth-First Search and Linear Graph Algorithms*, SIAM J. Comput. **1** (1972), no. 2, 146–160.

[Tho68] K. Thompson, *Regular Expression Search Algorithm*, Commun. ACM **11** (1968), no. 6, 419–422.

[Tur84] G. Turán, *On the succinct representation of graphs*, Discrete Applied Mathematics (1984), 289–294.

[UMD$^+$13] J. Urbani, J. Maassen, N. Drost, F. J. Seinstra, and H. E. Bal, *Scalable RDF data compression with MapReduce*, Concurrency and Computation: Practice and Experience **25** (2013), no. 1, 24–39.

[WL68] B. Weisfeiler and A. A. Lehman, *A reduction of a graph to a canonical form and an algebra arising during this reduction*, Nauchno-Technicheskaya Informatsia **Seriya 2** (1968), no. 9, 12–16, (in Russian).

[Woo12] P. T. Wood, *Query Languages for Graph Databases*, SIGMOD Record (2012), 50–60.

[Ši09] I. Šimeček, *Sparse Matrix Computations Using the Quadtree Storage Format*, SYNASC, 2009, pp. 168–173.