

Best Trees Extraction and Contextual Grammars for Language Processing

Anna Jonsson

UMEÅ UNIVERSITY



UMEÅ UNIVERSITY

Best Trees Extraction and Contextual Grammars for Language Processing

Anna Jonsson

Thesis, June 2021
Department of Computing Science
Umeå University
Sweden

This thesis was partially funded by Vetenskapsrådet (DNR 2012-04555).

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

aj@cs.umu.se

Copyright © 2021 by Anna Jonsson
Except for Paper I, © Association for Computational Linguistics, 2017
Paper II, © Elsevier, 2017
Paper V, © Springer-Verlag, 2018

ISBN 978-91-7855-521-5 (print)
ISBN 978-91-7855-522-2 (digital)
ISSN 0348-0542
UMINF 21.04

Cover illustration by Lina Lidmark.
Printed by UmU Tryckservice, Umeå University, 2021.

Abstract

In natural language processing, the *syntax* of a sentence refers to the words used in the sentence, their grammatical role, and their order. *Semantics* instead concerns the concepts represented by the words in the sentence and their relations – the meaning of the sentence. While a human can easily analyse a sentence to figure out its grammatical construction and meaning, this is a difficult task for a computer that lacks a language model in the form of a human brain. To start with, the computer needs data structures that can represent syntax and semantics. Then, the computer must have some information about what is considered correct syntax and semantics – this can be provided in the form of human-annotated corpora of natural language. There are several ways to capture the correctness aspect of a corpus in a computer-based language model, but here we consider rule-based formalisms only. Rule-based formalisms use rule sets to define *formal languages* and are in that sense very similar to the grammatical rules of natural language. The goal is thus to model natural languages using formal languages.

Trees are commonly used to represent syntactic analyses of sentences, and graphs can represent their semantics. Examples of rule-based formalisms that define languages of trees and graphs are *tree automata* and *graph grammars*, respectively. When used in language processing, these formalisms are normally assigned weights, meaning that the trees and graphs in the languages defined by the formalisms have weights; this assignment is done to allow us to rank the trees and graphs by comparing them to the information extracted from the corpora of correct linguistic data. There are numerous ~~systems that combine~~ rule-based formalisms ~~with algorithms working on their rule sets~~ to solve language processing tasks.

Since natural language is very complicated to model, there are many small gaps in the research of natural language processing to dive into. The research of this thesis regards two separate problems: First, we have the *N-best problem*, which is about finding $N \in \mathbb{N}$ top-ranked hypotheses given a hypothesis space represented by a rule-based formalism; the hypotheses can for example have the form of syntax trees. The second problem is that of *semantic modelling*, whose aim is to find a formalism powerful enough to define languages of semantic representations. This model can however not be too powerful since we still want to be able to efficiently compute answers to language processing tasks.

This thesis is divided into two parts according to the two problems introduced above. The first part covers the N-best problem for weighted tree automata. In this line of research, we develop and evaluate multiple versions of an algorithm that efficiently computes the N best (top-ranked) trees in a tree language given by a weighted tree automaton. In the second part, we study how rule-based formalisms can be used to model graphs that represent meaning, i.e., semantic graphs. We investigate an existing formalism for this purpose and thereby learn what properties of that formalism are necessary for semantic modelling. Finally, we use our new-found knowledge to develop an alternative formalism, and argue that it is better suited for the task of semantic modelling than existing formalisms.

"alternative" låter lite som om det skulle vara ett stort avsteg. Skulle föreslå "more specialised" e.dyl. Se även sid 5.

Acknowledgements

The cover art by my dear friend Lina portrays the first programmer Ada Lovelace, who here symbolises all underrepresented people in computer science who have historically paved the way for others and still do, all under the intersectional umbrella – I thank you and I stand with you. Interwoven in the cover art are also symbols that represent my grandparents who all passed away in the course of a year and a half during my time as a doctoral student. I wish to dedicate this thesis to them: Margit, Vera and Åke, I love you and I miss you.

I am eternally grateful to my main supervisor Johanna Björklund and my co-supervisor Frank Drewes, who together form an excellent supervision team – the best there is, I dare to say. Johanna is an incredible writer and an unyielding idea developer, and Frank can formalise anything in the blink of an eye (and is probably the most meticulous proof-reader you can find). Thank you also to Lars Karlsson, who has been my reference person during the research education.

Next, I would like to thank the senior members of our research group (excluding my supervisors whom I have already mentioned): Suna Bensch and Henrik Björklund, who have been informal mentors to me – and for some reason always make me laugh. Moreover, I want to thank the rest of the foundations of language processing research group, in increasing order of seniority: Emil Häglund, Anton Eklund, Iris Mollevik, Hannah Devinney, Arezoo Hatefi, Adam Dahlgren, Petter Ericson and Martin Berglund. An extra thanks to Hannah and Martin who have read and commented on a draft of this thesis, and to Iris who has been my friend for a long time. Adam too deserves an extra mention for running like a champion through an entire airport to get us to the CIAA 2018 conference on time. Similar heroic actions have been performed by the fantastic support personnel at the CS department, and in particular I wish to thank Helena, AnnaKarin, Tomas (a.k.a. stric), Mattias and Anne-Lie. To the remaining colleagues at the department: thank you for sharing knowledge, and for all of the shared laughs. Finally, a special mention to Lena KW who has to be considered the backbone of the department.

Research support does not make up the sole important ingredient for managing a research education. Therefore I wish to thank Thomas and Monica at Feelgood and Frida at Ångestmottagningen for all of the support you have provided me with – I could not have done this without you.

Life is fortunately not only work. My spare time is infinitely brightened by my family and my friends, who have always shown me nothing but love and support. For this, I would like to immensely thank them. Thanks to my mother Carina, who is the funniest and most supportive person I know, my father Lars and my brother Erik. Thanks to my extended family – including my family-in-law. A special thanks to my youngest cousin Cecilia who inspires me with her clever and funny comments and ideas. Thanks to old and new members of my Umeå family: Adam, Cecilia, Frida, Gustav, Lina, Matilda, Martin, Nicklas, Thomas and William – you are the best. Finally, I want to thank the person who has stood by me throughout the thesis work, and whose patience is nearly inexhaustible: my beloved partner Micke.

List of Papers

This thesis is based on the following papers:

Paper I Johanna Björklund, Frank Drewes and Anna Jonsson. Finding the N Best Vertices in an Infinite Weighted Hypergraph. In *Theoretical Computer Science*, volume 682, 30–41, Elsevier, 2017.

Paper II Johanna Björklund, Frank Drewes and Anna Jonsson. A Comparison of Two N-Best Extraction Methods for Weighted Tree Automata. In *23rd International Conference on Implementation and Applications of Automata (CIAA)*, 97–108, Springer, 2018.

Paper III Johanna Björklund, Frank Drewes and Anna Jonsson. Faster Computation of N-Best Lists for Weighted Tree Automata. Submitted to *Theoretical Computer Science* in September 2019 and accepted in March 2021, but not yet published.

Paper IV Johanna Björklund, Frank Drewes and Anna Jonsson. A Comparative Evaluation of the Efficiency of N-Best Algorithms on Language Data. Submitted in March 2021.

Paper V Frank Drewes and Anna Jonsson. Contextual Hyperedge Replacement Grammars for Abstract Meaning Representations. In *13th International Workshop on Tree-Adjoining Grammar and Related Formalisms (TAG+13)*, 102–111, Association for Computational Linguistics, 2017.

Paper VI Johanna Björklund, Frank Drewes and Anna Jonsson. Polynomial Graph Parsing with Non-Structural Reentrancies. To be submitted.

List of Contributions

In the below list, the contributions of the thesis author to the papers are stated. Naturally, the author has – in addition to what is explicitly declared in the list – contributed to all parts of the scientific work.

Paper I	Implementation of the two algorithms as well as designing and performing experiments on the resulting software.
Paper II	Designing, performing, describing and reporting on the results of the experiments.
Paper III	Participating in the development of the algorithm, and proving its time complexity.
Paper IV	Implementing the algorithm and designing, performing, describing and giving the results of the experiments. Describing the competing algorithm and translating our algorithm to the hypergraph setting of the competing algorithm. Taking the lead in the scientific work.
Paper V	Writing of the first draft (which we then developed together).
Paper VI	Providing the first crude idea which was then collaboratively and iteratively refined into the formalism presented.

Contents

1 Introduction **1**
1.1 Outline 5

Part One: The *N*-Best Problem **7**

2 Theoretical Foundation **9**
2.1 Weights and Semirings 9
2.2 Structures and Devices 11
2.3 General Problem Statement 16

3 Related Work **19**
3.1 The Best Runs Problem 20
3.2 The Best Trees Problem 24
3.3 Applications 27

4 Contributions **29**

Part Two: Semantic Graph Grammars **33**

5 Background **35**
5.1 Semantic Graphs 35
5.2 Modelling Semantic Graph Languages 38
5.3 Mapping Between Language Representations 48

6 Contributions **51**

References **53**

Paper I **61**

Paper II **83**

Paper III **97**

Paper IV **117**

Paper V **139**

Paper VI **151**

Chapter 1
Introduction

While computers are strict rule-followers, humans are not – at least not when it comes to our use of language. For example, consider the sentence "They scratch the dog with the stick." Do they scratch the dog using a stick or does the scratched dog have a stick? How should a computer interpret this ambiguous sentence? First, we take a look at how these two interpretations can be represented in a computer. A sentence is a sequence of words, or a *string* of words. As we have seen, a sentence is not in itself a good representation of its own intended interpretation; we need more information. For this, we use analyses in the form of *syntax trees*.

A *tree* is a structure of *nodes* connected by *edges*. Each node is connected to at most one *parent* node and to an arbitrary number of *child* nodes; a node without a parent is a *root*, and a node without children is a *leaf*. In a (constituent) syntax tree for a sentence s, every node has a label; the words of s label the leaves of the tree whereas the remaining node labels are syntactic roles. The edges show how the roles work together. First, each word-labelled node is assigned a parent in the form of a node labelled with the syntactic role of the word: N marks a noun, V marks a verb, P marks a preposition, Det marks a determiner, and so on. These are then combined into larger entities such as a noun phrase (NP), a verb phrase (VP), a prepositional phrase (PP), and finally a complete sentence (S), which is the root of a syntax tree representing an entire sentence.

Here, we depict nodes as their labels and edges connecting two nodes as lines between the nodes. Moreover, the parent of a node n is drawn above n, and the children of n are drawn below n. In Figure 1.1 we see the syntax tree for the interpretation that the stick was used for scratching the dog, and Figure 1.2 portrays the interpretation that the dog has the stick. In our case, the syntax trees represent one interpretation or *semantics* each, but there can be different syntax trees that represent the same semantics (e.g., the syntax trees for "the dog's stick" and "the stick of the dog" are different but mean the same thing).

Now, which interpretation is more likely to be the correct one? If the example sentence is found in a book about someone who is very afraid of dogs, then the first interpretation might be the likeliest, but if the sentence came from an interview with an old person living alone with their dog, then the

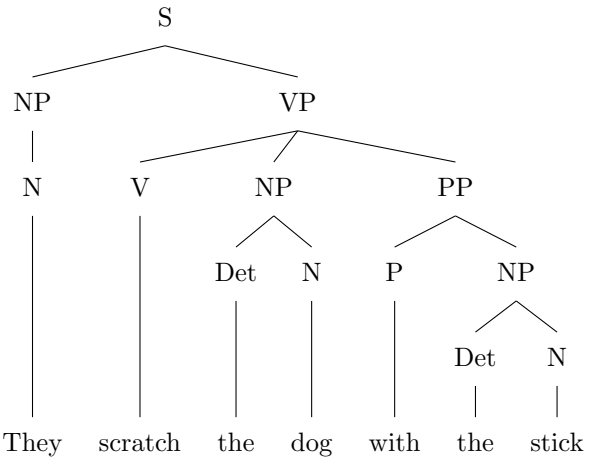


Figure 1.1: Analysis of "They scratch the dog with the stick" representing the interpretation that the stick is used for scratching the dog.

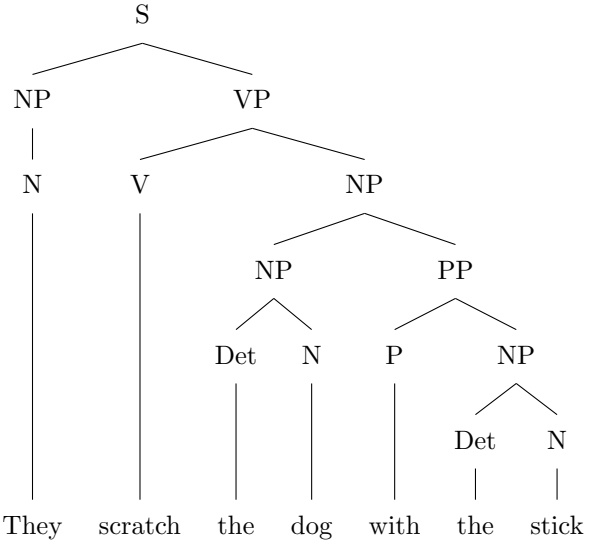


Figure 1.2: Analysis of "They scratch the dog with the stick" representing the interpretation that the scratched dog has a stick.

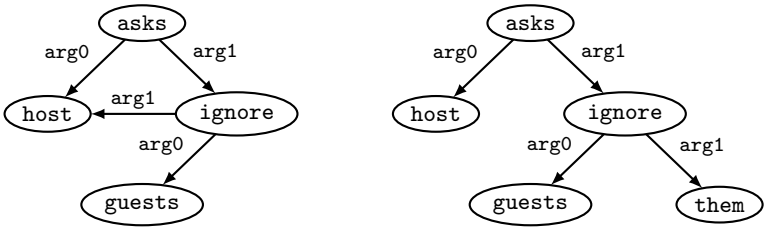
second one is perhaps more likely. Given the context of the sentence, we can learn which interpretation has the highest probability. This is exactly what machine learning is for: teaching a model to value sentence analyses based on the contexts of the sentences. Machine learning of a model reduces to assigning weights to various components of the model. Common models are neural networks and rule-based formalisms; the latter are the focus of this thesis. Doing the actual machine learning in the best possible way is a thesis on its own, which is why we assume that we have already trained models that can tell us the ranking of the various interpretations or analyses.

Rule-based models such as automata and grammars define *formal languages*, i.e., languages that a computer can "speak". In natural language processing (NLP), formal languages are used to approximate natural languages, i.e., languages that humans speak. In this thesis, we are particularly interested in the *weighted tree automata (wta)* formalism. A wta is a system that assigns weights (or ranks) to trees, and it is useful since an analysis of a sentence can, as we have seen, be represented as a tree. Using rule-based models for language processing requires algorithms that solve various problems for these models. Solving *the N-best problem* for wta consists in extracting the N trees of lowest (or highest, depending on our optimisation criterion) weight according to the wta. Thus, N-best trees extraction is useful for returning the top sentence analyses from a wta-based language model.

Every wta defines a potentially infinite *weighted tree language*, which means that a wta can be seen as a compact representation of its weighted tree language since its set of *transition rules* is finite. Given an input tree, a wta applies transition rules to it until the entire tree is processed, and then outputs a weight. The application of transition rules to the tree forms a *run* on the tree. For *deterministic* wta, there is only one run corresponding to every tree. If we instead have *nondeterministic* wta, there are several runs that result in the same tree; in fact, there can be an exponential number of runs relative to the size of the tree. Solving the easier *N-best runs problem* is an alternative solution to extracting the N best trees, but this only guarantees that the output list of N trees is free from duplicates if the input wta is deterministic. In Part One of this thesis, comprising Papers I, II, III and IV, we solve the N-best trees problem for wta and compare it to algorithms that solve the N-best runs problem.

Sometimes we need more than a syntax tree to distinguish between different semantics. For example, the sentence "The host asks the guests to ignore them" says nothing about who "them"[1] refers to; it could be the host who wishes to be ignored, it could be a person or a group of people who are to be ignored (but hopefully not the cockroaches crawling all over the place). Forming a syntax tree does not help us distinguish between interpretations in this case, all we would get from the syntax tree is that "them" is a (pro)noun and how it is linked to the rest of the syntax of the sentence – not more semantic information. This semantic phenomenon of referencing the same concept in different places

[1] Note that "they" and "them" can be used as both singular and plural.



(a) The interpretation that "them" refers to "the host".

(b) The interpretation that 'them' refers to some other person or group of people.

Figure 1.3: Two semantic graph representations, each showing one interpretation of "The host asks the guests to ignore them".

within a single sentence or over sentence borders is called *coreferencing* and demands a more complex representation.

A common solution to this problem is to model the semantics as a *graph*. A graph is a structure consisting of nodes and edges, just as trees but without the constraint that each node can have at most one parent. This implies that we have to indicate the relation between the nodes in some other way than drawing them above or below one another; here, we use directed edges pointing from the parent to the child. Figure 1.3 shows two graphs that each represent one interpretation of the example sentence "The host asks the guests to ignore them". The nodes are depicted as ellipses with their label inscribed, and the directed edges are drawn as arrows. The edges also have labels: `arg0` indicates that the child is the agent of the parent and `arg1` that the child is its patient. To interpret the graph, we start at the root in which we have the word "asks". By following the `arg0` edge, we find the asker, who is the host. Next, we want to know what the host is asking for, so we follow the `arg1` edge and arrive at "ignore". The ones who are asked to ignore are the guests, given by the outgoing `arg0` edge. Now we have arrived at the sole difference between these two graphs. In Figure 1.3(a), the `arg1` edge outgoing from "ignore" arrives at "host", indicating that the host is the one to ignore, but in Figure 1.3(b), "them" is a node of its own and not connected to anything we have already seen in this particular sentence, meaning that "them" is a concept different from what we already have. If we had more data, we could probably say more about to whom "them" actually refers.

Note that a piece of semantics can be expressed in several ways using syntax trees or sentences. For example, the semantics of Figure 1.3(a) can additionally be expressed as the sentence "The host asks to be ignored by the guests", which gives rise to a syntax tree that differs from the one for "The host asks the guests to ignore them". To achieve a semantic representation that is unique for every piece of semantics, we must disregard the syntactic information that is specific to the particular manner in which that piece of semantics is expressed in natural

language. This is exactly what the semantic graphs do: they model concepts (nodes) and their relation (edges), rather than specific natural language objects.

Representing semantics as graphs is not as well-researched as using trees to represent the syntax of natural language. To represent weighted tree languages, we used weighted tree automata. In the graph case, we need a more powerful formalism since we can express more complex dependencies using graphs. In our work, we use *contextual hyperedge replacement grammars (CHRGs)* to describe *semantic graph languages*. To use a CHRG to model a semantic graph language, we must be able to answer the question "Is the graph G in the graph language defined by the CHRG $\mathcal{G}$?" This question is central because we need to be able to decide if a graph is according to our model or not. Answering the question is done by solving the *membership problem*, and we say that an algorithm *parses* G *with respect to* $\mathcal{G}$ if it solves this problem. If both the graph and the grammar are considered as input, the problem is *uniform*, otherwise it is *non-uniform*.

Parsing of CHRGs is intractable in general, meaning that there is no known polynomial-time algorithm that solves their uniform membership problem. As is common when using formalisms for modelling language, adding power and expressibility damages the computability. In our work, we define a restricted version of CHRG that allows for polynomial parsing, and we show that the restricted formalism can express a rather large variety of semantic graphs. Semantic modelling using CHRGs make out Part Two of this thesis, and the papers included in this part are Papers V and VI.

1.1 Outline

Part One of this thesis handles the N-best problem and has the following outline: Chapter 2 establishes the theoretical foundation needed to formally define and discuss the N-best problem, Chapter 3 provides the background to the problem in the form of related work, and Chapter 4 ends Part One by summarising the papers on the N-best problem that are included in this thesis.

The topic of Part Two is semantic modelling with contextual graph grammars and is outlined as follows. Chapter 5 comprises both related work that use rule-based formalisms for semantic modelling and their accompanying formalisms, and Chapter 6 summarises the contributions of this thesis to the field of semantic modelling.

Part One

The *N*-Best Problem

Chapter 2
Theoretical Foundation

The N-best problem consists in extracting the N best hypotheses from a ranked hypothesis space. As we have seen, common hypotheses for language processing tasks take the form of strings or trees – we call the hypothesis type the *structure*. Then, we have the *devices* that define the hypothesis space compactly such as automata and grammars. Finally, there has to be something that ranks the structures, some criterion that decides if a structure is better than another, and this is where weights and semirings enter the picture. Let us now provide formal definitions for each of these components, starting with weights and semirings.

2.1 Weights and Semirings

When developing an algorithm that operates on a weighted structure or device, we have to make sure to choose appropriate weight operations – we must be able to prove correctness for the algorithm in question. A *semiring* is, informally, a set of values (weights) and operations defined on them. Proving correctness of an algorithm often depends on being able to prove that it is compatible with a certain semiring or family of semirings. For example, Dijkstra's greedy algorithm for finding the shortest path in a weighted directed graph [Dij59] assumes weights from $\mathbb{R}_+$ (non-negative reals), and uses the operations '+' and 'min': + combines weights along a path and min picks the smallest-weighted path over all paths in the graph. The min operation makes it possible to find the shortest path without considering all possible paths, and is therefore central to the correctness and efficiency of Dijkstra's algorithm. Another example is given by Mohri [Moh02] who generalises the work of Dijkstra by developing a shortest-path algorithm that works for not only one, but a family of semirings. In short, Mohri's algorithm requires that the semiring used is such that the path weights cannot be decreased by traversing cycles in the input graph, a condition that is trivially fulfilled by the preconditions of applying Dijkstra's algorithm but also works for other combinations of weights and operations. As is common when generalising to a richer domain, Mohri's algorithm is less efficient than Dijkstra's.

Below follows the formal definition of a semiring, which is based on the notion of monoids.

Definition 2.1 A *monoid* is a tuple $(\mathbb{K}, \diamond, e)$ where $\mathbb{K}$ is a set, $\diamond$ is a binary operation, and $e \in \mathbb{K}$ is the *identity element (of $\diamond$)*. Furthermore, the first two conditions in the list below must hold for $a, b, c \in \mathbb{K}$.

1. $(a \diamond b) \diamond c = a \diamond (b \diamond c)$. (Associativity)
2. $a \diamond e = e \diamond a = a$. (Identity)
3. $a \diamond b = b \diamond a$. (Commutativity)

If the third condition also holds, the monoid is *commutative*.

Definition 2.2 A *semiring* is a tuple $\mathcal{K} = (\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ such that the following conditions hold.

- $(\mathbb{K}, \oplus, \mathbb{0})$ is a commutative monoid.
- $(\mathbb{K}, \otimes, \mathbb{1})$ is a monoid.
- $\mathbb{0} \otimes a = a \otimes \mathbb{0} = \mathbb{0}$ for all $a \in \mathbb{K}$; we call $\mathbb{0}$ an *annihilator for* $\otimes$.
- $\otimes$ distributes over $\oplus$, that is, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$ for all $a, b, c \in \mathbb{K}$.

We call $\oplus$ *semiring addition* and $\otimes$ is the *semiring multiplication*. If the monoid $(\mathbb{K}, \otimes, \mathbb{1})$ is commutative, then we say that $\mathcal{K}$ itself is commutative. Another property is *idempotency* which requires that $a \oplus a = a$ for all $a \in \mathbb{K}$. Similarly, a semiring is *extremal* if $a \oplus b \in \{a, b\}$ for all $a, b \in \mathbb{K}$. Note that an extremal semiring is always idempotent whereas an idempotent semiring is not necessarily extremal (e.g., when the weights are vectors over $\mathbb{R}_+$ and $\oplus$ is element-wise min). For an idempotent semiring $\mathcal{K}$, we can define the *natural order $\leq_{\mathcal{K}}$ of $\mathcal{K}$* to be the partial order given by $a \leq_{\mathcal{K}} b \iff a \oplus b = a$ for $a, b \in \mathbb{K}$. A semiring is *finitely generated* if there is a finite subset S of $\mathbb{K}$ whose elements can form all of the elements of $\mathbb{K}$ by applying the operations $\oplus$ and $\otimes$ to the elements in S.

An example of a common semiring is the *Boolean semiring* which is defined as $(\{\text{TRUE}, \text{FALSE}\}, \vee, \wedge, \text{FALSE}, \text{TRUE})$. Since $\text{TRUE} \vee \text{TRUE} = \text{TRUE}$ and $\text{FALSE} \vee \text{FALSE} = \text{FALSE}$, the Boolean semiring is idempotent; it is also extremal since the $\vee$ operation always returns either TRUE or FALSE. Moreover, the commutative operation $\wedge$ is its semiring multiplication, which implies that the Boolean semiring is also commutative. Another common idempotent, extremal and commutative semiring is the *tropical semiring*.

Definition 2.3 The *tropical (or min-plus) semiring* is the semiring $\mathcal{T} = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$.

This semiring is the one used in Dijkstra's algorithm, and it is also used in the N-best trees algorithm that has been developed as part of this thesis work.

2.2 Structures and Devices

Here we define all structures and devices (computational models) that are relevant to the work on the N-best problem covered in this thesis.

Let S be a set, and let $|S|$ denote the cardinality of S; the set operator $\uplus$ denotes the disjoint union of sets. A *string* over S is a finite sequence of symbols from S, and S^* denotes the set of all strings over S; λ represents the empty string. Furthermore, $S^+ = S^* \setminus \{\lambda\}$, and S^n denotes the Cartesian product of S with itself n times. The length of a string s is given by $|s|$. A *ranked alphabet* $\Sigma = \biguplus_{k \in \mathbb{N}} \Sigma_k$ is a finite set of symbols with $\mathrm{rank}(\sigma) = k$ for $\sigma \in \Sigma_k$. A *(ranked) tree* labelled over a ranked alphabet Σ is a string t defined recursively as follows:

- $t = \sigma$ for $\sigma \in \Sigma_0$ is a tree consisting of a single *node* labelled by σ, and
- $t = \sigma[t_1, \ldots, t_k]$ where $\sigma \in \Sigma_k$ and t_i are trees over Σ for $i = 1, \ldots, k$ is a tree. (The square brackets and commas in t are delimiters that are not allowed to be elements of Σ.)

Let v denote the node labelled by σ in t. We say that t is *rooted* in v and that $t_1, \ldots, t_k$ are *(direct) subtrees* of t. Furthermore, the roots of $t_1, \ldots, t_k$ are *children* of v, and v is their *parent*. A *leaf* is a node without children. For convenience, we identify a node by its label when discussing specific trees.

Now that we have defined the structure types for which we are discussing the N-best problem, namely strings and trees, we move on to defining the devices that operate on them, starting with two device types that handle strings.

Definition 2.4 A *context-free (string) grammar (cfg)* is a quadruple $G = (N, \Sigma, S, R)$ where

- N is a finite set of *nonterminals*,
- Σ with $\Sigma \cap N = \emptyset$ is a finite set of symbols or *terminals*,
- $S \in N$ is the *start nonterminal*, and
- R is a finite set of *production rules* on the form $A \to \alpha$ with $A \in N$ and $\alpha \in (N \cup \Sigma)^*$; the substring of nonterminals of α is denoted by $\mathrm{nt}(\alpha)$.

Applying a production rule $r = (A \to \alpha) \in R$ to a string $s_1 A s_2$ is done by replacing A (the *left-hand side* of r) by α (the *right-hand side* of r), yielding $s_1 \alpha s_2$. We write $s_1 A s_2 \Rightarrow s_1 \alpha s_2$ to denote that we applied a rule from R to the first string to derive the latter, and $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$. The language defined by G is $\mathcal{L}(G) = \{s \in \Sigma^* \mid S \Rightarrow^* s\}$, and we say that s can be *derived from S (in G)* or *generated by G*. A derivation in G of a string $s \in \mathcal{L}(G)$ is a tree whose nodes are labelled by production rules from G such that the result of applying the production rules is s, and the root must be labelled with a production rule that has the left-hand side S.

Adding weights to a cfg can be done in various ways. Here, we assign a weight w from a semiring $(\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ to each production rule $r = (A \to \alpha)$ and denote this addition by $r = (A \xrightarrow{w} \alpha)$. To get the weight of a derivation d, we apply $\otimes$ to the weights of all rules in d, and the weight of a string s in $\mathcal{L}(G)$ is computed by applying $\oplus$ to all of the derivations in G that result in s.

Definition 2.5 A *weighted finite string automaton (wsa)* over a semiring $(\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ is a quintuple $M = (\Sigma, Q, \delta, q_0, Q_f)$ such that

- $\star$ Σ is a finite set of symbols, or an *alphabet*,
- $\star$ Q is a finite set of *states*,
- $\star$ $\delta\colon Q \times \Sigma \times Q \to \mathbb{K}$ is the *transition function*,
- $\star$ $q_0 \in Q$ is the *initial state*, and
- $\star$ $Q_f \subseteq Q$ is the set of *final states*.

We write $(q, \sigma) \xrightarrow{w} q'$ to represent the *transition* given by $\delta(q, \sigma, q') = w$. In practice, it is common to only store elements $(q, \sigma) \xrightarrow{w} q'$ for which $w \neq \mathbb{0}$; i.e., transitions that have non-zero weight. Thus, the absence of a transition from the stored part of δ should be interpreted as it having weight $\mathbb{0}$. (Formally, this turns δ into a partial function $\delta\colon Q \times \Sigma \times Q \to \mathbb{K} \setminus \{\mathbb{0}\}$.) For simplicity, we let δ denote the set of transitions given by the domain $Q \times \Sigma \times Q$, including those with zero weight.

Let M be a wsa and let s be an input string $s = \sigma_1 \dots \sigma_n \in \Sigma^*$. A *run* ρ of M on s is a string of states $\rho(s) = q_0 \dots q_n$ where q_0 is the initial state of M, $q_i \in Q$ and $((q_{i-1}, \sigma_i) \xrightarrow{w_i} q_i) \in \delta$ for $i \in \{1, \dots, n\}$. The run is *accepting* if $q_n \in Q_f$. The *weight of* $\rho(s)$ is given by

$$\mathrm{wt}(\rho(s)) = \bigotimes_{i=1}^{n} w_i \ ,$$

and the *weight of* s *assigned by* M is

$$M(s) = \bigoplus_{\rho} (\mathrm{wt}(\rho(s))) \ .$$

That is, we multiply the weights of the used transitions to get the weight of a run and add the weights of all runs on s to get the weight of s in M.

Note that the cfg and the wsa are not equally powerful formalisms: wsa defines weighted *regular* string languages whereas cfgs defines weighted *context-free* string languages. A consequence of this is that the runs are, as we have seen, represented differently: a run of a wsa can be represented with a string whereas we need the more expressive tree structure to represent a derivation of the more powerful cfg.

Next, we define a formal device that defines weighted tree languages.

Definition 2.6 A *weighted tree automaton (wta)* over a semiring $(\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ is a quadruple $M = (Q, \Sigma, \delta, Q_f)$ where

- $\Sigma = \biguplus_{k \in \mathbb{N}} \Sigma_k$ is a ranked alphabet,
- Q with $Q \cap \Sigma = \emptyset$ is the set of *states*,
- $\delta \colon Q^k \times \Sigma \times Q \to \mathbb{K}$ is the *transition function*,
- $Q_f \subseteq Q$ is the set of *final states*.

In analogy with the notation for wsa, we write $\sigma[q_1, \ldots, q_k] \xrightarrow{w} q$ to denote the transition given by $\delta(q_1, \ldots, q_k, \sigma, q) \to w$, and we let δ denote the set of transitions defined for the domain $Q^k \times \Sigma \times Q$.

To make the next definition simpler, we turn δ into a ranked alphabet by setting $\mathrm{rank}(\sigma[q_1, \ldots, q_k] \xrightarrow{w} q) = \mathrm{rank}(\sigma)$.

Definition 2.7 A *run* ρ of M on a tree t labelled over Σ is a relabelling $\rho(t)$ of t with symbols from δ. The *relabelling* $\rho(t)$ *of a tree* t *rooted in a node* v *labelled by* $\sigma \in \Sigma$ is defined recursively as follows.

1. If $t = \sigma$, i.e, v is a leaf node, then v can be relabelled by any $(\sigma \xrightarrow{w} q) \in \delta$, yielding $\rho(t) = (\sigma \xrightarrow{w} q)$.
2. If $t = \sigma[\rho(t_1), \ldots, \rho(t_k)]$ and the root of $\rho(t_i)$ is for $i \in \{1, \ldots, k\}$ labelled by $\sigma_i[q_{i_1}, \ldots, q_{i_{k_i}}] \xrightarrow{w_i} q_i$ where σ_i is the label of t_i, k_i is the rank of σ_i and $q_{i_j} \in Q$ for $j \in \{1, \ldots, k_i\}$, then v can be relabelled by any $(\sigma[q_1, \ldots, q_k] \xrightarrow{w} q) \in \delta$, yielding $\rho(t) = (\sigma[q_1, \ldots, q_k] \xrightarrow{w} q)[\rho(t_1), \ldots, \rho(t_k)]$.

If the right-hand side of the root of ρ is in Q_f, then ρ is an *accepting run* of M on t. An example run of the wta in Figure 2.1 can be seen in Figure 2.2.

The *weight of a run* ρ is denoted by $\mathrm{wt}(\rho)$ and given by the multiplication of the weights of the transition rules labelling ρ. A tree t can have several runs, and we define *the weight of the tree* t *in* M, $M(t)$, as the sum of the weights of all runs of M on t. Under the assumption that the wta in Figure 2.1 is weighted over the tropical semiring $(\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$, the weight of the run in Figure 2.2 is 5, and since $q_f \in Q_f$, it is accepting. The example run is however not minimal: there is another accepting run on the same tree that yields the weight 4, implying that the weight of the tree in Figure 2.2 assigned by the wta in Figure 2.1 is 4.

A wta defines a weighted regular tree language, and wta is thus equivalent to weighted regular tree grammar (wrtg, see e.g. [MK06]). Analogous to how a finite string automaton can be represented as a directed graph in which the nodes encode the states and the transitions are encoded by the edges, both wta and wrtg can be represented as *hypergraphs*.

Definition 2.8 A *weighted hypergraph over a semiring* $(\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ *with edge labels from a ranked alphabet* Σ_E (for short: *hypergraph*) is a quintuple $G = (V, E, \mathrm{att}, \mathrm{lab}_E, \mathrm{wt}_G)$ where:

$$Q = \{q_0, q_1, q_f\},\ Q_f = \{q_f\},\ \Sigma_0 = Q \cup \{a, b\},\ \Sigma_2 = \{f\},\ \Sigma_3 = \{g\},$$
$$\delta = \{a \xrightarrow{0} q_0,\ a \xrightarrow{1} q_1,\ b \xrightarrow{0} q_0,\ b \xrightarrow{1} q_1,\ f[q_1, q_1] \xrightarrow{0} q_f,$$
$$f[q_0, q_0] \xrightarrow{1} q_f,\ f[q_f, q_f] \xrightarrow{1} q_f,\ g[q_0, q_1, q_0] \xrightarrow{1} q_f\}$$

Figure 2.1: Example wta.

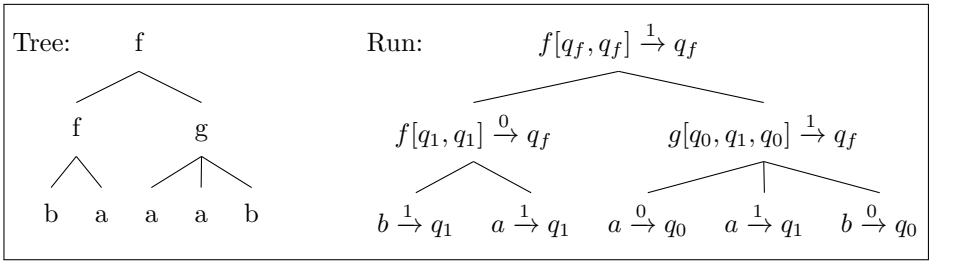


Figure 2.2: Left: A graphic representation of the tree $f[f[b, a], g[a, a, b]]$. Right: An example run for the wta given in Figure 2.1 on the tree to the left.

- ⋆ V is a finite set of *nodes*.
- ⋆ E is a finite set of *hyperedges*.
- ⋆ $\mathrm{att}\colon E \to V^+$ is the *attachment of hyperedges*.
- ⋆ $\mathrm{lab}_E\colon E \to \Sigma_E$ with $\mathrm{rank}(\mathrm{lab}_E(e)) = |\mathrm{att}(e)| - 1$ for all $e \in E$ is the *labelling of hyperedges*.
- ⋆ $\mathrm{wt}_G\colon E \to \mathbb{K}$ is the *assignment of weights to hyperedges*.

Moreover, for $e \in E$ we distinguish the first element of $\mathrm{att}(e)$, denote it $\mathrm{head}(e)$ and call it the *head of* e; the remaining elements comprise the *tail of* e and are denoted by $\mathrm{tail}(e)$. For convenience, we from here on shorten $|\mathrm{tail}(e)| = |\mathrm{att}(e)| - 1$ to $|e|$.

Definition 2.9 A *derivation* of a node v in a hypergraph G is a tree $d = e[d_1, \ldots, d_{|e|}]$ where $e \in E$ with $\mathrm{att}(e) = vv_1 \cdots v_{|e|}$ and d_i is a derivation of v_i for $i \in \{1, \ldots, |e|\}$. Thus, if $|e| = 0$, the derivation consists of a single node, $d = e$. The *weight of* d is defined by

$$\mathrm{wt}(d) = \mathrm{wt}_G(e) \otimes \bigotimes_{i=1}^{|e|} \mathrm{wt}(d_i)\ .$$

To access the node at the top of the derivation $d = e[d_1, \ldots, d_{|e|}]$, i.e., $\mathrm{head}(e)$, without having to explicitly access e, we write $\mathrm{root}(d)$. The tree labelled over

Σ_E that results from a derivation $d = e[d_1, \ldots, d_{|e|}]$ is given by $\text{result}(d) = \text{lab}_E(e)[\text{result}(d_1), \ldots, \text{result}(d_{|e|})]$.

The *weighted tree language defined by a set of nodes* $V_0 \subseteq V$ *in* G is a mapping from trees t for which there exists at least one derivation d such that $\text{result}(d) = t$ and $\text{root}(d) \in V_0$ in G, to the sum of the weights of all such derivations of t in G.

For the purpose of showing that wta and hypergraphs are equivalent devices, we define the construction of a hypergraph given a wta and vice versa.

Definition 2.10 Given a wta $M = (Q, \Sigma, \delta, Q_f)$, we construct a hypergraph $G_M = (V, E, \text{att}, \text{lab}_E, \text{wt}_{G_M})$ in the following way:

- ⋆ for every state $q \in Q$, add a node v_q to V, and
- ⋆ for every transition rule $r = (\sigma[q_1, \ldots, q_k] \xrightarrow{w} q) \in \delta$, add a hyperedge e_r to E with $\text{att}(e_r) = v_q v_{q_1} \ldots v_{q_k}$, $\text{lab}_E(e_r) = \sigma$, $\text{rank}(\sigma) = k$ and $\text{wt}_{G_M}(e_r) = w$.

Definition 2.11 Given a hypergraph $G = (V, E, \text{att}, \text{lab}_E, \text{wt}_G)$, we construct a wta $M_G = (Q, \Sigma, \delta, Q_f)$ as follows:

- ⋆ for every node $v \in V$, add a state q_v to Q, and
- ⋆ for every hyperedge $e \in E$ with $\text{att}(e) = q_v q_{v_1} \ldots q_{v_k}$ and $\text{wt}_G(e) = w$, add a transition rule $r_e = (\text{lab}_E(e)[q_{v_1}, \ldots, q_{v_k}] \xrightarrow{w} q_v)$ to δ.

We define the homomorphism h from trees labelled over δ to trees labelled over E as $h(r[\rho_1, \ldots, \rho_k]) = e_r[h(\rho_1), \ldots, h(\rho_k)]$.

konstigt med avstånden här, eller ?

Theorem 2.12 *Let M be a wta and G_M its corresponding hypergraph. Then, ρ is a run of M on a tree t if and only if $h(\rho)$ is a derivation of G_M with* $\text{wt}(\rho) = \text{wt}(h(\rho))$ *and* $\text{result}(h(\rho)) = t$.

Proof. In the following, let ρ be a run of a wta M on a tree t. If t consists of a single node labelled σ, then ρ is by Definition 2.7 a single node labelled with some transition rule $r = (\sigma \xrightarrow{w} q)$, and applying h to ρ yields $h(\rho) = e_r$. Moreover, e_r is by Definitions 2.9 and 2.10 a derivation of v_q with $\text{wt}(h(\rho)) = w = \text{wt}(\rho)$ and $\text{result}(h(\rho)) = \sigma = t$. Now assume for the sake of induction that Theorem 2.12 holds for runs $\rho_1, \ldots, \rho_k$ on trees $t_1, \ldots, t_k$, respectively. Then, for the run $\rho = r[\rho_1, \ldots, \rho_k]$ with $r = (\sigma[q_1, \ldots, q_k] \xrightarrow{w} q)$ on $t = \sigma[t_1, \ldots, t_k]$, we have $h(\rho) = e_r[h(\rho_1), \ldots, h(\rho_k)]$. Since $h(\rho_1), \ldots, h(\rho_k)$ are derivations of $v_{q_1}, \ldots, v_{q_k}$, respectively, then – again by Definitions 2.9 and 2.10 – $e_r[h(\rho_1), \ldots, h(\rho_k)]$ is a derivation of v_q in G_M with $\text{wt}(e_r[h(\rho_1), \ldots, h(\rho_k)]) = w \otimes \text{wt}(\rho_1) \otimes \cdots \otimes \text{wt}(\rho_k) = \text{wt}(\rho)$ and $\text{result}(e_r[h(\rho_1), \ldots, h(\rho_k)]) = \sigma[\text{result}(h(\rho_1)), \ldots, \text{result}(h(\rho_k))] = \sigma[t_1, \ldots, t_k] = t$.

Conversely, let d be a derivation in a hypergraph G of a node $v \in V$. First, consider the case when $d = e$, i.e., d consists of a single node labelled

by the hyperedge e. By Definitions 2.9 and 2.11, there is a transition rule $r_e = (\mathrm{lab}_E(e) \xrightarrow{w} q_v)$ of M_G with $\mathrm{wt}(d) = \mathrm{wt}_G(e) = w$, and also $\mathrm{att}(e) = v$ and $\mathrm{result}(d) = \mathrm{lab}_E(e)$. Defining a run $\rho = r_e$ and a tree $t = \mathrm{lab}_E(e) = \mathrm{result}(d)$ yields $h(\rho(t)) = e = d$ and $\mathrm{wt}(\rho(t)) = w = \mathrm{wt}(d)$. Again, we make use of induction: For the case when $d = e[d_1, \ldots, d_{|e|}]$, assume that Theorem 2.12 holds for derivations $d_1, \ldots, d_{|e|}$ of nodes $v_1, \ldots, v_{|e|}$ in M_G, respectively. By Definition 2.9, we have $\mathrm{att}(e) = vv_1 \ldots v_{|e|}$, $\mathrm{wt}(d) = \mathrm{wt}_G(e) \otimes \mathrm{wt}(d_1) \otimes \cdots \otimes \mathrm{wt}(d_{|e|})$, and $\mathrm{result}(d) = \mathrm{lab}_E(e)[\mathrm{result}(d_1), \ldots, \mathrm{result}(d_{|e|})]$. By Definition 2.11, there is a rule $r_e = (\mathrm{lab}_E(e)[q_{v_1}, \ldots, q_{v_{|e|}}] \xrightarrow{w} q_v)$ in M_G such that $w = \mathrm{wt}_G(e)$ and q_{v_i} corresponds to v_i for $i \in \{1, \ldots, |e|\}$. According to the induction hypothesis, there are ρ_i and t_i such that $h(\rho_i(t_i)) = d_i$, $\mathrm{wt}(\rho_i(t_i)) = \mathrm{wt}(d_i)$ and $t_i = \mathrm{result}(d_i)$ for $i \in \{1, \ldots, |e|\}$. Defining $\rho = r_e[\rho_1, \ldots, \rho_{|e|}]$ and $t = \mathrm{lab}_E(e)[t_1, \ldots, t_{|e|}] = \mathrm{result}(d)$ yields $h(\rho(t)) = e[h(\rho_1), \ldots, h(\rho_{|e|})] = e[d_1, \ldots, d_{|e|}] = d$ and $\mathrm{wt}(\rho(t)) = w \otimes \mathrm{wt}(\rho_1) \otimes \cdots \otimes \mathrm{wt}(\rho_{|e|}) = \mathrm{wt}_G(e) \otimes \mathrm{wt}(d_1) \otimes \cdots \otimes \mathrm{wt}(d_{|e|}) = \mathrm{wt}(d)$. □

Theorem 2.13 *Let $L_G^{V_0}$ be the weighted tree language of G given by the node set V_0 consisting of all nodes v_q for which the corresponding state q in M_G is in Q_f. Furthermore, let L_{M_G} be the weighted tree language defined by M_G. Then, $L_{M_G} = L_G^{V_0}$.*

Proof. From Theorem 2.12, we know that a run ρ in M_G of weight w has exactly one equivalent derivation d in G – also of weight w. Since V_0 is given by the set of final states in M_G, the derivations ending in nodes of V_0 correspond exactly to the set of final runs of M_G. Thus, the sum of the weights of all derivations d for which $\mathrm{result}(d) = t$ equals to $M_G(t)$. □

2.3 General Problem Statement

For devices that define weighted formal languages, the minimisation version of the *N-best problem* is formulated as:

Definition 2.14 Given a device A and a positive integer N, the *N-best problem* amounts to computing a list of N elements $x_1, \ldots, x_N$ such that

1. all elements in the list are pairwise distinct,
2. the weights assigned by A to the elements in the list are non-decreasing going from position 1 to N, and
3. there is no element x distinct from $x_1, \ldots, x_N$ in the language defined by A with smaller weight than x_N.

The input device can for example be a wta M over the tropical semiring. The elements are then either runs or trees, giving rise to the *N-best runs problem* and the *N-best trees problem*, respectively. (Note that choosing $A = M$ means that we still consider the minimisation version, since minimisation is

the optimisation criterion used for computing $M(t)$ for a tree t.) First, let us consider the case when the elements are runs. We make the observation that a computation of a run of M only uses the semiring's $\otimes$ operator (here $+$). In other words, we would only have to sum up the transition weights of the individual run to compute its weight – the weight of a run is deterministically determined regardless of whether M is deterministic (i.e., has exactly one run per tree). If the elements we want to extract are instead trees, then we must in addition to $\otimes$ use the semiring's $\oplus$ operator to find the weight $M(t)$ of a tree t. This is because of the possibility of M being nondeterministic (i.e., there can be more than one run on t in M). Since $\oplus$ is in this case the min operator, we do not have to consider all of the runs that result in t to find $M(t)$, but we might have to discard duplicate trees that come from runs $\rho(t)$ with $\mathrm{wt}(\rho(t)) \geq M(t)$.

In our above instantiation of the general problem, the $\oplus$ operator of the semiring is extremal, which is why the N-best trees problem is, while not as straightforward as the N-best runs problem, still manageable. In contrast, suppose that $\oplus$ is not extremal; then it does not suffice to discard duplicates. Instead, we have to find all runs ρ on t in M and apply the $\oplus$ operator to all of them to compute $M(t)$.

Here, we instantiated the N-best problem for wta with respect to both best runs and best trees extraction. In Chapter 3 we will see additional instances of the N-best problem.

Chapter 3
Related Work

To start our review of research on the N-best problem, we go back to 1977 when Knuth [Knu77] presented an algorithm that finds the best (smallest) derivation for a weighted context-free grammar (cfg, Definition 2.4). Recall that in order to make a cfg $G = (N, \Sigma, S, R)$ weighted, we assign weights from $\mathbb{R}_+$ to each of its production rules. Knuth, however, takes a more general approach: Every production rule $r = (A \to \alpha)$ is assigned a function f_r. The variables of f_r are given by $\mathrm{nt}(\alpha) = \{x_1, \ldots, x_{|\mathrm{nt}(\alpha)|}\}$, and if $\mathrm{nt}(\alpha) = \emptyset$, f_r is constant-valued. The weight of a derivation of $s \in \mathcal{L}(G)$ is the result of recursively applying the functions for every production rule used in the generation of s. Moreover, f_r has to fulfil the *superiority property*, meaning that f_r is non-decreasing in every variable and that $f_r(x_1, \ldots, x_{|\mathrm{nt}(\alpha)|})$ cannot be smaller than any individual value x_i. In other words: applying a rule cannot decrease the total weight of the derivation. Using a function that sums up all variables and adds a constant representing the production cost for every production rule, as we have chosen to do here, certainly fulfils the superiority criterion. This is naturally less general, but sufficient for the purpose of reviewing the algorithm developed by Knuth.

Algorithm 1 shows the pseudocode for Knuth's algorithm. The algorithm is based on the observation that for nonterminals A that occur in productions $A \xrightarrow{w} \alpha$ with $\mathrm{nt}(\alpha) = \lambda$, it holds that $A \Rightarrow \alpha$. Furthermore, the α that minimises the rule weight w over all production rules $(A \xrightarrow{w} \alpha) \in R$ with $\mathrm{nt}(\alpha) = \lambda$ is optimally derived from its corresponding nonterminal A. Due to the superiority property of the production rules, there cannot be an $\alpha' \neq \alpha$ for which $A \Rightarrow \alpha'$ with smaller weight than the minimal w. Thus, w can be stored as a result in the array $res[A]$ (given that the nonterminals can be internally represented as integers, we allow that arrays are indexed using nonterminals). To mark that the weight of A's best derivation has now been determined, A is added to the formerly empty set D. Now knowing the best derivation for the nonterminal in D, we are no longer restricted to rules that have only terminals in their right-hand sides, but these can now also contain the nonterminal in D since its smallest weight is already determined. This allows us to pick the smallest-weighted derivation for a new nonterminal in the same way as we did before. We do this iteratively and when D contains all nonterminals, we are done, and we return res containing the result.

Algorithm 1 An algorithm that given a weighted cfg $G = (N, \Sigma, S, R)$ with the superiority property produces a string of smallest weight in $\mathcal{L}(G)$.

```
1:  D ← ∅                                              ▷ defined nonterminals
2:  K ← QUEUE-EMPTY()                                  ▷ create priority queue
3:  res ← [ ]                                          ▷ array for storing the results
4:  for A ∈ N do
5:      A.weight ← ∞
6:      QUEUE-ENQUEUE(K, A, A.weight)
7:  end for
8:  while |D| < |N| do
9:      for A ∈ N ∩ D do
10:         m ← min{w + Σ_{B∈N} res[B] | (A →^w α) ∈ R, nt(α) ⊆ D}
11:         if m < A.weight then
12:             A.weight ← m
13:             QUEUE-INCREASE-PRIORITY(K, A, A.weight)
14:         end if
15:     end for
16:     A ← QUEUE-DEQUEUE(K)
17:     res[A] ← A.weight
18:     D ← D ∪ {A}
19: end while
```

Note that since the operations used are min and +, and the weights are from $\mathbb{R}_+$, the algorithm specified here uses the tropical semiring (Definition 2.3). The time complexity of Knuth's algorithm is $O(|R| \log |N| + t)$ where t is the total length of all elements in R.

In practice we often want to know not only the smallest weights, but also the actual derivations, which we can keep track of during the computation of the weights. The 1-best derivation is then the derivation corresponding to $res[S]$ (which also yields the best string in $\mathcal{L}(G)$).

In the rest of this chapter, we summarise papers that solve the N-best problem for $N > 1$, and these are divided into two groups: those that find the N best runs (or derivations) (Section 3.1) and those that find the N best strings or trees (Section 3.2). Finally, we review applications of the N-best trees problem, and related problems (Section 3.3).

3.1 The Best Runs Problem

Huang and Chiang [HC05] present an algorithm that solves the N-best derivations problem for a finite hypergraph (see Definition 2.8). As shown in Theorem 2.13, this is equivalent to solving the N-best runs problem for wta.

In the original paper, each hyperedge has its own weight function that takes the weights of the tail's nodes to a weight that is then assigned the head

node. Moreover, those weight functions have to be monotonic in the sense that increasing one of the tail weights does not decrease the head's weight. (This is similar to the superiority property of [Knu77], but Huang and Chiang call it monotonic since they formulate it as a property of the hypergraph.) Using an idempotent and commutative semiring such as the tropical semiring is therefore less general in what weight relations we can have, but fulfils the requirements and lets us compare this algorithm to similar ones in a clearer manner.

The input to Huang and Chiang's algorithm is a triple (G, v_0, N) where $G = (V, E, \mathrm{att}, \mathrm{lab}_E, \mathrm{wt}_G)$ is a hypergraph, $v_0 \in V$ is the *target node*, and $N \in \mathbb{N}^\infty$. The goal is to solve the N-best problem (see Definition 2.14) to extract derivations for the node v_0 in G.

The algorithm builds on the idea that if we have the 1-best derivation, then we can build the 2-best derivation from that using the information given by the hyperedge connections. Generally: the N-best derivation can be built from the N'-best derivation for some $N' < N$. The best derivations for a node v are saved in a list $res[v]$, and *cands* is an array of heaps; $cands[v]$ contains the candidates for the next best derivation for v. New candidates are built based on the derivations in res by exchanging each direct subderivation d' (which is thus in $res[\mathrm{root}(d)]$) with the derivation following d' in $res[\mathrm{root}(d)]$. If an element in res that is not yet present is needed for building new candidate derivations, the algorithm computes it recursively. The best candidate for v_0 is then picked by extracting the minimal element from the heap $cands[v_0]$, which unlocks new candidates.

To facilitate the algorithm presentation, the operation $\mathrm{index}(d)$ on a derivation d gives us the index of d in $res[\mathrm{root}(d)]$. If d is not in $res[\mathrm{root}(d)]$, $\mathrm{index}(d)$ is undefined; therefore, we use $\mathrm{index}(d)$ exclusively on elements that we know are in $res[\mathrm{root}(d)]$.

The algorithm in its entirety is outlined in Algorithm 2. Line 3 can be performed in $O(|E|)$ time using the Viterbi algorithm (see [HC05] for a short description), if we consider the rank of the alphabet used a constant. The time complexity of the algorithm is in total $O(|E| + d_{\max} N \log N)$ where $d_{\max}$ denotes the length of the longest derivation out of the N results.

A factor that limits the Huang and Chiang algorithm is that the Viterbi algorithm requires the input hypergraph to be acyclic. However, Huang and Chiang conjecture that their algorithm excluding the Viterbi procedure works for cyclic hypergraphs – a conjecture that has since been proven to be correct, as we shall now see.

Büchse et al. [BGSV10] generalise the Huang and Chiang algorithm by

1. using labelled hypergraphs,
2. exploiting the call-by-need lazy evaluation in the functional programming language Haskell,
3. allowing for structured weight domains, and
4. explicitly handling cyclic hypergraphs.

Algorithm 2 Compute $N \in \mathbb{N}^\infty$ derivations rooted at $v_0 \in V = \{1, \ldots, |V|\}$ of minimal weight according to a hypergraph $G = (V, E, \mathrm{att}, \mathrm{lab}_E, \mathrm{wt}_G)$.

```
 1: procedure HuangChiang(G, v_0, N)
 2:     res[v] ← List-Empty() for all v ∈ V
 3:     List-Add(res[v], the 1-best derivation of v in G) for all v ∈ V
 4:     BestDerivations(v_0, N)
 5:     for d ∈ res[v_0] do
 6:         Output(d)
 7:     end for
 8: end procedure
 9:
10: procedure BestDerivations(v, c)
11:     if |res[v]| ≥ c then
12:         return
13:     end if
14:     if cands[v] is undefined then
15:         l ← Sort({e[d_1, ..., d_|e|] | head(e) = v;  d_i are 1-best runs; i = 1, ..., |e|})
16:         cands[v] ← Heap-Ify(l[1] ... l[N])
17:         List-Add(res[v], Heap-Extract-Min(cands[v]))
18:     end if
19:     while List-Size(res[v]) < c and Heap-Size(cands[v]) > 0 do
20:         s ← List-Size(res[v])
21:         d ← List-Get(res[v], s)                    ▷ the s-best derivation of v
22:         NextBest(cands[v], d)                      ▷ build new candidates from d
23:         List-Add(res[v], Heap-Extract-Min(cands[v]))
24:     end while
25: end procedure
26:
27: procedure NextBest(vcands, d = e[d_1, ..., d_|e|])
28:     for i ← 1, ..., |e| do
29:         list ← res[root(d_i)]
30:         c' ← index(d_i) + 1
31:         BestDerivations(root(d_i), c')      ▷ make sure list has c' ≤ N elements
32:         if c' ≤ List-Size(list) then
33:             d'_i ← List-Get(list, c')
34:             d' ← e[d_1, ..., d'_i, ..., d_|e|]
35:             if d' ∉ vcands then
36:                 Heap-Insert(vcands, d')
37:             end if
38:         end if
39:     end for
40: end procedure
```

We present the reasoning behind each of the generalisations, but leave out an in-depth presentation of the Büchse et al. algorithm since it is very similar to the one by Huang and Chiang.

The first generalisation is the labelling of the hypergraphs which is motivated by the need to represent tree-based language models such as weighted tree automata. This generalisation simply adds labels to the hyperedges, and does not require any changes to the algorithm itself. Similarly, the use of the programming language Haskell and its lazy evaluation scheme (the second generalisation) does not change the algorithm other than expressing it in a functional programming setting – the Huang and Chiang algorithm is already lazy by only doing recursive calls to construct a derivation when ready to output it.

Third, there is the generalisation to structured weight domains such as vectors over $\mathbb{R}_+$. This is done by requiring a linear pre-order $\precsim$ on the weights, and by extension, on the derivations. Let $c = e[c_1, \ldots, c_{|e|}]$ and $d = e[d_1, \ldots, d_{|e|}]$ be any two derivations with identical roots. A *linear pre-order* $\precsim$ is a binary relation that is reflexive, transitive and linear, the latter meaning that if not $c \precsim d$ (c is better than d), then $d \precsim c$ (d is better than c). Two more properties have to be fulfilled by $\precsim$:

$$d_i \precsim d \text{ for } i \in \{1, \ldots, |e|\} \tag{3.1}$$

$$c \precsim d \text{ if } c_i \precsim d_i \text{ for } i \in \{1, \ldots, |e|\} \tag{3.2}$$

To compare this approach with the one of Huang and Chiang, recall that Huang and Chiang require that each of the weight functions that are connected to the hyperedges is monotonic in every parameter; this fulfils the criteria for applying Knuth's algorithm for computing the 1-best derivation, and corresponds to properties 3.1 and 3.2 above. The resulting derivation weights are then compared using a total ordering on $\mathbb{R}_+$. The Büchse et al. approach is thus a generalisation in that they require only reflexivity and transitivity (not antisymmetry) which ensures that structured weight domains can be used.

Finally we have the explicit handling of cycles in the hypergraph. As previously mentioned, Huang and Chiang conjecture that their second phase (i.e., the part that exploits the pre-computed 1-best runs to compute the N best runs) works on cyclic hypergraphs. Büchse et al. prove this claim. Even before the proof of Büchse et al., this claim was experimentally supported by the fact that the wta toolkit Tiburon [MK06] implements the Huang and Chiang algorithm but with the Viterbi algorithm exchanged for Knuth's algorithm – which works for cyclic graphs.

Since the first three generalisations do not alter the algorithm by Huang and Chiang, the main contribution of Büchse et al. is showing that cycles can in fact be handled by Huang and Chiang, given that we use Knuth's algorithm to compute the 1-best derivations, and that more general weights can be used.

3.2 The Best Trees Problem

Mohri and Riley [MR02] consider the N-best problem (Definition 2.14) for string extraction from weighted string automata (wsa, Definition 2.5) over the tropical semiring. Since strings are monadic trees, this is a special case of the N-best trees problem.

The algorithm of Mohri and Riley is based on an on-the-fly determinisation of $M = (\Sigma, Q, \delta, q_0, Q_f)$, i.e., they create the equivalent wsa $M' = (\Sigma, Q', \delta', q'_0, Q'_f)$ with a single run per input string such that $M'(s) = M(s)$ for all $s \in \Sigma^*$. Not all weighted automata are determinisable (i.e., their determinisation create infinite automata), but since only the portion of M' that is used in the N best computation needs to be created, this does not pose a problem here.

The determinisation of M creating M' is done as follows: The initial state of M' is defined by $q'_0 = \{(q_0, 0)\}$ where 0 is a remainder weight. Next, we iteratively create new transitions $(q'_a, \sigma) \xrightarrow{w} q'_b$ where

- $q'_a = \{(q_i, r_i) \mid 0 \leq i \leq n\}$ is the current state,
- $w' = \min\{r_i + w \mid 0 \leq i \leq n,\ (q_i, \sigma) \xrightarrow{w} q' \in \delta\}$ is the transition weight,
- and $q'_b = \{(q', r') \mid 0 \leq i \leq n,\ (q_i, \sigma) \xrightarrow{w} q' \in \delta,\ r' = \min(r_i + w - w')\}$ is the next state,

and add them to δ'. If q'_b was previously unseen, we add it to Q', and also to Q'_f if any $q' \in q'_b$ is in Q_f.

The pseudocode of the algorithm (in which the on-the-fly determinisation is implicit) can be seen in Algorithm 3. K is a priority queue that holds the partial runs that have already been explored. The format of the runs is (q, s, c) where q is the current state, s is the string that has been processed this far, and c is the cost of the partial run. A list L is used for storing output entries; these have the same format as the partial runs with the difference that the state is final. While K is not empty, the algorithm dequeues an element, checks if the state is final, and if so, adds that run to the output. Moreover, it expands the search space by adding one new partial run to K for each (deterministic) next step that can be taken by applying a rule in T'. When N accepting runs have been found, L is outputted, and because of the determinisation, the outputted runs correspond to distinct trees.

The trick that makes this algorithm extra efficient is hidden in the priority queue K. The priority is not only decided by the weight of the partial run, but also by the weight of the smallest-weighted partial run that takes us from the current state to a final state. In other words, the partial runs in K are sorted according to the sum of the weights of the current partial run ending in q and another partial run that takes us from q to a final state in the least weight. The computation of the latter weight can be done as a preprocessing step on M, and then those weights are translated to be expressed in terms of M' during the run of the algorithm.

Algorithm 3 An algorithm that given a wsa M weighted over the tropical semiring produces N distinct strings of smallest weight.

```
 1: L ← List-Empty()
 2: K ← Queue-Empty()
 3: Queue-Enqueue(K, (q'_0, λ, 0))
 4: while ¬Queue-Is-Empty(K) do
 5:     (q, w, c) ← Queue-Dequeue(K)
 6:     if q ∈ Q'_f then
 7:         List-Add(L, (q, w, c))
 8:         if List-Size(L) = N then
 9:             return L
10:         end if
11:     end if
12:     if List-Size(L) ≤ N then
13:         for (q, σ) -t-> q' ∈ δ' do
14:             c' ← c + t
15:             Queue-Enqueue(K, (q', wσ, c'))
16:         end for
17:     end if
18: end while
```

Björklund et al. [BDZ15] solve the N-best trees problem for tree automata that are weighted over the tropical semiring (wta, Definition 2.6). Their algorithm is an extension of the N-best strings algorithm of Mohri and Riley to the tree domain. We from here on refer to this algorithm as Best Trees v.1.

To facilitate the description of the algorithm, the input wta $M = (Q, \Sigma, \delta, Q_f)$ is split into two wtas given a state $q \in Q$: $M^q = (Q, \Sigma, \delta, \{q\})$ which defines all runs of M that end in q, and $M_q = (Q, \Sigma \uplus \{\Box\}, \delta \uplus \{\Box \xrightarrow{0} q\}, Q_f)$ which defines all runs of M starting at q and ending at a final state of M. The symbol $\Box$ of rank 0 is merely a placeholder for some subtree given by M^q. A tree labelled over $\Sigma \cup \{\Box\}$ with exactly one occurrence of $\Box$ is a *context*. Thus, finding a tree t that minimises $M(t)$ is now a question about finding a tree t and a context c that together minimise $M_q(c) + M^q(t)$ over all $q \in Q$. The *best context* for a state q is a context c_q that minimises $M_q(c)$ over all contexts c. Since c_q only depends on the input wta, c_q can be pre-computed for all $q \in Q$ using Knuth's algorithm. What remains is minimising $M_q(c_q) + M^q(t)$ for some t and q, which can be done using dynamic programming, as we shall see.

Consider the pseudocode in Algorithm 4. The algorithm uses a set T of trees that have been processed by the algorithm and a priority queue K that contains trees that are (potentially) unseen. On line 5, K is initialised with the trees σ for which there are rules $\sigma \xrightarrow{w} q$ (i.e., rules of rank 0) in M. The procedure Prune takes T and K and makes sure that the total number of trees that can reach q does not exceed N in those structures for all $q \in Q$.

If the number is exceeded by c, then the c trees with the highest weights are removed from K. The main loop is entered at line 7. In each iteration, a tree t is dequeued from K and considered for output. Then, the search space is expanded by exploring the trees that can now be built using t. For this, the algorithm by Eppstein [Epp98] for finding the N shortest paths in a graph is used (for more details, see [BDZ15]). Then, the trees found by Eppstein's algorithm are inserted into K, which is in its turn constantly kept pruned. When N trees have been output or K is empty, the algorithm terminates.

Algorithm 4 An algorithm that given a wta M weighted over the tropical semiring produces N distinct trees of smallest weight.

1: $T \leftarrow \emptyset$ ▷ seen trees
2: $K \leftarrow$ QUEUE-EMPTY() ▷ priority queue
3: $counter \leftarrow 0$ ▷ counter for outputted trees
4: **for** $\sigma \in \Sigma_0$ **do** ▷ initialisation
5: PRUNE(T, QUEUE-ENQUEUE($K, \sigma, \min_{q \in Q}\{w + M^q(c_q) \mid (\sigma \xrightarrow{w} q) \in R\}$))
6: **end for**
7: **while** $counter < N \wedge \neg$QUEUE-IS-EMPTY(K) **do**
8: $t \leftarrow$ QUEUE-DEQUEUE(K)
9: $T \leftarrow T \cup \{t\}$
10: **if** $M(t) = \min_{q \in Q}\{M_q(t) + M^q(c_q)\}$ **then**
11: OUTPUT(t)
12: $counter \leftarrow counter + 1$
13: **end if**
14: **for** $r = (\sigma[q_1, \ldots, q_k] \xrightarrow{w} q)$ **do**
15: $E \leftarrow$ EPPSTEIN(r, t, N) ▷ the N best instantiations of r that use t
16: **for** $t' \in E$ **do**
17: $M(t') \leftarrow M_q(t') + M^q(c_q)$ ▷ note that q is fixed by r
18: PRUNE(T, QUEUE-ENQUEUE($K, t', M(t')$))
19: **end for**
20: **end for**
21: **end while**

As mentioned, wtas and the finite hypergraphs used by Huang and Chiang are interchangeable representations of problem instances. Thus, Björklund et al. solve the N-best trees problem for wtas whereas Huang and Chiang solve the N-best *runs* problem for wtas. If the input wta is deterministic, the two problems essentially coincide. Otherwise, there can be up to an exponential number of derivations for one tree (measured in the size of the tree), which illustrates why the N-best trees problem is potentially a more difficult one.

3.3 Applications

To motivate the research on the N-best trees problem, we review applications in which N-best lists are extracted. All of the applications mentioned here use algorithms that extract (the trees corresponding to) the N best runs instead of the trees themselves directly. This can lead to duplicate trees in the N-best list, unless the input device is deterministic. Since duplicates do not contribute with more information, they do not contribute to increased result quality. Instead extracting the best trees would guarantee that the resulting N-best list is free from duplicates and thereby contains more information.

Socher et al. [SBMN13] introduce the compositional vector grammar (cvg), which is built on top of a standard probabilistic context-free grammar (pcfg) model which is then combined with a recursive neural network (RNN). First, the pcfg is trained on the input data, then the top 200 derivations are extracted and input into the RNN which then outputs the final result. This is a type of re-ranking of the trees, with the exception that the output tree of the RNN can be distinct from any of the input trees. Training a pcfg or a similar model and then extracting the top derivations is a common way of propagating partial results to the next step of the language processing pipeline, and Socher et al. show that it is useful to start out with a basic formal method even in a deep learning setting.

Zhao et al. [ZZT18] generalise the cvgs of Socher et al.; they use grammars in which each nonterminal has sub-nonterminals that are given by points in a vector space, creating latent vector grammars (lvegs). Thus, there is no upper limit as to how many sub-nonterminals can be modelled. The rules are however formulated in terms of nonterminals and not their subtypes. Each rule of the grammar has a weight function which assigns a weight to all possible combinations of subtypes of the nonterminals involved. As a part of their pipeline, they extract the 200 best derivations and propagate them, just as Socher et al. do. Zhao et al. find that while cvgs do not allow for dynamic programming to be used for inference, it can be done for other classes of lvegs such as the in the same paper proposed Gaussian mixture lvegs. (Where dynamic programming is not applicable, one has to resort to greedy or beam search, which only provides approximate results.)

Knight and Graehl [KG05] note that despite the popularity of probabilistic finite-state string transducers (fsts) in NLP, fsts are not always suitable for NLP tasks due to their limitation to strings. Therefore, Knight and Graehl investigate whether it would be possible to instead use probabilistic *tree* transducers to allow for tree representations of language. The study consists in looking at existing string-based methods and exploring the literature to find replacement tree transducer types and algorithms that work for the tree case.

Pipelines for NLP tasks often consist of a number of fsts and end in a finite-state string acceptor (fsa) that represents and ranks the output strings. Knight and Graehl consider several possible replacements for fsts in the form of different kinds of tree transducers. Their conclusion is that there are in

fact replacement tree transducers that could be employed for every use case considered. Next, probabilistic regular tree grammars (prtgs) are mentioned as suitable replacements for fsas. In connection with this, Knight and Graehl write that one of the bottlenecks for replacing string formalisms by tree ones is the lack of an efficient algorithm for best trees extraction for prtgs. Since prtgs correspond to wta, this is a request for the exact N-best trees algorithm presented in this thesis.

Finkel et al. [FMN06] develop an alternative method for finding good samples to propagate in NLP pipelines. Instead of extracting the N best runs, they model the entire pipeline as a Bayesian network and consider every step as a variable. This allows them to take alternative labels from each prior step into consideration when choosing what labels to propagate in the current step. They claim this model to be both simpler to implement and faster than N best runs extraction. The motivation of Finkel et al. for developing this new method is that the N-best lists algorithms that are available are insufficient since they do not extract N distinct hypotheses but N distinct runs. To support this motivation, they run the Stanford parser, extract the $N = 50$ best runs and observe that about half of the output trees are actually duplicates – enough to affect the outcome of the language model pipeline. Thus, extracting the best trees instead of the best runs is not only theoretically better, but also of practical significance.

Chapter 4
Contributions

This chapter outlines the contributions of the papers in this thesis and puts them in the context given by the related work of Chapter 3.

Paper I: Finding the N Best Vertices in an Infinite Weighted Hypergraph

Paper I presents a generalised and abstract algorithm for N-best extraction. The device is a hypergraph (see Definition 2.8) that is acyclic and possibly infinite, and the goal is to compute N vertices (or nodes) that are optimal with respect to a *nice* semiring. A nice semiring is a semiring that is finitely generated, idempotent and that has $\mathbb{1}$ as its smallest element. The generalisation is done with respect to the Best Trees v.1 algorithm (Björklund et al. [BDZ15]) which only covers wta over the tropical semiring: this algorithm covers all devices whose computations can be modelled as a hypergraph, and the algorithm is proven to work for a family of semirings instead of a single one. The abstraction consists in having to implement a number of unimplemented functions that depend on the input of the instantiating algorithm.

By considering the hypergraph a computation graph of a wta, we show that the Best Trees v.1 algorithm is an instantiation of the hypergraph version. An obstacle is that the tropical semiring used in Best Trees v.1 is not finitely generated. However, considering only the weights that are actually used in the computation makes it finitely generated from the set of transition weights.

The paper also contains an experimental evaluation of the Best Trees v.1 algorithm; Best Trees v.1 is compared to itself but with the pruning turned off. The purpose of this evaluation is to find out how much the pruning improves the algorithm's performance in practice, and the result was according to the theoretical expectation, although slow in absolute numbers.

Paper II: A Comparison of Two N-Best Extraction Methods for Weighted Tree Automata

In Paper II, we compare Best Trees v.1 with the state-of-the-art algorithm of Huang and Chiang [HC05] for extracting the best runs. The latter is implemented in Tiburon [MK06] – a toolkit for tree automata. In preparation for the comparison, the implementation of Best Trees v.1 was improved. The comparison was done by using Tiburon to extract as many runs as needed to find N distinct trees and comparing its running time to the running time of Best Trees v.1 for finding N trees (file handling excluded). The wta files used were artificially created to model increasing amounts of nondeterminism, i.e., the wta languages had increasingly many duplicates. As expected, the conclusion is that Tiburon is faster on wtas with small amounts of nondeterminism, whereas Best Trees v.1 is better when there is more nondeterminism.

Paper III: Faster Computation of N-Best Lists for Weighted Tree Automata

In Paper III, we develop a new and more efficient version of Best Trees v.1, which we call Best Trees. The two versions both use a priority queue of trees that prioritises a tree t that is in the state q based on the sum of t's current weight at q and the best context weight of q (see Section 3.2). However, Best Trees additionally uses a second level of priority queues, one for each transition rule of the input wta, which allows us to prune the tree candidates with respect to the transitions. The second-level priority queues are in fact elements in the main priority queue, and they are prioritised based on their top element – the currently highest-prioritised tree for the corresponding rule. New tree candidates are created as in the Huang and Chiang algorithm [HC05]: For every tree t that is dequeued from the main queue via the rule queue corresponding to the rule $r = (\sigma[q_1, \ldots, q_k] \xrightarrow{w} q)$, we successively exchange each direct subtree t' of t that thus is in some state q_i for $i \in \{1, \ldots, k\}$ by the next best tree for q_i, yielding as many candidates as the rank of r, namely k. (We might have seen some of the candidates before for the rule queue of r, and in that case, the already seen candidates are disregarded.) When a tree t is dequeued from a rule queue, only that particular rule queue has to be updated with new candidates. If a candidate cannot be instantiated immediately since at least one of its direct subtrees has not been dequeued from the main queue yet, the candidate is marked as pending. For the rest of the rule queues, we at most have to instantiate the pending candidate trees that t has now made available and add them to their corresponding rule queues.

The time complexity of Best Trees is $O\big(Nm(\log m + r^2 + r\log(Nr))\big)$ where m is the number of transition rules in the input wta and r is the maximum rank over all of its transitions.

Best Trees is implemented as Betty[1] (named after Betty Holberton who was a mathematician and one of the famous six ENIAC programmers). Betty can solve not only the N-best trees problem but also the N-best runs problem by ignoring the equality tests made by Best Trees on trees before saving them for use in further computation.

Paper IV: A Comparative Evaluation of the Efficiency of N-Best Algorithms on Language Data

Paper IV consists of an experimental evaluation of Best Trees in relation to both the previous version Best Trees v.1 and the Huang and Chiang algorithm [HC05] in the form of Tiburon [MK06] – just as for Paper II. However, this time we do not only use artificial data for the sake of investigating the effect of nondeterminism, but first and foremost wta that represent natural language data. We use Betty to solve both the best trees and the best runs problem, and Tiburon to solve the best runs problem. Betty outperforms Tiburon for the best runs task on all 2 269 input wtas except one. The exception is a controlled wta for Latin with more than a million transitions which produces many very small trees with low weights. This shows how extreme a situation we need to provoke the case where the computation of the best contexts actually creates a significant overhead. Furthermore, using Betty for solving the best trees task yields only slightly worse running times than using Betty for the best runs task. The exception here is the artificial corpus with exponentially many duplicates for each tree (in the size of the tree): it slows Betty down significantly in absolute numbers when used for best trees extraction. As a result, we had to shrink the test domain for this particular experiment. Fortunately, the asymptotic behaviour displayed by Betty for this test case is well-aligned with the asymptotic analysis of Best Trees.

Moreover, we evaluate the theoretical differences between the algorithms in more detail. The time complexity of the Huang and Chiang algorithm is $O(m + d_{\max} N \log N)$ where $d_{\max}$ is the size of the largest outputted run. In their time complexity proof, they consider the rank a constant. Doing the same for Best Trees yields a time complexity of $O(Nm(\log m + \log N)) = O(Nm \log m + Nm \log N)$. The upper bounds of the two algorithms only differ by a factor $\max\{N \log m, m\}$, which is surprising considering that Best Trees solves a problem that can potentially require pruning away an exponential number of duplicate runs that are all valid outputs for the best runs task.

Previously, we reported that the running time of Betty is relatively large in absolute numbers on a corpus displaying exponential nondeterminism; we explain this phenomenon as follows. Since the size of a tree is the exponent that decides the number of duplicate runs for the tree, there are more duplicates for larger trees than smaller ones. This together with the fact that Best Trees builds larger trees from smaller ones to find more candidate output trees yields

[1] `https://github.com/tm11ajn/betty`

that the larger the N, the larger the difference between the size of the Nth tree and the Nth run. This in turn means that the size of the output is in total much larger for best trees extraction than for best runs extraction on this corpus. Thus, best trees extraction requires more memory and thereby causes a quicker exhaustion of the computer's resources.

We argue that the success of Best Trees is due to both the best context weights and the two levels of priority queues. This is supported by a diagnostic test where we removed the best contexts procedure. The result was that the number of candidates in the second-level priority queues was increased by roughly a factor 10. The exact effect of the various parts would however have to be investigated more thoroughly to be entirely certain.

Part Two

Semantic Graph Grammars

Märkligt på flera sätt. Vad är ett semantiskt grafspråk? Sedan "can be used to model" – kanske bättre "to describe" eller "to specify" eller "as a finite description of"?

Chapter 5

Background

In this chapter, we review central work on rule-based formalisms, such as graph grammars, that can be used to model semantic graph languages. We begin the review by introducing and summarising work on semantic graphs. Then we proceed to review formalisms (mainly grammars) that can represent semantic graph languages. First, let us recall some basic notation from Chapter 2.

For a set S, the cardinality of S is denoted by $|S|$, and for two sets S and S', $S \uplus S'$ denotes the disjoint union. A *string over* S is a finite sequence of symbols from S, λ is the empty string, and S^* is the set of all strings over S. An *alphabet* Σ is a finite set of symbols, and Σ is *ranked* if every symbol $\sigma \in \Sigma$ is assigned a rank by $\text{rank}(\sigma)$. Previously, we saw that trees can be ranked, meaning that a symbol σ can only label a node v if the number of children of v is $\text{rank}(\sigma)$. In a similar manner, graphs can also be ranked, and in the following, we will see formalisms using both ranked and unranked graphs.

5.1 Semantic Graphs

may!

The Penman language system [Pen89] is a large natural language generation system that handles hundreds of semantic features (e.g., if a sentence is in active or passive form). However, the subset of features that is actually needed depends on the application. This means that a programmer using Penman in their application might have to set all of the features when ten of them suffice for the application at hand. To make the use of Penman more practical, Kasper [Kas89] develop the *sentence plan language (SPL)* that can be used for describing sentences at different abstraction levels with a varying amount of details. Entering a sentence plan into the Penman system allows the system to extract the available information without the programmer having to specify all of the semantic features. The syntax of SPL is expressed using production rules on Backus–Naur form (see [Kas89] for these production rules). An example sentence plan that represents the sentence "The Tudor was en route home to Skelleftehamn" can be seen in Figure 5.1. The `/` operator defines a variable: for example, `e` is a variable that represents an instance of the concept `en route`. Moreover, the `:` operator denotes an attribute of the variable. For example, `e` has an `actor` named `s` which is a `ship` whose `name` is `Tudor` (`name` is an

```
(e / en route
    :destination.r (p / port :name Skelleftehamn)
    :actor (s / ship
              :name Tudor
              :home p)
    :tense past)
```

Figure 5.1: Sentence plan for "The Tudor was en route home to Skelleftehamn."

attribute of `s`). Note that the variable `p` occurs twice: as the `destination` and as `Tudor`'s `home`. Sentence plans are in fact directed graphs in which the concepts are node labels and the attributes are edge labels.

Definition 5.1 A *(node- and edge-labelled directed) graph* over an alphabet $\Sigma = \Sigma_V \uplus \Sigma_E$ is a tuple $G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab}_V, \mathrm{lab}_E)$ where

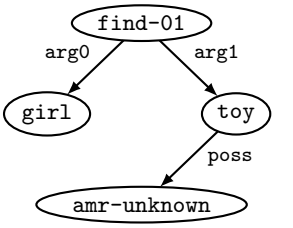
- V is a finite set of *nodes*,
- E is a finite set of *edges*,
- $\mathrm{src}\colon E \to V$ is the *assignment of source nodes to edges*,
- $\mathrm{tar}\colon E \to V$ is the *assignment of target nodes to edges*,
- $\mathrm{lab}_V\colon V \to \Sigma_V$ is the *labelling of nodes*, and
- $\mathrm{lab}_E\colon E \to \Sigma_E$ is the *labelling of edges*.

Langkilde and Knight [LK98] present the software Nitrogen that maps from semantic representations to word lattices, which can in their turn be used to generate natural language sentences. They use a semantic representation that is a developed version of Kasper's sentence plan: the *abstract meaning representation (AMR)*. The AMR syntax is richer than that of a sentence plan, ~~but the two are otherwise identical.~~ an extension of (?)

Fifteen years after the development of Nitrogen, Banarescu et al. [BBC+13] present an updated version of the AMR with the intent of fuelling work on semantic representations in natural language processing. Therefore, their AMR is specifically designed to be appropriate for creating corpora of semantic data. We use an example AMR as a basis for discussing the changes, and as is common, the nodes of a graph are depicted as circles with their label inscribed, and directed edges are arrows with edge labels next to them that point from the source node to the target node of the edge. In Figure 5.2 we see two different ways of expressing the AMR for the semantics described by the sentence "Whose toy did the girl find?". Comparing this to the example sentence plan in Figure 5.1, the syntax is similar, but all labels are now taken from PropBank [PGK05]. PropBank defines semantic roles labels for every word interpretation, which is why some concepts are numbered (e.g. `find-01`). Moreover,

```
(f / find-01
    :arg0 (g / girl)
    :arg1 (t / toy
        :poss (a / amr-unknown)))
```

(a) SPL/AMR format.



(b) Graph format.

Figure 5.2: The semantics of "Whose toy did the girl find?" expressed in two different formats.

the attributes are now called *relations* and are defined in relation to the concept from which they originate. Here, `arg0` denotes the agent of `find-01` and `arg1` denotes its patient. The special concept `amr-unknown` indicates that we do not know the owner (`poss`) of `t` (an instance of `toy`), marking the AMR as a question. Banarescu et al. write that a more detailed specification can be found at the AMR specification page[1], which states that AMRs are acyclic (and thereby also rooted) – a property that is central to some suggested semantic graph language models, as we shall see in the next section.

Corpora are important for the development of language processing algorithms, but it also requires that we are able to evaluate the quality of the output. Letting human experts evaluate the output is, although highly reliable, very expensive. Therefore, automatic evaluation of natural language output is vital for an efficient development process. In the case where the output is a string, the metric Bleu by Papineni et al. [PRWZ02] can be used. Bleu is a similarity score for pairs of sentences and was developed for automatic evaluation of machine translation results. It is based on n-grams, which means that a window of $n \in \mathbb{N}$ sentence parts (e.g., words or morphemes) are evaluated at a time, and that the window is shifted iteratively by one step until all n-grams of a sentence have been evaluated. The output score is a weighted logarithmic average of the precisions of the n-grams. Papineni et al. show that Bleu scores correlate well with human judgements. For automatic semantic graph evaluation, the state-of-the-art method has for some time been to use Smatch, a metric introduced by Cai and Knight [CK13]. Simply put, Smatch greedily calculates the overlap between two semantic structures, but can get stuck in local optima when performing its greedy step. Recently, Song and Gildea [SG19] extended Bleu from strings to graphs in SemBleu, allowing for the evaluation of semantic graphs in terms of a Bleu score. They report that SemBleu does not only correlate better with human judgement than Smatch but is also significantly faster, making SemBleu the new state-of-the-art for automatic semantic graph evaluation.

[1] `https://github.com/amrisi/amr-guidelines/blob/master/amr.md` (visited April 2021).

5.2 Modelling Semantic Graph Languages

Chiang et al. [CDG+18] investigate the AMR corpus of Banarescu et al. with the goal of characterising the AMRs and thereby facilitating the development of a model for semantic graph languages. They arrive at the conclusion that AMRs can be seen as either singly-rooted and possibly cyclic directed graphs, or as multiply-rooted directed acyclic graphs (dags). Adopting the latter point of view, Chiang et al. themselves choose a dag automata formalism as a base for their continued study. Their definition of dag is based on the graph type in Definition 5.1, which allows multiple edges between two nodes. This is convenient from a semantic-graph perspective since one might want to, e.g., be able to use a single person as both the agent and patient of a verb.

Definition 5.2 Let $\Sigma = \Sigma_V \uplus \Sigma_E$ be an alphabet. A *dag* over Σ is a graph $G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab}_V, \mathrm{lab}_E)$ over Σ that does not contain any nonempty directed cycle. The set of all nonempty and connected dags is denoted by $\mathcal{D}_\Sigma$, and a *dag language* is a subset of $\mathcal{D}_\Sigma$.

The dag automata used by Chiang et al. is a simplified version of the dag automata of Quernheim and Knight [QK12b]. The Quernheim and Knight formalism was developed with semantic modelling in mind, after having judged the classical dag automata by Kamimura and Slutzki [KS81] unsuitable for semantic graph generation. Quernheim and Knight investigate properties of their dag automata and later implement a toolkit for their formalism called Dagger [QK12a]. The simplified version used by Chiang et al. is defined below. Note that in their version, the edge labels of dags are dropped, i.e., $|\Sigma_E| = 1$, but since edge labels can be modelled by an intermediate node in between two edges, this is not a severe restriction.

Definition 5.3 A *weighted dag automaton* is a system $M = (\Sigma, Q, \delta, \mathcal{K})$ where

- Σ is an alphabet of *node labels*,
- Q is a finite set of *states*,
- $\mathcal{K} = (\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ is a semiring of *weights*, and
- $\delta \colon \Theta \to \mathbb{K} \setminus \{\mathbb{0}\}$ is a *transition function* that assigns weights to a finite set Θ of *transitions* of the form

$$\{q_1, \ldots, q_m\} \xleftarrow{\sigma/w} \{r_1, \ldots, r_n\}$$

where $\sigma \in \Sigma$, $w \in \mathbb{K}$, and $\{q_1, \ldots, q_m\}$ and $\{r_1, \ldots, r_n\}$ are multisets of elements from Q.

Let $in(v) = \{e \in E \mid \mathrm{tar}(e) = v\}$ and $out(v) = \{e \in E \mid \mathrm{src}(e) = v\}$ be the *incoming* and *outgoing* edges of a node $v \in V$, respectively. Similarly, $|in(v)|$ is the *in-degree* of v and $|out(v)|$ is the *out-degree* of v. A *run of* A on a dag

$G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab}_V, \mathrm{lab}_E)$ is a mapping $\rho\colon E \to Q$ such that R contains the rule

$$\rho(in(v)) \xleftarrow{\mathrm{lab}_V(v)/w} \rho(out(v))$$

for every $v \in V$, where the application of ρ to a set is done element-wise. The processing of a dag can, as indicated by the double arrow in the rule syntax, be done in either a top-down or a bottom-up fashion.

When studying rule-based formalisms, it is common to investigate the properties of their *languages*, which is facilitated by removing their weights. Removing the weights from the dag automaton in Definition 5.3 results in a dag automaton (Q, Σ, R) where R is now the set of rules on the form

$$\{q_1, \ldots, q_m\} \xleftarrow{\sigma} \{r_1, \ldots, r_n\} .$$

Let M be such an unweighted dag automaton, and let G be a dag. If there exists a run of M on G, then M *accepts (recognises)* G, and the *language accepted (recognised) by* M is the set $\mathcal{L}(M)$ of all dags in $\mathcal{D}_\Sigma$ accepted by M. Chiang et al. show that the problem of deciding whether $\mathcal{L}(M)$ is empty is decidable in polynomial time for an unweighted dag automata M, and that the paths of the dags in $\mathcal{L}(M)$ form a regular string language.

kanske skriv en mening om varför man vill det?

While investigating the Banarescu et al. corpus, Chiang et al. make two further observations about the AMRs in the corpus: the first one regards the treewidth of the AMRs, and the second one is about the node degrees of the AMRs. We shall consider both of these observations and how they were used by Chiang et al. to develop algorithms for and extensions of their semantic graph model, starting with the one regarding treewidth.

The treewidth is, informally put, a measure on how tree-like a graph is – we provide the formal definition below.

labelled with

Definition 5.4 A *tree decomposition* of a graph $G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab}_V, \mathrm{lab}_E)$ is a tree T consisting of *bags* (nodes) that are subsets of V and *arcs* (edges). The label of a bag b is denoted by $cont(b)$. T must satisfy the following conditions.

- ⋆ For every $v \in V$, there is a bag b such that $v \in cont(b)$.
- ⋆ For every $e \in E$, there is a bag b such that $\{\mathrm{src}(e), \mathrm{tar}(e)\} \subseteq cont(b)$.
- ⋆ For every $v \in V$, the subgraph of T induced by the bags containing v is connected.

The *width* of T is $\max(|cont(b)| - 1)$ over all bags b of T. The *treewidth* of G (denoted by $tw(G)$) is the minimum of the widths over all its tree decompositions.

Borde kanske säga något om att (begränsad) tw är en viktig parameter för att göra många NP-kompletta graf-problem lös-bara inom poly. tid.

For the AMRs in the Banarescu et al. corpus, the treewidth is at most 4, meaning that algorithms that depend exponentially on the treewidth of the input graph could be feasible in practice. With this in mind, Chiang et al. develop an algorithm that given an input graph G and an ordering of its edges

returns the sum of the weights of all runs of a fixed weighted dag automaton M on G. Informally, the algorithm processes the input graph in the given order and merges nodes, assigning the merged nodes combined weights, until there is only one node left, carrying the resulting weight. The time complexity of the algorithm is shown to depend exponentially on the treewidth, something that was consciously allowed in the light of the empirical investigation of AMRs.

A central problem for all unweighted automata is the recognition problem; it is stated for dag automata below.

Definition 5.5 The *recognition problem* for (unweighted) dag automata asks the question: Given a dag automaton M and a dag G, does M accept G?

It is common to consider two distinct versions of the recognition problem: the *uniform membership problem* that takes both M and G as input, and the *non-uniform membership problem* that only takes G as input. The existence of an efficient algorithm that solves the recognition problem is central to the usability of a rule-based formalism in natural language processing applications: if we cannot efficiently check if a fragment of natural language is correct according to our model, then our model is unlikely to be of any practical use. Since the algorithm by Chiang et al. solves a problem that is more general than the recognition problem for dag automata, their result holds for the recognition of dag automata as well. In general, the non-uniform membership problem for dag automata is NP-complete (and thereby also the harder uniform membership problem), but the result of Chiang et al. shows that when G has bounded treewidth, the recognition problem can be solved in polynomial time.

The second observation about the AMR corpus made by Chiang et al. is that even though the in- and out-degrees of nodes are generally low, there are nodes v for which $|in(v)| + |out(v)| = 17$. This implies that handling an unbounded node degree is desirable for a semantic graph model. For this reason, they extend weighted dag automata to allow for unbounded node degrees: in the extended version, the left- and right-hand sides can be restricted regular expressions that are expressible by an *m-automaton* (see [CDG+18] for its definition and further results). For these extended weighted dag automata, the emptiness and finiteness problems are decidable, and the path language is – as for the non-extended weighted dag automata – regular. Recognition of a graph G for a fixed extended unweighted dag automaton M has the time complexity $O\big(|E|\,(|Q|\,m^{2(tw(G)+2)} + m^{3(tw(G)+1)})\big)$ where m is, informally put, the maximum number of states of the m-automata that describe the left- and right-hand sides of the transitions of M.

Blum and Drewes [BD16, BD19] work on the same type of (unweighted) dag automata as Chiang et al. and prove properties of the resulting languages. The class of *regular dag languages (RDL)* comprises all dag languages that are accepted by dag automata. Blum and Drewes however place one restriction on the dags: they can no longer contain parallel edges. Formally, the requirement is expressed as follows for a dag $G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab}_V, \mathrm{lab}_E)$: For a string s, let $[s]$ denote the smallest set S such that $s \in S^*$. Then, for every edge $e \in E$,

there is exactly one pair $(u, v) \in V \times V$ such that $e \in [out(u)] \cap [in(v)]$. Given this slight modification, Blum and Drewes prove the following statements for RDL and a dag automaton M:

- ⋆ RDL is closed under union and intersection, but not under complement.
- ⋆ There is a pumping lemma for RDL.
- ⋆ It is decidable in polynomial time whether $\mathcal{L}(M)$ is finite, and whether $\mathcal{L}(M)$ is empty.
- ⋆ Unfolding M creates an automaton that defines a regular tree language; also, the path language of M is a regular string language.
- ⋆ It is decidable in polynomial time whether two deterministic dag automata are equivalent.

Next, we survey the work on semantic graph modelling using grammars that operate on graphs. *Hyperedge replacement grammars (HRGs)* were introduced simultaneously and independently by Habel and Kreowski [HK86] and Bauderon and Courcelle [BC87], and has since been shown useful in natural language processing. Overviews of results for the HRG formalism can be found in [Hab92] and [DKH97]. In the following, let $\Sigma = \Sigma_V \uplus \Sigma_E \uplus \Sigma_N$ be an alphabet where Σ_V are the *node labels*, Σ_E are the *hyperedge labels* and Σ_N are the *nonterminal labels*. We require that a function arity$\colon \Sigma_E \uplus \Sigma_N \to 2^{\Sigma_V^*}$ is defined on these sets, which in effect makes the alphabet Σ ranked.

Definition 5.6 A *hypergraph labelled over* Σ (*hypergraph*, for short) is a tuple $G = (V, E, \mathrm{att}, \mathrm{lab}_V, \mathrm{lab}_E)$ where

- ⋆ V is a finite set of *nodes*,
- ⋆ E is a finite set of *hyperedges*,
- ⋆ $\mathrm{att}\colon E \to V^{\circledast}$ is the *attachment of hyperedges*,
- ⋆ $\mathrm{lab}_V\colon V \to \Sigma_V$ is the *labelling of nodes*, and
- ⋆ $\mathrm{lab}_E\colon E \to \Sigma_E$ with $\mathrm{lab}_V^*(\mathrm{att}(e)) \in \mathrm{arity}(\mathrm{lab}_E(e))$ for all $e \in E$ is the *labelling of hyperedges*.

We define the *rank of a hyperedge* $e \in E$ to be $|\mathrm{arity}(\mathrm{lab}_E(e))|$. A hyperedge whose label comes from Σ_N is called a *nonterminal*. The removal of a hyperedge e from a hypergraph G is denoted by $G - e$. Given two hypergraphs G and H, an *isomorphism* $m\colon G \to H$ is a pair of bijective functions $(m_V\colon V_G \to V_H, m_E\colon E_G \to E_H)$ that preserve the labels and the attachments of the graphs. Formally: for every $v \in V_G$ and $e \in E_G$, it holds that $\mathrm{lab}_{V_H}(m_V(v)) = \mathrm{lab}_{V_G}(v)$, $\mathrm{lab}_{E_H}(m_E(e)) = \mathrm{lab}_{E_G}(e)$ and $\mathrm{att}_H(m_E(e)) = m_V^*(\mathrm{att}_G(e))$. If m exists, we say that G and H are *isomorphic*. Henceforth, we do not distinguish between isomorphic hypergraphs.

Given a hypergraph containing one or more nonterminals, we can apply a hyperedge replacement rule to it to generate a new hypergraph.

Definition 5.7 A *context-free (hyperedge replacement) rule* over a labelling alphabet Σ is a pair (L, R) of hypergraphs L, R over Σ such that

- ⋆ the *left-hand side* L is connected and contains exactly one nonterminal, namely e_L, and
- ⋆ the *right-hand side* R is any supergraph of $L - e_L$.

Definition 5.8 Let $r = (L, R)$ be a rule over Σ for which e_L is the nonterminal in L labelled by some $A \in \Sigma_N$, and let $G \in \mathcal{G}$ be a hypergraph that contains a nonterminal e also labelled by A. Then *the application of r to G* is done as specified in the below list.

1. Remove e from G, yielding $G - e$.
2. Disjointly add R to $G - e$.
3. Finally, identify the nodes in $L - e_L$ with the corresponding nodes in R.

The resulting graph is denoted by $H = G[e/R]$, and we write $G \Rightarrow H$ to indicate that H was *derived* from G (by the application of r).

An example context-free rule can be found in Figure 5.3 and an application of that rule to a hypergraph is shown in Figure 5.4. Hyperedges are depicted using squares with the ~~hyperedge or nonterminal~~ label inscribed; for nonterminal labels, we exclusively use capital letters. The attachment of a hyperedge to nodes is shown using lines, and the attachment order of the nodes is indicated using numbers incident to the lines – these numbers are however left out when it is clear from the orientation of the nonterminal how the rule should be applied. Binary hyperedges labelled from Σ_E are equivalent to ordinary directed edges and are therefore drawn as such. To separate the left- and right-hand sides, the symbol ::= is used.

A set of context-free rules together with a fixed hypergraph form a *hyperedge replacement grammar*.

Definition 5.9 A *hyperedge replacement grammar (HRG)* is a tuple $\Gamma = (\Sigma, \mathcal{R}, Z)$ where

- ⋆ Σ is a finite labelling alphabet,
- ⋆ $\mathcal{R}$ is a finite set of context-free rules, and
- ⋆ $Z \in \mathcal{G}_\Sigma$ is a *start hypergraph*.

Let $\Rightarrow^*$ denote the reflexive and transitive closure of $\Rightarrow$. The language generated by an HRG Γ is defined by $\mathcal{L}(\Gamma) = \{G \in \mathcal{G}_{\Sigma \setminus \Sigma_N} \mid Z \Rightarrow^* G\}$, i.e., the set of graphs that can be derived from Z in $\mathcal{G}$. In Definition 5.5, we defined the recognition problem for dag automata. The corresponding problem for graph grammars – such as HRGs – is stated below.

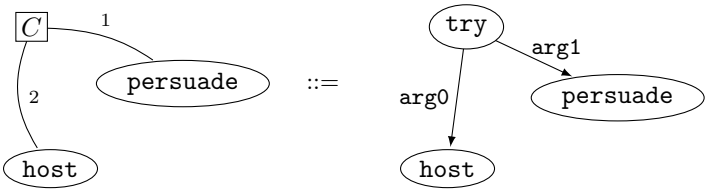


Figure 5.3: A context-free rule adding the control verb "try" to an already existing semantic graph containing the concepts "persuade" and "host". For appropriate usage, the rule requires that `host` is the agent of `try`.

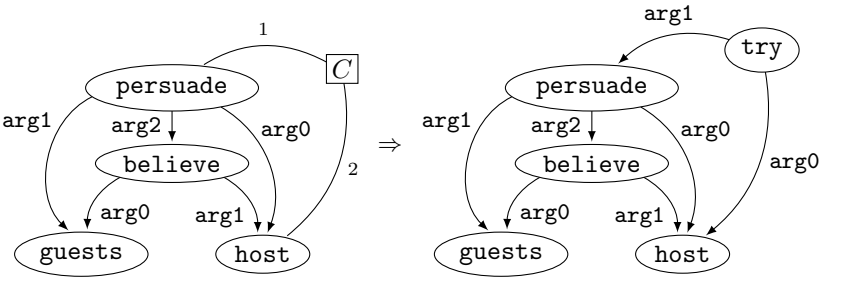


Figure 5.4: An application of the rule in Figure 5.3 to a semantic graph, resulting in the addition of a verb that requires subject control. The resulting graph can be read as "The host tries to persuade the guests to believe them."

Definition 5.10 Given a graph grammar Γ and a graph G, the *parsing problem* asks if there is a derivation of G in Γ (which would imply that $G \in \mathcal{L}(\Gamma)$).

Analogous to the recognition problem, the non-uniform parsing problem takes only G as input whereas both Γ and G are inputs to the uniform variant. Again, efficient recognition or parsing is vital for practical usability since it lets us check if, e.g., a semantic graph is in the language of our model. It is well-known that there are NP-complete HRG languages, meaning that unless P = NP, there is no efficient parsing algorithm for HRGs. The only chance of achieving efficient parsing is restricting the HRG formalism in ways that prevent the construction of NP-complete languages. This road has been travelled on multiple occasions, and here we review a number of restricted HRG formalisms.

Lautemann [Lau90] proves by the construction of an algorithm that uniform parsing of an HRG Γ can be done in polynomial time given the two sufficient conditions listed below:

1. The removal of s nodes from a graph $G = (V, E, \text{att}, \text{lab}_V, \text{lab}_E)$ in $\mathcal{L}(\Gamma)$ never creates more than $O(\log |V|)$ connected components, where the constant s depends on Γ. (This is a generalisation of the commonly seen condition that the graphs generated by Γ have to be connected and of bounded degree to allow for efficient parsing.)

2. For every graph $G \in \mathcal{L}(\Gamma)$ and nonterminal e with $\text{att}(e) = v_1, \ldots, v_r$ and $\text{lab}_E(e) = A \in \Sigma_N$, G can be derived from e if and only if the graph induced by G on $C \cup \{v_1, \ldots, v_r\}$ can be derived from e for every connected component C of $G \setminus \{v_1, \ldots, v_r\}$. In other words: every graph $G \in \mathcal{L}(\Gamma)$ has component-wise derivations.

The parsing algorithm by Lautemann is refined by Chiang et al. [CAB+13] to make it more suitable for natural language processing tasks, and in particular for the parsing of AMRs. The refined algorithm is implemented in the HRG toolkit Bolinas.

Similarly, Drewes et al. make use of restrictions to develop two polynomial-time parsing algorithms for HRG languages: *predictive top-down (PTD)* parsing [DHM15] and *predictive shift-reduce (PSR)* parsing [DHM17]. Both algorithms require an initial analysis of the input HRG to verify that it fulfils a set of requirements that ascertain that the algorithm can be applied to it. The exact set of requirements differ between the two algorithms and is slightly easier to check for the PSR algorithm, but none of the requirement sets can be easily verified by, e.g., inspection of the rules of the HRG. In addition, the PSR parsing requires that an order is specified on the input nonterminals for every right-hand side; the order is used by the algorithm to decide in what order to expand nonterminals. If the requirements are fulfilled, a parser for the particular input HRG is constructed and run on the input hypergraph. For PTD, the time complexity of the (non-uniform) parsing itself is quadratic; for PSR, it is linear. PSR is also shown to be more general, i.e., PSR parses a larger variety of HRG languages than PTD. The PTD and PSR parsing algorithms are extensions to HRGs of $SLL(1)$ parsers and $SLR(1)$ parsers, respectively. In conclusion, the PSR parsing algorithm is more efficient, more general, and has an easier requirement check than the PTD algorithm.

Björklund et al. [BDE16] also develop uniform polynomial parsing for a subset of HRGs, but in the form of dag languages; they call it parsing for *restricted dag (rdag) grammars*. The precondition for applying their parsing algorithm is that the rules of the input grammar are written on a specific format – an rdag grammar normal form. Moreover, Björklund et al. generalise their rdag grammars to *order-preserving HRGs (OPHGs)* [BDES21] by allowing more general rule types in an extended normal form while maintaining uniform polynomial (in fact, quadratic) parsing. A second extension is made to their parsing algorithm by generalising it to weighted OPHGs [BDE19]. In contrast to the PTD and PSR parsing algorithms by Drewes et al., the normal forms used by Björklund et al. make it easy to decide whether the corresponding parsing algorithm can be applied to a particular input grammar.

When modelling semantic graph languages, we want to be able to express all semantic graphs over a domain of concepts and relations that represent a sensical meaning. At the same time, we want to have the ability to leave out semantic graphs with nonsensical meaning. A strength of hyperedge replacement is that it provides the structural control that some language constructions

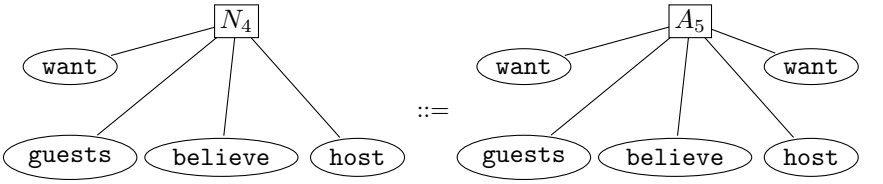


Figure 5.5: Context-free rule for adding the concept `want` to a semantic graph.

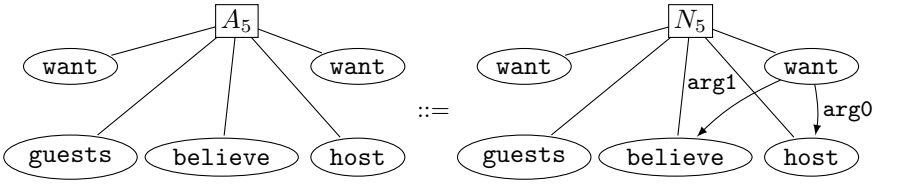


Figure 5.6: Context-free rule for adding outgoing edges from the concept `want` to other nodes of a semantic graph.

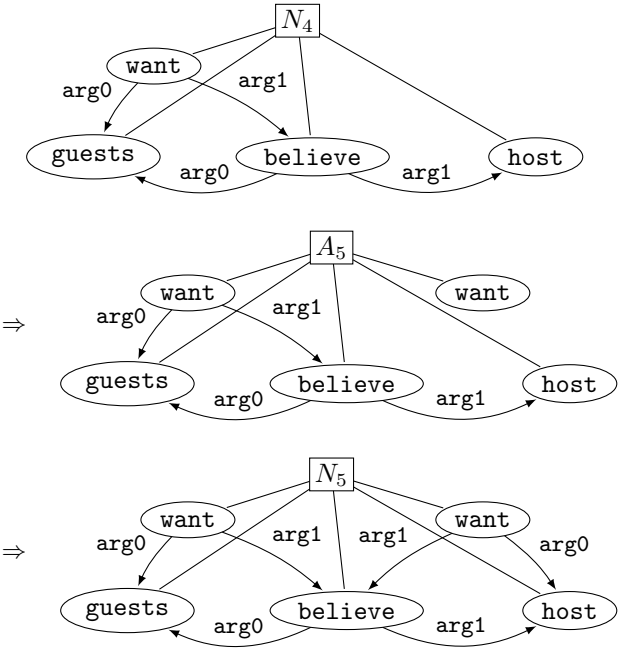


Figure 5.7: A derivation of a verb that does not require any control using the context-free rules in Figures 5.5 and 5.6, in that order. The first hypergraph can be interpreted as "The guests want to believe the host" and the last one reads "The guests want to believe the host and the host wants the guests to believe them."

require. To exemplify this, again consider Figure 5.3 which depicts a context-free rule that defines the addition of the verb "try" to a semantic graph. The verb "try" requires what we here call *subject control*, i.e., it needs access to not only the concept that is to be tried, but also the agent of that concept. In other words, the one doing the trying should do the thing that is to be tried as well. For example "The host tries that their pet persuades the guests to believe the host" does not make any sense. Similarly, the verb "persuade" (also in the example) requires *object control*: the guests have to be the ones doing the believing. Figure 5.4 shows the application of the rule in Figure 5.3 to an existing semantic graph with one nonterminal that mark the verb that "try" should refer to as well as the agent of that verb, i.e, the one who is to try.

The weakness of hyperedge replacement is that there is no possibility of relaxing the structural control in cases we do not need it. For example, when adding a non-control verb, for example "believe", it does not matter who does the believing – the resulting semantic graph will be sensical as long as the believer is a creature with consciousness. Figures 5.5 and 5.6 portray two context-free rules; in Figure 5.7, they are used to derive a new semantic representation from an existing one by adding the concept `want`. Note that any choice of target of `arg1` would have created a valid semantic representation, and for `arg0`, any node representing one or more persons could have been chosen. In this case, we have depicted the case where the nodes `host` and `believe` are chosen as the targets of `arg0` and `arg1`, respectively. However, to cover all of the possible correct semantic representations that can be created in this way, we would need a rule for every combination of `arg0` and `arg1` targets. The dag automata we reviewed earlier has the ability to add structures without any control, but instead completely lack structural control.

A formalism that provides structural control while allowing free addition of structures is the *contextual hyperedge replacement grammar (CHRG)* developed by Drewes et al. [DHM12, DH15]. CHRG is a generalisation of HRG by the addition of what its creators call *contextual rules*. Below, we provide formal definitions for the new rule type and its formalism.

Definition 5.11 A *contextual (hyperedge replacement) rule is a context-free rule* (L, R) *for which* L *is disconnected.*

Definition 5.12 A *contextual hyperedge replacement grammar (CHRG)* is an HRG $\Gamma = (\Sigma, \mathcal{R}, Z)$ that allows $\mathcal{R}$ to contain both context-free rules and contextual rules.

Figure 5.8 contains two rules of which the right-most is contextual. The application of a contextual rule is done exactly as for a context-free rule: Figure 5.9 shows the application of the two rules in Figure 5.8 to a semantic graph. Note that the example semantic graph is exactly the example we looked at for context-free rules to demonstrate how HRG forces the usage of structural control regardless of whether it is needed. The current example illustrates how the addition of contextual rules allows us to let go of the control of nodes and still

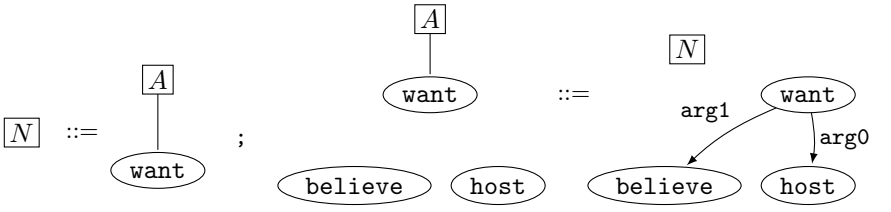


Figure 5.8: Two semicolon separated rules. To the left: a context-free rule. To the right: a contextual rule whose left-hand side contains two contextual nodes (`believe` and `host`).

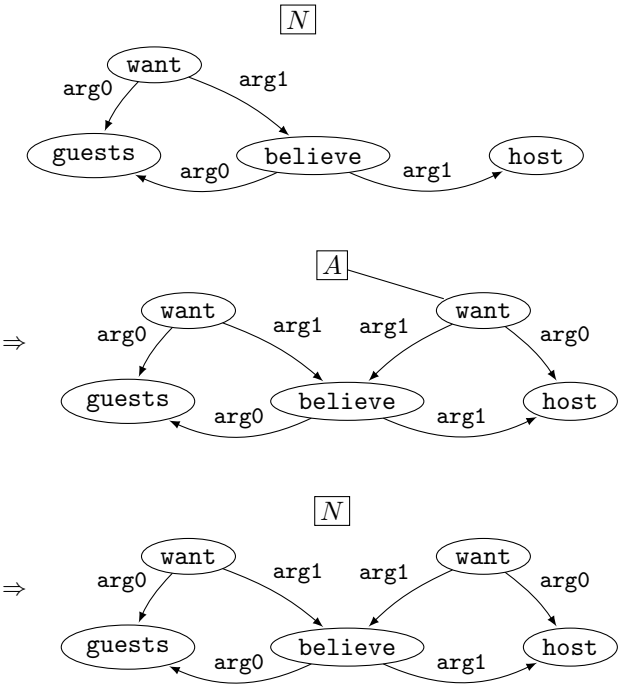


Figure 5.9: The same derivation as in Figure 5.7 but this time using the rules from Figure 5.8 out of which one is contextual.

access them at a later stage purely based on their label. We see that the rules needed to achieve this behaviour are simpler and contain fewer nonterminals than when only context-free rules were allowed. It is not surprising that the addition of contextual rules create a more powerful formalism, but as we shall see, not too powerful for efficient parsing of non-trivial subclasses of CHRG languages. (The observations made here regarding the suitability of CHRGs for semantic representations are largely taken from Paper V – see Chapter 6 for a paper summary).

Drewes et al. extend their PSR parsing algorithm for HRG to the recently introduced CHRG [DHM19]. The approach is the same: First, an analysis is performed on the input grammar, and if it passes the analysis, the grammar is parsable and a parser is created for it. Next, the created parser is used to parse the input hypergraph – still in linear time. Furthermore, a recent study by Drewes et al. show that their PTD parsing algorithm can also be extended to the contextual case [DHM21].

Recall that hyperedge replacement was first introduced by two independent groups of researchers. The formalism presented here is based on how hyperedge replacement was described by Habel and Kreowski [HK86]. In contrast, Bauderon and Courcelle [BC87] use the approach of Mezei and Wright [MW67], namely evaluating a tree from a regular tree language using operations from an algebra. An algebra is, informally, a set of structures together with operations defined on those structures, and Bauderon and Courcelle were the first to use an algebra over graphs for graph generation. In Paper VI, we extend the rewriting system by Bauderon and Courcelle with contextuality to achieve a formalism with non-uniform polynomial parsing that can be used to model semantic graph languages (see Chapter 6 for an extended summary).

5.3 Mapping Between Language Representations

In the previous section, we considered various formalisms representing graph languages and their recognition or parsing problem. Another important problem in natural language processing is the mapping between different language representations. For example, given a sentence, we might want to map it to a syntax tree (performing a syntactic analysis of the sentence) or to a semantic graph (conducting a semantic analysis). Mapping is commonly referred to as "parsing", but here we use "mapping" to avoid the frequent confusion over the two different usages of the word "parsing" in language processing.

Even though the papers included in this thesis do not handle mapping problems, it is interesting to review the mapping literature since the mapping task is as important as the parsing task, and methods for doing one might be interesting for the other. In other words, the solutions used overlap.

Braune et al. [BBK14] create a corpus of pairs of English strings and their corresponding semantic graphs in the form of AMRs. The strings are chosen to make the semantic graphs highly reentrant, meaning that the proportion of nodes with several incoming edges is high. Reentrancies are interesting since they are what separates graphs from trees, and thus what makes problems on graphs more difficult to solve than their tree counterpart. In Figure 5.4, both hypergraphs contain two reentrant nodes, namely the ones labelled `host` and `guests`. The purpose of creating this corpus is to promote research on AMRs, and to lead by example, Braune et al. make a study of their own in which they compare three ways of mapping between the AMRs and the strings:

1. A synchronous HRG (SHRG), which generates a string and a graph in parallel using the same rule set. The SHRG can thus easily be used for mapping in both directions.
2. A composition of two devices: a bottom-up dag-to-tree transducer (d2t) that creates a tree from the AMR, and a top-down tree-to-string transducer (LNT) that takes the yield of the tree to extract the English string.
3. A composition of four devices: a d2t as in the previous composition, two extended tree-to-tree transducers (xLNT) – one that introduces verbs and one that introduce pronouns – and finally an LNT, again for extracting the yield of the resulting tree.

They compare each of the methods on two tasks: (1) natural language generation, which produces a string given an AMR and is therefore evaluated using Bleu, and (2) natural language understanding, which maps a string to its corresponding AMR by applying the methods backwards and is consequently evaluated using Smatch. The results show that the two compositional methods outperform the SHRG method for (1), but that the roles are reversed for (2). This is not very surprising since the SHRG holds all information to directly map from a string to its semantic graph, while the two other methods are specialised in forming a syntactically correct string given a semantic graph.

An alternative formalism that can be used for the mapping of syntactic trees to semantic graphs is the tree-to-graph transducer t2g developed by Björklund et al. [BCDS20]. The t2g formalism is based on Z-automata, which are tree automata for unranked trees, such as dependency trees. A dependency tree is a tree showing the relations of the syntactic roles in a natural language sentence, but different from a constituent syntax tree as seen in Chapter 1. Given a dependency tree, t2g transforms it into a corresponding semantic graph in polynomial time.

Next, we consider the mapping from a string in one natural language (the *source language*) to a string in another (the *target language*), commonly known as *machine translation*. Jones et al. [JAB+12] develop a machine translation pipeline based on SHRGs. We previously saw that an SHRG is an HRG that generates a string and a graph in parallel. This is achieved by having two right-hand sides for each of the rules of the SHRG: one that works on a string and another that works on a graph. Taking only the right-hand sides that work towards generating a string, we get the *string projection* of the SHRG; the *graph projection* is defined analogously.

The main idea of the translation pipeline is the following: Create an SHRG each for the source and target language, convert the input string to a derivation in the source SHRG, find the corresponding derivation in the target SHRG, and convert that derivation to an output string. To create these SHRG, data in the form of semantic graphs with corresponding strings in each language is needed.

Given a corpus of string-graph pairings in a single language (e.g., the Braune et al. AMR corpus for English), the first step is to align the con-

cepts of the semantic graphs with the words in the string. For this alignment, Jones et al. develop an algorithm they call DepDep (for algorithm specification, see [JAB+12]). Next, they must extract SHRG rules from the set of aligned string-graph pairings. Two approaches are developed for this purpose: CanSem and SynSem. CanSem specifies a set of template rules in advance that allows them to find a derivation tree of a graph directly, and then acquire the rules needed for deriving the string-graph pairs of the corpus. SynSem works directly on the graph, processing it in such a way that its structure is reflected in the nonterminals of the resulting SHRG (again, see [JAB+12] for details on these two methods). For the purpose of comparing CanSem and SynSem, Jones et al. create two SHRGs per language, one per rule extraction method, even though only one SHRG per language can be used at a time in the actual translation. These SHRGs are then assigned weights, making them probabilistic. Now as the SHRGs are finished, the translation can start.

The source language string – the one we want to translate into a string in the target language – is first transformed into a corresponding semantic graph using the SHRG of the source language: the string is parsed with the string projection of the grammar yielding a derivation, and then the graph corresponding to the derivation is created. Then, the graph is transformed into an output string in the target language by parsing the graph with respect to the graph projection of the target SHRG – the resulting derivation is used together with the string projection of the SHRG to produce a string.

Their results contain an experimental evaluation of their resulting machine translation system. As part of the evaluation, they compare CanSem to SynSem and DepDep to a standard alignment technique by IBM. To summarise the comparison result: For the alignment, DepDep scores better in precision but worse in recall than the IBM method, and according to the Bleu score, CanSem clearly outperforms SynSem for the translation task.

Peng et al. [PSG15] introduce an SHRG based approach for transforming strings into AMRs, which should be considered an extension of the method used by Jones et al. [JAB+12] for turning their input strings into semantic graphs. In contrast to Jones et al., Peng et al. base the creation of the SHRG from a corpus on top-down sampling [CFGŠ14] and take the concepts and relations from PropBank. Moreover, Peng et al. use the Earley parsing algorithm together with cube pruning [Chi07] for the string-to-graph mapping itself whereas Jones et al. use standard CKY parsing for the same purpose.

Chapter 6

Contributions

Here, the papers on semantic modelling included in this thesis are summarised and related to the work in Chapter 5.

Paper V: Contextual Hyperedge Replacement Grammars for Abstract Meaning Representations

In Paper V, we investigate whether CHRGs are a good formalism for representing semantic graph languages, and in particular AMR languages. We show that the context-free rules give us precise control over the appearance of the surrounding structure when adding a node to a hypergraph – provided that we keep structural information about previously added nodes in memory by attaching a nonterminal to them. In addition, we show that we can easily allow nodes and hyperedges to be added without any other control than what can be achieved via the node labels. In terms of AMRs, this means that we can handle concepts that require subject and object control while the contextual rules give us ~~a certain freedom:~~ the freedom to attach new relations to concepts with the concept label being the only controlling factor.

To illustrate our findings, we build an example CHRG that generates an AMR language of dags ~~that may have multiple roots~~. In our CHRG, the concepts `try` and `persuade` exemplify verbs that need subject and object control, respectively, whereas the concepts `want` and `believe` represent verbs that do not require any structural control. The example CHRG builds the dags bottom-up and generates one node at a time, and the node currently being generated is said to be *active*. To ensure acyclicity of the resulting graphs, only the active node can be assigned new outgoing edges. Similarly, the CHRG is defined in such a way that the graphs are kept connected. The rules of the CHRG are fewer and in total there are fewer nonterminals with a smaller number of attached nodes ~~in~~ on average than a corresponding HRG.

Paper VI: Polynomial Graph Parsing with Non-Structural Reentrancies

Paper VI is a continuation of the work in Paper V: based on our observations from the latter, we develop a formalism which is simpler than CHRG but still exploits the advantage that contextual rules embody.

Our formalism is called *graph extension grammars* and extends hyperedge replacement as defined by Bauderon and Courcelle [BC87] with contextuality. A graph extension grammar generates the graph in a bottom-up fashion and allows us to refer to the parts of the graph that are already generated (but not the other way around). Furthermore, outgoing edges from a node can only be added when the node is created. Thus, graph extension grammars generate (possibly) multiple-rooted dags. Moreover, we present an algorithm for polynomial non-uniform parsing, but we also present conditions on the grammar under which the parsing can be done uniformly in linear time. We show that an example graph extension grammar that generates semantic graphs over a small domain of concepts and relations is parsable in ~~quadratic~~ time.

To compare our formalism with the ones summarised in Section 5.2: Graph extensions constitute, if you will, a context-free version of contextual rules, referring to the fact that we can only contextually access the part of the graph that is already generated, meaning the nodes "under" the currently generated node. We noted earlier that dag automata has the problem that they cannot implement the control needed for control verbs. CHRGs solve this, but instead they have the problem that we might refer to something that is not yet generated – depending on in what order we evaluate the nonterminals. The graph extension grammar formalism fixes these two problems. However, since the contextual aspect of the graph extensions are not as powerful as contextual rules, there are potentially semantic constructions that cannot be generated in this way. What types of semantic graphs can and cannot be generated by this formalism remains to be investigated.

References

[BBC+13] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, 2013.

[BBK14] Fabienne Braune, Daniel Bauer, and Kevin Knight. Mapping between English strings and reentrant semantic graphs. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation*, pages 4493–4498, 2014.

[BC87] Michel Bauderon and Bruno Courcelle. Graph expressions and graph rewritings. *Mathematical Systems Theory*, 20(1):83–127, 1987.

[BCDS20] Johanna Björklund, Shay B. Cohen, Frank Drewes, and Giorgio Satta. Bottom-up unranked tree-to-graph transducers for translation into semantic graphs. *Theoretical Computer Science*, 2020.

[BD16] Johannes Blum and Frank Drewes. Properties of regular dag languages. In *10th International Conference on Language and Automata Theory and Applications*, 2016.

[BD19] Johannes Blum and Frank Drewes. Language theoretic properties of regular DAG languages. *Information and Computation*, 265:57 – 76, 2019.

[BDE16] Henrik Björklund, Frank Drewes, and Petter Ericson. Between a rock and a hard place — parsing for hyperedge replacement DAG grammars. In *10th International Conference on Language and Automata Theory and Applications*, 2016.

[BDE19] Henrik Björklund, Frank Drewes, and Petter Ericson. Parsing weighted order-preserving hyperedge replacement grammars. In *Proceedings of the 16th Meeting on the Mathematics of Language*, pages 1–11, Toronto, Canada, July 2019. Association for Computational Linguistics.

[BDES21] Henrik Björklund, Frank Drewes, Petter Ericson, and Florian Starke. Uniform parsing for hyperedge replacement grammars. *Journal of Computer and System Sciences*, 118:1–27, 2021.

[BDZ15] Johanna Björklund, Frank Drewes, and Niklas Zechner. An efficient best-trees algorithm for weighted tree automata over the tropical semiring. In *Proc. 9th Intl. Conf. on Language and Automata Theory and Applications (LATA 2015)*, volume 8977 of *LNCS*, pages 97–108, 2015.

[BGSV10] Matthias Büchse, Daniel Geisler, Torsten Stüber, and Heiko Vogler. n-best parsing revisited. In *Proceedings of the 2010 Workshop on Applications of Tree Automata in Natural Language Processing*, pages 46–54, Uppsala, Sweden, July 2010. Association for Computational Linguistics.

[CAB+13] David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 924–932, 2013.

[CDG+18] David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. Weighted DAG automata for semantic graphs. *Computational Linguistics*, 44(1):119–186, March 2018.

[CFGŠ14] Tagyoung Chung, Licheng Fang, Daniel Gildea, and Daniel Štefankovič. Sampling tree fragments from forests. *Computational Linguistics*, 40(1):203–229, March 2014.

[Chi07] David Chiang. Hierarchical phrase-based translation. *Computational Linguistics*, 33(2):201–228, 2007.

[CK13] S. Cai and K. Knight. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of ACL*, 2013.

[DH15] Frank Drewes and Berthold Hoffmann. Contextual hyperedge replacement. *Acta Informatica*, 52(6):497–524, 2015.

[DHM12] Frank Drewes, Berthold Hoffmann, and Mark Minas. *Applications of Graph Transformations with Industrial Relevance: 4th International Symposium, Revised Selected and Invited Papers*, chapter Contextual Hyperedge Replacement, pages 182–197. 2012.

[DHM15] Frank Drewes, Berthold Hoffmann, and Mark Minas. Predictive top-down parsing for hyperedge replacement grammars. In *Proceedings of the 8th International Conference on Graph Transformation*, pages 19–34, 2015.

[DHM17] Frank Drewes, Berthold Hoffmann, and Mark Minas. Predictive shift-reduce parsing for hyperedge replacement grammars. In *Proc. 10th Intl. Conf. on Graph Transformation (ICGT'17)*, Lecture Notes in Computer Science, pages 106–122, 2017.

[DHM19] Frank Drewes, Berthold Hoffmann, and Mark Minas. Extending predictive shift-reduce parsing to contextual hyperedge replacement grammars. In Esther Guerra and Fernando Orejas, editors, *Graph Transformation*, pages 55–72, Cham, 2019. Springer International Publishing.

[DHM21] Frank Drewes, Berthold Hoffmann, and Mark Minas. Rule-based top-down parsing for acyclic contextual hyperedge replacement grammars. In *Proc. 14th Intl. Conf. on Graph Transformation (ICGT'21)*, Lecture Notes in Computer Science, 2021.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DKH97] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. Hyperedge replacement graph grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 95–162. World Scientific Publishing Co., Inc., 1997.

[Epp98] David Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.

[FMN06] Jenny Rose Finkel, Christopher D. Manning, and Andrew Y. Ng. Solving the problem of cascading errors: Approximate Bayesian inference for linguistic annotation pipelines. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 618–626, Sydney, Australia, July 2006. Association for Computational Linguistics.

[Hab92] Annegret Habel. *Hyperedge replacement: grammars and languages*, volume 643. Springer Science & Business Media, 1992.

[HC05] Liang Huang and David Chiang. Better k-best parsing. In *Proceedings of the Conference on Parsing Technology 2005*, pages 53–64. Association for Computational Linguistics, 2005.

[HK86] Annegret Habel and Hans-Jörg Kreowski. May we introduce to you: Hyperedge replacement. In *Graph-Grammars and Their Application to Computer Science*, pages 15–26. Springer, 1986.

[JAB+12] Bevan Jones, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, and Kevin Knight. Semantics-based machine translation with hyperedge replacement grammars. In *Proceedings of COLING 2012: Technical papers*, pages 1359–1376, 2012.

[Kas89] Robert T. Kasper. A flexible interface for linking applications to Penman's sentence generator. In *HLT '89: Proceedings of the workshop on Speech and Natural Language*, pages 153–158. Association for Computational Linguistics, 1989.

[KG05] Kevin Knight and Jonathan Graehl. An overview of probabilistic tree transducers for natural language processing. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 1–24. Springer, 2005.

[Knu77] Donald E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6:1–5, 1977.

[KS81] Tsutomu Kamimura and Giora Slutzki. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51, 1981.

[Lau90] Clemens Lautemann. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27(5):399–421, 1990.

[LK98] Irene Langkilde and Kevin Knight. Generation that exploits corpus-based statistical knowledge. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics - Volume 1*, pages 704–710, 1998.

[MK06] Jonathan May and Kevin Knight. Tiburon: A weighted tree automata toolkit. In *International Conference on Implementation and Application of Automata*, pages 102–113. Springer, 2006.

[Moh02] Mehryar Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

[MR02] Mehryar Mohri and Michael Riley. An efficient algorithm for the n-best-strings problem. In *Proceedings of the Conference on Spoken Language Processing*, 2002.

[MW67] Jorge Mezei and Jesse B. Wright. Algebraic automata and context-free sets. *Information and Control*, 11:3–29, 1967.

[Pen89] Penman. *The Penman Documentation and User Guide*. The Penman project, USC/Information Sciences Institute, Marina del Rey, California, 1989.

[PGK05] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1):71–106, March 2005.

[PRWZ02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[PSG15] Xiaochang Peng, Linfeng Song, and Daniel Gildea. A synchronous hyperedge replacement grammar based approach for AMR parsing. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning*, pages 32–41, Beijing, China, July 2015. Association for Computational Linguistics.

[QK12a] D. Quernheim and K. Knight. Dagger: A toolkit for automata on directed acyclic graphs. In *10th International Workshop on Finite State Methods and Natural Language Processing*, page 40, 2012.

[QK12b] Daniel Quernheim and Kevin Knight. Towards probabilistic acceptors and transducers for feature structures. In *Proceedings of the Sixth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 76–85, Jeju, Republic of Korea, July 2012. Association for Computational Linguistics.

[SBMN13] Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 455–465, 2013.

[SG19] Linfeng Song and Daniel Gildea. SemBleu: A robust metric for AMR parsing evaluation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4547–4552, Florence, Italy, July 2019. Association for Computational Linguistics.

[ZZT18] Yanpeng Zhao, Liwen Zhang, and Kewei Tu. Gaussian mixture latent vector grammars. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Volume 1: Long Papers*, pages 1181–1189, 2018.

I

Finding the N Best Vertices in an Infinite Weighted Hypergraph[☆,☆☆]

Johanna Björklund[a], Frank Drewes[a], Anna Jonsson[a]

[a]*Department of Computing Science, Umeå University, 901 87 Umeå, Sweden*

Abstract

We propose an algorithm for computing the N best vertices in a weighted acyclic hypergraph over a *nice* semiring. A semiring is nice if it is finitely-generated, idempotent, and has $\mathbb{1}$ as its minimal element. We then apply the algorithm to the problem of computing the N best trees with respect to a weighted tree automaton, and complement theoretical correctness and complexity arguments with experimental data. The algorithm has several practical applications in natural language processing, for example, to derive the N most likely parse trees with respect to a probabilistic context-free grammar.

Keywords: Hypergraph, N-best problem, Idempotent semiring

1. Introduction

Suppose that we can solve an optimisation problem A by solving, in succession, the problems $A_1, \ldots, A_n$. One way of approaching the joint optimisation over the cascade $A_1, \ldots, A_n$ is to find the N best solutions to A_1, and take these as input to A_2. We then compute the N best solutions to A_2 for each of these inputs, and prune the combined output down to the N best alternatives. The computation continues in this fashion until we have the outputs for A_n, at which point we take the top-ranking one as the best solution to A. In general, this approach will not yield an optimal solution, but it is often a viable heuristic.

The problem of finding N top-scoring elements with respect to some ranking device is referred to as the N-best problem. Such rankings can be computed by weighted automata, which associate with each element a value in some ordered semiring. The ranking prefers cheaper elements to more expensive ones, that is, N distinct elements with as small values as possible are sought. Typically,

☆Dedicated to Prof. Dr. Jürgen Dassow on the occasion of his 70th birthday.

☆☆Preliminary presentations of different parts of this work have been given at *Weighted Automata: Theory and Applications* (WATA 2016) and *Trends in Tree Automata and Tree Transducers* (TTATT 2016).

Email addresses: johanna@cs.umu.se (Johanna Björklund), drewes@cs.umu.se (Frank Drewes), aj@cs.umu.se (Anna Jonsson)

such automata may be nondeterministic, in which case the values of the individual runs (that is, of the nondeterministic computations) are summed up. The N-best problem is thus a harder problem than the related *N-best derivations problem*, which asks for the N best individual runs of the automaton. In the most common situation, where the semiring is the min-plus semiring with addition being minimum, the best run is always a run on the best element, but among the N best runs several may actually be runs on the same element.

Mohri and Riley [7] provide an algorithm for solving the N-best problem for weighted string automata over the min-plus semiring $\mathcal{A} = (\mathbb{R}_+^\infty, \min, +, \mathbb{0}, \infty)$. To keep the running time polynomial, they use a combination of lazy determinisation and Dijkstra's N-shortest paths algorithm. In [2], we generalise this algorithm to work for weighted tree automata over $\mathcal{A}$.[1] Such an extension is of interest because weighted tree automata are widely used in natural language processing to represent the parse trees of probabilistic context-free grammars. The probability values can be computed in the Viterbi semiring, or alternatively in the min-plus semiring by moving to the domain of negative log likelihoods. The latter method is preferred in many practical applications as it leads to better numerical precision. The extension to weighted tree automata in [2] simplifies the search technique by working directly with the input automaton rather than an on-the-fly determinisation. To mitigate the added dimensionality caused by working with trees rather than strings, an additional pruning technique is applied in order to arrive at an efficient algorithm. The resulting running time is roughly comparable to that in [3] for computing the N-best derivations.

In this paper we consider the N-best problem for weighted hypergraphs over nice semirings, i.e. semirings which are idempotent, finitely generated, and have $\mathbb{1}$ as their smallest element. The convenience of idempotency in the context of search algorithms has already been shown in [6]. The hypergraphs may be infinite, but may not contain cycles. This provides an abstraction and generalisation of the approach in [2]. Roughly speaking, the hypergraph may be instantiated to represent the set of all trees over a given ranked alphabet. Each vertex corresponds to the root of a tree, and a hyperedge connects it to the vertices representing the roots of its direct subtrees. Naturally, the weight of each vertex should be the weight of the tree it represents, and hence the N-best problem is to find the N cheapest vertices of the hypergraph. The exact correspondence is somewhat more complex than that, because we also have to account for the states of the tree automaton and the distinction between final and non-final states.

The framework based on weighted hypergraphs over nice semirings can be adapted to weighted finite-state devices whose computation graphs are acyclic. These include finite-state string automata, unranked tree automata, and context-free grammars. Every extremal semiring is idempotent, so for example the Łukasiewicz-, Viterbi-, and Boolean semirings meet this condition. These semirings have also the property that the multiplicative $\mathbb{1}$ is their smallest element.

[1] A version of [2] providing more detail can be found in [1].

Out of these, only the Boolean semiring is finitely generated and hence nice. However, for all three semirings, the subset of weights that actually appear in the computation graphs of the previously mentioned devices are generated by the devices' edge- and transition weights, which make up finite sets. This observation makes our framework applicable to, e.g., weighted string automata over the Viterbi semiring.

1.1. Outline

This paper is organised as follows. Section 2 recalls basics of graph theory and algebra, and introduces nice semirings and layered graphs. In Section 3, we present an N-best meta algorithm for weighted hypergraphs and prove its correctness. The algorithm is abstract in the sense that it builds on top of a small set of auxiliary procedures, and the realisation of these procedures depends on the type of graph being studied. The algorithm is used in Section 4 to solve the N-best problem for wta over the min-plus semiring and the theoretical results are complemented with practical experiments. Section 5 concludes this paper by summarizing our findings and mapping out future work.

2. Preliminaries

We write $\mathbb{N}$ for the set of non-negative integers, $\mathbb{N}^\infty$ for $\mathbb{N} \cup \{\infty\}$, $\mathbb{R}_+$ for the set of non-negative reals, and $\mathbb{R}_+^\infty$ for $\mathbb{R}_+ \cup \{\infty\}$. Given $n \in \mathbb{N}$, we let $[n] = \{1, \ldots, n\}$, and $[\infty] = \mathbb{N}$. In particular, $[0] = \emptyset$. Let S and S' be a pair of (possibly infinite) sets. The number of elements of S is written $|S|$, and the powerset of S is denoted by $\mathit{pow}(S)$. Given a k-tuple $v = (a_1, \ldots, a_k)$ we may denote its ith component a_i ($i \in [k]$) by $v(i)$. A sequence of elements of S is *non-repetitive* if it does not contain the same element in two distinct positions. Given a function $\pi\colon S \to S'$, we extend π to a function from sequences over S to sequences over S' in the usual elementwise fashion. Similarly, π is extended to a function $\pi\colon \mathit{pow}(S) \to \mathit{pow}(S)$. Note that we do not make a notational distinction between π and these two canonical extensions of π. Given a non-empty sequence w, $[w]$ denotes the smallest set S such that w is a sequence over S. The set of all strings (i.e., finite sequences), over S is denoted by S^*; it includes the empty string λ.

A commutative *semiring* is a tuple $\mathcal{A} = (A, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ such that $(A, \oplus, \mathbb{0})$ and $(A, \otimes, \mathbb{1})$ are commutative monoids, $\otimes$ distributes both-sided over $\oplus$, and $\mathbb{0}$ is an absorbing element with respect to $\otimes$. In the following, we generally assume that $\otimes$ binds stronger than $\oplus$, so $a \oplus b \otimes c$ is interpreted as $a \oplus (b \otimes c)$.

Since all semirings considered in this paper are commutative, we do not explicitly mention commutativity throughout the rest of the paper.

A *quasi-order* on S is a reflexive, transitive, binary relation $\leq$. We write $a < b$ to express that $a \leq b$ but $b \not\leq a$. As usual, $\geq$ and $>$ denote the inverses of $\leq$ and $<$, resp. A quasi-order is a *partial order* if it is antisymmetric. It is *well-founded* if there are no infinite descending chains, i.e., there is no infinite sequence $a_1 > a_2 > a_3 > \cdots$. A stronger notion than that of well-foundedness

is that of a *well quasi-order* (wqo). A quasi-order is a wqo if every infinite sequence $a_1, a_2, \ldots$ eventually increases, i.e., there are $i < j$ such that $a_i \leq a_j$.

A semiring $\mathcal{A}$ is *idempotent* if $a \oplus a = a$ for all $a \in A$. In this case, there is a partial order $\leq_{\mathcal{A}}$ on $\mathcal{A}$, called the *natural order* of $\mathcal{A}$, which is given by $a \leq_{\mathcal{A}} b \iff a \oplus b = a$. Idempotent semirings are *monotonic* with respect to their natural order [6, Lemma 2], in other words, $a \leq_{\mathcal{A}} b$ implies $a \odot c \leq_{\mathcal{A}} b \odot c$ for all $c \in A$ and $\odot \in \{\oplus, \otimes\}$.

The semiring $\mathcal{A}$ is *finitely generated* if there is a finite subset A' of A such that every $a \in A$ is a sum of products of elements in A'. A finitely generated idempotent semiring in which $\mathbb{1}$ is the smallest element is said to be *nice*. In almost all applications, the fact that nice semirings are required to be finitely generated is an insignificant restriction because weights are usually constructed on the basis of a finite subset of $\mathcal{A}$ (such as the weights of the rules of a weighted tree automaton), and the sub-semiring generated by this subset is by definition finitely generated.

Example 1. Clearly, every extremal semiring is idempotent, but not every idempotent semiring is extremal: The min-plus semiring over $\mathbb{R}_+^\infty$ is an idempotent and extremal semiring with $\min(a, b)$ serving as semiring addition and ordinary addition as semiring multiplication. However, by generalising the domain to vectors of length $k \in \mathbb{N}$, $k \geq 2$, over $\mathbb{R}_+^\infty$ and applying semiring addition and multiplication component-wise, we get an idempotent semiring that is not extremal. If we restrict the domain of the min-plus semiring to $\mathbb{N}^\infty$, it becomes finitely generated (by $\{\mathbb{0}, \mathbb{1}, \infty\}$) and, in fact, nice because $\mathbb{1} = 0$ is its smallest element. The extension to vectors is still nice.

Rather than working on ordinary graphs, in which edges have a single source and a single target, we consider hypergraphs in which hyperedges may have multiple sources ordered as a sequence, but still only one target. This is particularly convenient for representing sets of (ordered) trees: a hyperedge labelled f with n sources corresponds to an occurrence of the symbol f of rank n in a tree. The sources and the target represent the roots of the direct subtrees and of the tree itself, respectively.

Definition 1 (Hypergraph). A *hypergraph* is a tuple $G = (V, E, src, tar)$ such that

- V and E are disjoint sets of *vertices* and *hyperedges*, respectively,
- $src\colon E \to V^*$ assigns to each $e \in E$ a sequence of *sources* $src(e)$, and
- $tar\colon E \to V$ assigns to each $e \in E$ a *target* $tar(e)$.

A vertex $v \in V$ is called an *end* if $v \notin [src(e)]$ for all $e \in E$. A *path* in G is a nonempty sequence of edges $\pi = e_0 \cdots e_n$ such that, for all $i \in [n]$, $tar(e_{i-1}) \in [src(e_i)]$. The *sources* and the *target* of π are $src(\pi) = src(e_0) \in V^*$ and $tar(\pi) = tar(e_n) \in V$, respectively, and we say that $tar(\pi)$ is *reachable*

from e_0. The set of all paths in G is denoted $\mathit{paths}(G)$. The set of vertices that are *descendants* of v is given by

$$\mathit{descendants}(v) \;=\; \{v\} \cup \bigcup_{\substack{\pi \,\in\, \mathit{paths}(G),\\ v \,=\, \mathit{tar}(\pi)}} [\mathit{src}(\pi)] \;.$$

If $v' \in \mathit{descendants}(v)$, then v' is a *descendant* of v, and v is an *ancestor* of v'.

With this definition, hypergraphs can be infinite structures and may have parallel hyperedges. Ordinary graphs are a special case of hypergraphs, namely the one where each hyperedge has exactly one source. In the following, we shall simply speak of graphs and edges instead of hypergraphs and hyperedges.

Given a graph $G = (V, E, \mathit{src}, \mathit{tar})$, we define $\mathit{hull}\colon \mathit{pow}(V) \to \mathit{pow}(V)$ by

$$\mathit{hull}(U) = \{\mathit{tar}(e) \mid e \in E,\ \mathit{src}(e) \in U^*\}$$

for all $U \subseteq V$. Hence, $\mathit{hull}(U)$ yields the set of all vertices that can be reached in one step from edges all of whose sources are in U. In particular, $\mathit{hull}(\emptyset)$ is the set of vertices that are targets of edges without sources (the *leaves*). Furthermore, we let $\mathit{hull}^{\leq 0}(U) = U$ and $\mathit{hull}^{\leq n+1}(U) = \mathit{hull}^{\leq n}(\mathit{hull}(U) \cup U)$ for $n \in \mathbb{N}$.

Definition 2 (Layered graph). A graph G is *layered* if (a) $\mathit{hull}^{\leq n}(\emptyset)$ is finite for every $n \in \mathbb{N}$, (b) for every vertex $v \in V$ there are only finitely many paths π such that $\mathit{tar}(\pi) = v$, and (c) $V = \mathit{tar}(E)$.

In particular, layered graphs are acyclic by requirement (b). Requirement (c) can be guaranteed by attaching to every vertex that is not the target of any edge a dummy edge having the vertex in question as a target and no sources. However, this is only possible if there are only finitely many such vertices, because otherwise it would make $\mathit{hull}(\emptyset)$ infinite, thus violating (a). Requirement (c) makes sure that all of G can gradually be built up from the "bottom" by starting with the empty set of vertices and repeatedly applying *hull*, thus following edges upward from children to their parents. In this process requirement (a) makes sure that the subgraph obtained always stays finite.

Example 2 (Layered graph). Definition 2 identifies a class of graphs for which weight functions can be conveniently defined, and which can be searched for optimal vertices by starting at the leaves and gradually expanding the search area using the operation *hull*. Not all graphs are of this kind. Suppose for instance that we have a non-trivial algebra and consider the graph in which every term in this algebra is a vertex, and in which there is an edge from a term t to a term s if t is a proper subterm of s. Clearly, every vertex has infinitely many outgoing edges, so condition (a) of Definition 2 is not satisfied. However, as every term has finitely many subterms, condition (b) is fulfilled. (Condition (c) is not satisfied either, but this could be remedied in the way mentioned above.)

Consider then the graph in which the set of vertices is $\{u_0, u_1, u_2, \dots\} \cup \{v_0, v_1, v_2, \dots\}$ and where there are edges from u_i to u_{i+1}, from v_{i+1} to v_i, and

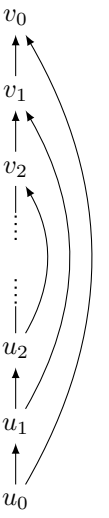


Figure 1: A graph that satisfies condition (a) but not condition (b)

from u_i to v_i, for every $i \in \mathbb{N}$ (see Figure 1). Thus, u_0 is the unique leaf and v_0 is the unique end. Since no vertex has more than two outgoing edges, the hull of any finite set of vertices can be nothing but a finite set. As $\mathit{hull}(\emptyset)$ is finite as well, condition (a) of Definition 2 is fulfilled. On the other hand, every vertex v_i can be reached from every u_j such that $j \geq i$ and is, hence, the target of infinitely many paths. Thus condition (b) is not met. In a weighted setting such as the one developed below, such graphs do not seem to admit reasonably defined weights unless the semiring is assumed to have infinite sums.

Finally, suppose that we have a non-deterministic Turing machine M and a fixed initial configuration c_0. We consider the graph in which the vertices are the pairs (c, n) such that c is a configuration of M that is reachable from c_0 in n steps. There is an edge from (c, n) to $(c', n+1)$ if c' can be reached in one step of M from c. Since the number of outgoing edges leaving each vertex is finite and the number of configurations reached in at most n steps is also finite, both condition (a) and (b) are satisfied. If we, in addition, mark $(c_0, 0)$ with a sourceless edge, condition (c) is fulfilled as well.

Let $\mathcal{A} = (A, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ be a semiring. A *graph with weights in* $\mathcal{A}$, also called a *weighted graph*, is a tuple $G = (V, E, \mathit{src}, \mathit{tar}, \mathit{wgt})$ such that $(V, E, \mathit{src}, \mathit{tar})$ is a layered graph and $\mathit{wgt} \colon E \to \mathcal{A}$ is its *weight assignment*. The induced *weight function* $w \colon V \cup E \to A$ determined by the following conditions:

- $w(v) = \bigoplus_{e \in \mathit{tar}^{-1}(v)} w(e)$ for every vertex v and
- $w(e) = \mathit{wgt}(e) \otimes \bigotimes_{i \in [n]} w(v_i)$ for every edge e, where $\mathit{src}(e) = v_1 \cdots v_n$.

To see that w is uniquely determined, let $\ell(v) = 2m$ for every $v \in V$, where m is the maximum length of all paths π such that $\mathit{tar}(\pi) = v$. Due to

conditions (b) and (c) of Definition 2, the set of those paths is nonempty and finite, and hence $\ell(v)$ is well defined. For an edge e, let $\ell(e) = \ell(tar(e)) - 1$. Now, induction on $\ell(x)$ uniquely determines $w(x)$ for all $x \in V \cup E$, as follows. For $v \in V$, if $e_1, \ldots, e_m$ are the (finitely many!) pairwise distinct edges having v as their target, then $\ell(e_i) < \ell(v)$ for all $i \in [m]$. Hence, the induction hypothesis yields that $w(e_i)$ is uniquely determined for every $i \in [m]$, which determines $w(v) = \bigoplus_{i \in [m]} w(e_i)$. Similarly, for every edge e with $src(e) = v_1 \cdots v_n$, we have $\ell(v_i) < \ell(e)$ and thus, by the induction hypothesis, $w(v_i)$ is uniquely determined for every $i \in [n]$, which gives us $w(e) = wgt(e) \otimes \bigotimes_{i \in [n]} w(v_i)$.

For $m, m' \in \mathbb{N}^k$, we let $m \leq m'$ if $m(i) \leq m'(i)$ for all $i \in [k]$. An element $m \in M$ of a finite set $M \subseteq \mathbb{N}^k$, is *minimal* if there is no $m' \in M$ such that $m' < m$. The subset of M of minimal elements is denoted $min(M)$.

Given a semiring $\mathcal{A} = (A, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ and $k \in \mathbb{N}$, we denote by $\mathcal{A}^k$ the semiring whose domain is A^k and in which $\oplus$ and $\otimes$ are applied element-wise and zero and one are the vectors $\mathbb{0}^k = (\mathbb{0}, \ldots, \mathbb{0})$ and $\mathbb{1}^k = (\mathbb{1}, \ldots, \mathbb{1})$, respectively. It is easy to verify that $\mathcal{A}^k$ is a semiring, and that $\mathcal{A}^k$ is nice if $\mathcal{A}$ is nice.

3. Computing N-Best Vertices

Let us now define the N-best vertices problem. As is common for many search applications where N-best problems are of interest (for example, shortest path, maximum likelihood, least weight) one may restrict oneself to semirings that are idempotent and choose their natural order as the order to be considered. If the natural order of such an idempotent semiring $\mathcal{A}$ is furthermore well-founded, we say that $\mathcal{A}$ is *well-founded*.

Definition 3 (N-best vertices). The *N-best vertices problem* is defined as follows. An instance is a pair (G, V^T, N) consisting of

- a layered weighted graph $G = (V, E, src, tar, wgt)$ with weights in a well-founded semiring $\mathcal{A}$,
- a set of *target vertices* $V^T \subseteq V$ such that each $v \in V^T$ is an end, and
- an integer $N \in \mathbb{N}^\infty$ such that $N \leq \left|V^T\right|$.

A solution is a sequence of N pairwise distinct elements $v_1, v_2, \cdots$ of V^T such that there do not exist $i \in [N]$ and $v \in V^T \setminus \{v_1, \ldots, v_i\}$ with $w(v) <_{\mathcal{A}} w(v_i)$.

The requirement that each $v \in V^T$ is an end deserves commenting on. The N-best vertices problem could equally well be formulated for arbitrary sets of target vertices. However, in this case there could be $v, v' \in V^T$ such that $v \in hull^{\leq i}(\emptyset)$ and $v' \in hull^{\leq j}(\emptyset) \setminus hull^{\leq i}(\emptyset)$ with $i < j$ but $w(v) > w(v')$. The algorithm devised below would then become more involved, as it would have to account for the possibility to visit v on its way to discovering and outputting v', later on potentially having to reconsider v in order to output v itself. With each vertex in V^T being an end, there are fewer cases to consider.

The restriction of target vertices to ends is not severe, but mainly serves to simplify the presentation. If a vertex $v \in V^T$ violates the restriction, one may modify G by adding a new vertex $\hat{v}$ and a single edge with weight $\mathbb{1}$ (i.e., the multiplicative unit) pointing from v to $\hat{v}$, and then replacing v by $\hat{v}$ in V^T. This rewriting process essentially creates new 'dummy' vertices that are ends, to replace target vertices that are not. Clearly, if $\hat{v}_1, \ldots, \hat{v}_N$ is a solution to the modified problem instance, then $v_1, \ldots, v_N$ is a solution to the original one.

In practical instantiations of Definition 3, the graph G can be used to model the runs of an automaton, and V^T to identify accepting states. This is for instance the case in Section 4, where we compute the N best trees with respect to a weighted tree automaton.

We note here that Definition 3 does not specify how G and V^T are represented. Regarding V^T, it is only assumed that there is a procedure that computes the characteristic function of V^T, in other words, that it decides for a vertex $v \in V$ whether $v \in V^T$. As for the graph G, since it is layered we know that for each vertex v the subgraph G^v is finite, where G^v consists of all vertices and edges lying on a path π such that $tar(\pi) = v$. Below, we will add the assumption that there are procedures that construct G^v or subgraphs thereof, for certain vertices v. The algorithm will use this in order to construct the portion of G needed to solve the N-best vertices problem.

Before continuing, let us verify that the N-best vertices problem is well defined in the sense that a solution always exists. This is what the assumption of well-foundedness is needed for.

Lemma 1. *Every instance of the N-best vertices problem has a solution.*

Proof. Choose (not necessarily constructively) any element u_0 of V^T and build a sequence of vertices $u_0, u_1, u_2, \ldots$ in V^T with strictly decreasing weights. By assumption, the natural order of $\mathcal{A}$ is well-founded. Therefore, every such sequence is finite. Thus, the process eventually arrives at a vertex v_1 such that no vertex of strictly lesser weight exists in V^T. Now, fix v_1 and repeat the argument with $V^T \setminus \{v_1\}$ instead of V^T. Continue until N elements $v_1, \ldots, v_N$ have been found (or *ad infinitum* in case $N = \infty$). By construction, $v_1, \ldots, v_N$ is a solution. □

Not all idempotent semirings are well-founded; examples showing this are easy to construct. However, as we shall prove next, nice semirings are indeed well-founded.

Lemma 2. *Every nice semiring $\mathcal{A}$ is well-founded.*

Proof. The proof is divided into two parts. We first show that every element in $\mathcal{A}$ can be written as a sum of products of elements in a generating set $\{a_1, \ldots, a_k\}$. Due to idempotency and the definition of the natural order, the value of this sum is decided by its minimal summands, and these can be represented as vectors in $\mathbb{N}^k$. In the second part, we show that if there were an infinite strictly decreasing sequence of elements in $\mathcal{A}$, then there would also be

an infinite non-increasing sequence of vectors in $\mathbb{N}^k$. Dickson's lemma, which states that $(\mathbb{N}^k, \leq)$ is a wqo, ensures us that this is not possible.

Let $\mathcal{A}$ be generated by $\{a_1, \ldots, a_k\}$. For $m \in \mathbb{N}^k$ let $\varphi(m) = \bigotimes_{j \in [k]} a_j^{m(j)}$, and for a finite set $M \subseteq \mathbb{N}^k$ let $\Phi(M) = \bigoplus_{m \in M} \varphi(m)$, where $\Phi(\emptyset) = \mathbb{0}$. Then, since $\otimes$ distributes over $\oplus$, every $a \in A$ is represented by at least one finite set $M \subseteq \mathbb{N}^k$ in the sense that $a = \Phi(M)$. We call such a set a *representative* of a.

Let us now prove that the value of $\Phi(M)$ is determined by the minimal elements of M, i.e., that the following statement holds:

$$\text{For finite } M \subseteq \mathbb{N}^k,\ \Phi(M) = \Phi(min(M)). \tag{1}$$

Indeed, if $m, m' \in M$ are such that $m < m'$, then we have $\varphi(m') = \varphi(m) \otimes a$ for $a = \varphi(m' - m)$. Since $\mathcal{A}$ is monotonic, $\varphi(m) = \varphi(m) \otimes \mathbb{1} \leq_{\mathcal{A}} \varphi(m) \otimes a = \varphi(m')$ and thus $\varphi(m) \oplus \varphi(m') = \varphi(m)$ by the definition of $\leq_{\mathcal{A}}$. Every such non-minimal element $m' \in M$ can therefore be removed from M with no effect on $\Phi(M)$, which proves (1).

Suppose there is a descending sequence $s = b_0 >_{\mathcal{A}} b_1 >_{\mathcal{A}} b_2 >_{\mathcal{A}} \cdots$ and let M_i be a representative of b_i for all $i \in \mathbb{N}$. By the definition of the natural order, $b_i >_{\mathcal{A}} b_{i+1}$ implies that $\Phi(M_i \cup M_{i+1}) = \Phi(M_i) \oplus \Phi(M_{i+1}) = \Phi(M_{i+1})$, so we may assume that $M_0 \subseteq M_1 \subseteq \cdots$ and thus $M_0 \subsetneq M_1 \subsetneq \cdots$ because $b_i \neq b_{i+1}$ for all $i \in \mathbb{N}$. Now, pick $m_i \in min(M_i) \setminus min(M_{i-1})$ for all $i > 0$. Then $m_i \not\leq m_j$ for $0 < i < j$ since $m_i \leq m_j$ would imply $m_j \notin min(M_j) \setminus min(M_i)$. In other words, the sequence $m_1, m_2, \ldots$ is non-increasing and must thus be finite because $(\mathbb{N}^k, \leq)$ is a wqo. This shows that the sequence s is finite, and thus that $\mathcal{A}$ is well-founded. □

We shall now devise a simple algorithm that solves the N-best vertices problem. For the remainder of this section, we consider an instance (G, V^T, N), where $G = (V, E, src, tar, wgt)$, and take $\mathcal{A}$ to be nice. As mentioned earlier, we require that G can be explored effectively by means of a few procedures that represent G. What we provide is therefore a meta algorithm, in which different realisations of the procedures yield different instantiations of the overall algorithm. These procedures are as follows (where $u \in V$):

- There is a procedure that computes $hull(U)$ for every finite set $U \subseteq V$.
- The procedure $minWeight(u)$ returns a minimal element of the set of all $w(v)$ such that $v \in V^T$ and v is reachable from u. If no vertex in V^T is reachable from u, then an error element $\top$ is returned, which is considered to be larger than all elements of $\mathcal{A}$. Note that $minWeight(u)$ is well-defined owing to the well-foundedness of $\mathcal{A}$ (cf. the proof of Lemma 1).
- If $minWeight(u) \neq \top$ then $bestAncestor(u)$ returns some $v \in V^T$ with $w(v) = minWeight(u)$ that is reachable from u.
- Finally, $descendants(u)$ returns the set of vertices v of which u is an ancestor (including u itself). Note that this set is finite as G is layered.

Algorithm 1 Solving the generic N-best vertices problem

```
1: procedure BestVertices(G, V^T, N)
2:     U ← ∅;
3:     for i = 1, ..., N do
4:         H ← hull(U) \ U;
5:         select u ∈ H such that minWeight(u) is minimal;
6:         v ← bestAncestor(u);
7:         output v;
8:         U ← U ∪ descendants(v)
9:     end for
10: end procedure
```

The pseudocode of our algorithm is given in Algorithm 1. We now show that it is correct.

Lemma 3. *After $i \leq N$ executions of the loop body in Algorithm 1, it will have outputted i distinct vertices $v_1, \ldots, v_i \in V^T$ such that*

(1) there are no $v \in V^T \setminus \{v_1, \ldots, v_i\}$ and $j \in [i]$ such that $v <_{\mathcal{A}} v_j$, and
(2) $U = \bigcup_{j \in [i]} descendants(v_j)$.

Proof. We proceed by induction on i. For $i = 0$ the assertions are trivially true as U is initialised to $\emptyset$. Thus, assume that (1) and (2) hold for some $i - 1 < N$ and consider the ith execution of the loop. Let $H^{\uparrow}$ be the set of all vertices that are reachable from some vertex in $H = hull(U) \setminus U$. Thus, $H^{\uparrow}$ is the set of vertices the algorithm chooses v_i from. We show that exactly those vertices not yet in U are reachable from a vertex in H, i.e. that $H^{\uparrow} = V \setminus U$.

By (2) all descendants of every $u \in U$ are in U as well. But $H \cap U = \emptyset$, which implies that no vertex in U is reachable from H, i.e., that $U \cap H^{\uparrow} = \emptyset$. (If a vertex $u \in U$ would be reachable from $h \in H$ then $h \in descendants(u) \subseteq U$, contradicting the fact that $H \cap U = \emptyset$.) To show that $H^{\uparrow} = V \setminus U$, it thus remains to check that $v \in H^{\uparrow}$ for all $v \in V \setminus U$. As G is layered, the set Π of all paths π that satisfy $tar(\pi) = v$ is nonempty (by condition (c)) and finite (by condition (b)). It follows that one can construct a path $\pi = e_1 \cdots e_n \in \Pi$ such that $tar(e_1) \notin U$ and $[src(e_1)] \subseteq U$. (Start with any path of length 1 in Π. If the current path is $e_1 \cdots e_n$ but there is $u \in [src(e_1)] \setminus U$, extend it to $e_0 e_1 \cdots e_n$ where $tar(e_0) = u$; note that e_0 exists by condition (c), and that this process must eventually stop by condition (b).) Thus $tar(e_1) \in H$, which proves that $v \in H^{\uparrow}$. This proves that, indeed, $H^{\uparrow} = V \setminus U$. In particular, since all vertices in V^T are ends,

$$V^T \setminus \{v_1, \ldots, v_{i-1}\} = V^T \setminus \bigcup_{j \in [i]} descendants(v_j) \subseteq V \setminus U = H^{\uparrow} \ .$$

Since a solution does exist, this means that there is a $v_i \in (V^T \cap H^{\uparrow}) \setminus U$ of minimal weight, which implies that $\{v_1, \ldots, v_i\}$ satisfies (1). It follows that

Line 5 assigns u this vertex v_i (or another vertex in V^T of the same weight), that Line 6 assigns v the vertex $\mathit{bestAncestor}(u)$, and that Line 7 outputs v. Thus, (2) is now satisfied for i. □

It may be worthwhile noting that, since the set U in Algorithm 1 is always of the form $\bigcup_{j\in[i]} \mathit{descendants}(v_j)$ for the already outputted ends $v_1, \ldots, v_i$, the procedures $\mathit{hull}(U)$, $\mathit{minWeight}(u)$, and $\mathit{bestAncestor}(u)$ only need to be implemented for this special case.

4. Application to Weighted Tree Automata

Let us now discuss in which sense Algorithm 1 generalises the N-best algorithm in [2] for weighted tree automata (wta) over the min-plus semiring $\mathcal{T} = (\mathbb{R}_+^\infty, \min, +, \infty, 0)$ (also called the *tropical semiring*). To regain the original algorithm in [2] from Algorithm 1, the auxiliary procedures *hull*, *minWeight*, *bestAncestor*, and *descendants* can be realised in the domain of computation graphs generated by wta. For this it is useful to recall parts of the theoretical framework of [2].

4.1. Weighted Tree Automata

For a set Σ, a Σ-labelled *tree* is a partial function $t\colon \mathbb{N}^* \to \Sigma$ such that the domain $\mathit{dom}(t)$ of t is a finite prefix-closed set, and for every $v \in \mathit{dom}(t)$ there exists a $k \in \mathbb{N}$ such that $\{i \in \mathbb{N} \mid vi \in \mathit{dom}(t)\} = [k]$. An element v of $\mathit{dom}(t)$ is called a *vertex of* t, and k is the *rank of* v.

A *ranked alphabet* is a finite nonempty set of symbols $\Sigma = \bigcup_{k\in\mathbb{N}} \Sigma_{(k)}$ which is partitioned into pairwise disjoint subsets $\Sigma_{(k)}$. For every $k \in \mathbb{N}$ and $f \in \Sigma_{(k)}$, the *rank* of f is $\mathit{rank}(f) = k$. The set $\mathbb{T}_\Sigma$ of all trees over Σ consists of all Σ-labelled trees t such that the rank of every vertex $v \in \mathit{dom}(t)$ equals the rank of $t(v)$.

The *subtree* of $t \in \mathbb{T}_\Sigma$ rooted at v is the tree t/v defined by $\mathit{dom}(t/v) = \{u \in \mathbb{N}^* \mid vu \in \mathit{dom}(t)\}$ and $t/v(u) = t(vu)$ for every $u \in \mathbb{N}^*$. If $t(\lambda) = f$ and $t/i = t_i$ for all $i \in [k]$, where k is the rank of λ in t, then we denote t by $f[t_1, \ldots, t_k]$. If $k = 0$, then $f[]$ is usually abbreviated as f. In other words, a tree t with domain $\{\lambda\}$ is identified with $t(\lambda)$.

For a set T of trees we denote by $\Sigma(T)$ the set of trees

$$\{f[t_1, \ldots, t_k] \mid k \in \mathbb{N}, f \in \Sigma_{(k)}, \text{ and } t_1, \ldots, t_k \in T\} \ .$$

Thus, $\Sigma(T)$ consists of all trees which have a symbol from Σ at their root, with direct subtrees in T.

We let $\square \notin \Sigma$ be a special symbol of rank 0. The set of *contexts over* Σ is

$$C_\Sigma = \{c \in \mathbb{T}_{\Sigma\cup\{\square\}} \mid \text{ there is exactly one } v \in \mathit{dom}(c) \text{ with } c(v) = \square\} \ .$$

Given a context $c \in C_\Sigma$, where $v \in \mathit{dom}(c)$ is the unique vertex such that $c(v) = \square$, we define the *substitution* of a tree t into c to be the tree $c[\![t]\!]$ given

as follows: $dom(c[\![t]\!]) = dom(c) \cup \{vu \mid u \in dom(t)\}$ and, for $w \in dom(c[\![t]\!])$,

$$c[\![t]\!](w) = \begin{cases} c(w) & \text{if } w \in dom(c) \setminus \{v\}, \text{ and} \\ t(u) & \text{if } w = vu \text{ for some } u \in dom(t) \end{cases} .$$

A *weighted tree language* (over the min-plus semiring $\mathcal{T}$) is simply a mapping $\mathcal{W}\colon \mathbb{T}_\Sigma \to \mathcal{T}$, where Σ is a ranked alphabet. A *weighted tree automaton* over $\mathcal{T}$ is a system $M = (Q, \Sigma, \delta, \rho)$ where

- Q is a ranked alphabet of *states*, all of rank 0, called *states*;
- Σ is a ranked alphabet of *input symbols* disjoint with Q;
- $\delta\colon \Sigma(Q) \times Q \to \mathcal{T}$ is the *transition function*; and
- $\rho\colon Q \to \mathcal{T}$ is the assignment of *final weights*.

For $q \in Q$, δ inductively gives rise to a function $\delta_q\colon \mathbb{T}_\Sigma \to \mathcal{T}$, as follows: For all $t = f[t_1, \ldots, t_k] \in \mathbb{T}_\Sigma$

$$\delta_q(t) = \min_{q_1,\ldots,q_k \in Q} \left(\delta(f[q_1, \ldots, q_k], q) + \sum_{i \in [k]} \delta_{q_i}(t_i) \right) . \tag{2}$$

Now, the weighted tree language $\mathcal{W}_M\colon \mathbb{T}_\Sigma \to \mathcal{T}$ recognised by M is given by

$$\mathcal{W}_M(t) = \min_{q \in Q} \ (\rho(q) + \delta_q(t)) \text{ for all } t \in \mathbb{T}_\Sigma .$$

We extend δ_q, and thus $\mathcal{W}_M$ to $\mathbb{T}_{\Sigma \cup Q}$ by defining $\delta_q(q) = 0$ and $\delta_q(q') = \infty$ for all $q' \in Q \setminus \{q\}$. It follows that, for all trees t and all contexts c,

$$\mathcal{W}_M(c[\![t]\!]) = \min_{q \in Q} \ (\mathcal{W}_M(c[\![q]\!]) + \delta_q(t)) . \tag{3}$$

For the sake of completeness, and for later reference, let us now recall the N-best trees problem for this device. An instance of this problem is a pair (M, N) consisting of a wta $M = (Q, \Sigma, \delta, \rho)$ over $\mathcal{T}$ and an integer $N \in \mathbb{N}^\infty$. A solution is a sequence of N pairwise distinct trees $t_1, t_2, \cdots$ such that there do not exist $i \in [N]$ and $t \in \mathbb{T}_\Sigma \setminus \{t_1, \ldots, t_i\}$ with $\mathcal{W}_M(t) <_\mathcal{T} \mathcal{W}_M(t_i)$. To make sure that a solution always exists, we may for simplicity assume that $\Sigma \neq \Sigma_{(0)} \neq \emptyset$, in other words, that $\mathbb{T}_\Sigma$ is an infinite set.

4.2. The Computation Graph of a Weighted Tree Automaton

Let $M = (Q, \Sigma, \delta, \rho)$ be a wta over $\mathcal{T}$. In the following, we abbreviate $Q \times \mathbb{T}_\Sigma$ by $Q\langle\mathbb{T}_\Sigma\rangle$ and a pair $(q, t) \in Q\langle\mathbb{T}_\Sigma\rangle$ by $q\langle t\rangle$. The weighted graph $G_M = (V, E, src, tar, wgt)$ associated with M, which also has weights in $\mathcal{T}$, is given by the following components:

- $V = Q\langle\mathbb{T}_\Sigma\rangle \cup \mathbb{T}_\Sigma$,

- For every $t = f[t_1, \ldots, t_k] \in \mathbb{T}_\Sigma$ and all $q, q_1, \ldots, q_k \in Q$, there is an edge $e \in E$ with $src(e) = q_1\langle t_1\rangle \cdots q_k\langle t_k\rangle$, $tar(e) = q\langle t\rangle$, and $wgt(e) = \delta(f[q_1, \ldots, q_k], q)$. Below, we denote this edge e by $e^t_{q_1 \cdots q_k, q}$.
- Further, for every $q \in Q$ and $t \in \mathbb{T}_\Sigma$, there is an edge e such that $src(e) = q\langle t\rangle$, $tar(e) = t$, and $wgt(e) = \rho(q)$.

It is straightforward to show that G_M is layered. The ends of G_M are the vertices in $\mathbb{T}_\Sigma$, and thanks to the edges of the second kind we have $w(t) = \min_{q \in Q}(\rho(q) + w(q\langle t\rangle))$ for every tree $t \in \mathbb{T}_\Sigma$. Thus, in order to establish the desired equality $w(t) = \mathcal{W}_M(t)$ we need to verify that $w(q\langle t\rangle) = \delta_q(t)$. We prove this by induction on $t = f[t_1, \ldots, t_k]$. Thus, assume that the claim holds for all trees smaller than t. Then the definition of the weight function of G_M (on both vertices and edges) yields

$$\begin{aligned} w(q\langle t\rangle) &= \min_{q_1,\ldots,q_k \in Q} w(e^t_{q_1 \cdots q_k, q}) \\ &= \min_{q_1,\ldots,q_k \in Q} \left(wgt(e^t_{q_1 \cdots q_k, q}) + \sum_{i \in [k]} w(q_i\langle t_i\rangle) \right) \\ &= \min_{q_1,\ldots,q_k \in Q} \left(\delta(f[q_1, \ldots, q_k], q) + \sum_{i \in [k]} \delta_{q_i}(t_i) \right) \\ &= \delta_q(t) \end{aligned}$$

as required. We have thus shown the following theorem:

Theorem 1. *Let $M = (Q, \Sigma, \delta, \rho)$ be a wta over $\mathcal{T}$ and $N \in \mathbb{N}^\infty$. Then a solution to the N-best vertices problem for $(G_M, \mathbb{T}_\Sigma, N)$ is also a solution to the N-best trees problem for (M, N).*

4.3. Specialisation of Algorithm 1

Since the min-plus semiring $\mathcal{T}$ is extremal, it is idempotent, and $\mathbb{1} = 0$ is its smallest element. However, it is not nice because it is not finitely generated. Upon closer consideration, we see that the subset $\mathcal{F}$ of $\mathcal{T}$ that is actually used in the computations of M is generated by 0, 1, the set $\{\rho(q) \mid q \in Q\}$ of final weights, and the set $\{\delta(f[q_1, \ldots, q_k], q) \mid k \in \mathbb{N}, f \in \Sigma_{(k)}, q, q_1, \ldots, q_k \in Q\}$ of edge weights. In other words, $\mathcal{F}$ is finitely generated. This makes Algorithm 1 applicable to $(G_M, \mathbb{T}_\Sigma, N)$, and it follows from Theorem 1 that the result of running it on $(G_M, \mathbb{T}_\Sigma, N)$ is a solution to the N-best trees problem for (M, N). What remains to obtain a concrete algorithm is to provide efficient implementations of *hull*, *minWeight*, *bestAncestor*, and *descendants*.

For simplicity, let us assume that there are at least N trees of finite weight. We give only a rough description of how Algorithm 1 can be specialised to this case, focusing on the main points rather than reiterating all the arguments from [2].

For the sake of efficiency, an explicit search of $H = \mathit{hull}(U) \setminus U$ should be avoided in line 5 of Algorithm 1. To make this easier, we can assume that $\mathcal{W}_M(s) = \infty$ for all proper subtrees s of trees of finite weight. As argued in [2] this is easy to achieve by adding a special root symbol r to $\Sigma_{(1)}$. As a consequence we can simplify our further considerations by removing all ends (i.e., all vertices in $\mathbb{T}_\Sigma$) from H, as their weight is ∞ (due to the fact that t is a subtree of a tree of finite weight for all $q\langle t\rangle \in U$). In other words, from now on let $H = (\mathit{hull}(U) \setminus U) \cap Q\langle \mathbb{T}_\Sigma\rangle$.

The algorithm in [2] was made to run in polynomial time by exploiting the regular structure of wta computations. Consider a tree t, and suppose that we want to minimise $\mathcal{W}_M(c[\![t]\!]) = w(c[\![t]\!])$ over all $c \in C_\Sigma$. From Equation (3) we know that $w(c[\![t]\!]) = \mathit{min}_{q\in Q}\mathcal{W}_M(c[\![q]\!]) + \delta_q(t)$. Therefore, we precompute a set of contexts c_q such that, for each $q \in Q$, $\mathcal{W}_M(c_q[\![q]\!]) = \mathit{min}_{c\in C_\Sigma}\mathcal{W}_M(c[\![q]\!])$. As shown in [2] this can be done efficiently using an algorithm by Knuth [5]. Let then $w_q = \mathcal{W}_M(c_q[\![q]\!])$ for all $q \in Q$, a quantity that we may also precompute. Then it is very cheap to compute $\mathit{minWeight}(p\langle t\rangle) = \mathit{min}_{c\in C_\Sigma} w(c[\![t]\!]) = \mathit{min}_{q\in Q} w_q + \delta_q(t)$ (which is, in fact, independent of the state p). Moreover, if q is the state that minimises the rightmost expression, then $\mathit{bestAncestor}(p\langle t\rangle) = c_q[\![t]\!]$. This yields Algorithm 2. Note that the computation of $\delta_q(t)$ in Line 5 is

Algorithm 2 Solving the N-best trees problem

```
1: procedure BestVertices(M, N)
2:     U ← ∅;
3:     for i = 1, ..., N do
4:         H ← Q⟨Σ(U ∩ T_Σ)⟩ \ U;
5:         select q⟨t⟩ ∈ H such that w_q + δ_q(t) is minimal;
6:         s ← c_q[[t]];
7:         output s;
8:         U ← U ∪ {p⟨s'⟩ | p ∈ Q, s' ∈ S} ∪ S where S = {s/v | v ∈ dom(s)}
9:     end for
10: end procedure
```

inexpensive because the algorithm may, along with every tree $t \in \mathbb{T}_\Sigma$ encountered during the run of the algorithm, keep the vector of weights $\delta_q(t)$, $q \in Q$. When a tree t is encountered for the first time, the weight vectors of its subtrees are already available, which means that the weight vector of t is obtained by $|Q|$ applications of Equation (2), not requiring any recursion.

To enable a quick selection of $q\langle t\rangle$ in line 5 of Algorithm 2, the vertices in H can be maintained in a priority queue K, where the priority of an element $q\langle t\rangle$ is determined by $\Delta(q\langle t\rangle) = w_q + \delta_q(t)$, with lower-weighted vertices given precedence. Since $\mathit{minWeight}(q\langle t\rangle) = \mathit{min}_{p\in Q} w_p + \delta_p(t)$, it holds for the first queue element $q\langle t\rangle$ (though not in general) that $\Delta(q\langle t\rangle) = \mathit{minWeight}(q\langle t\rangle)$. Furthermore, this $\mathit{minWeight}(q\langle t\rangle)$ is minimal over all enqueued trees. The selection of an appropriate $q\langle t\rangle \in H$ thus boils down to a dequeueing operation,

and a subsequent removal of all remaining elements in $\{p\langle t\rangle \mid p \in Q\}$ from K.[2] The procedure *descendants* is easily implemented, as shown in Line 8.

Finally, let us discuss briefly the pruning technique introduced in [2] to make the algorithm run faster. The efficiency of the basic algorithm is less than optimal because the queue K usually contains many more elements than necessary if we populate it by all of H. The key observation is that, since we are only interested in outputting N trees, for each $q \in Q$ at most N of the most promising (i.e. lowest-weight) vertices $q\langle t\rangle \in U$ may indeed be needed. In other words, U can be pruned down by keeping, for every $q \in Q$, only a set of N vertices $q\langle t_1\rangle, \ldots, q\langle t_N\rangle \in U$. These must be chosen in such a way that $\sum_{i=1}^{N} w_q(t_i)$ is minimised. (If U contains fewer than N vertices $q\langle t\rangle$ for the state q in question, no pruning takes place.) The usage of pruning makes the algorithm much more efficient without affecting its correctness, leaving the non-pruned version superfluous for this particular specialisation of Algorithm 1.

We omit a correctness proof and the accompanying running time estimations because they carry over from [2] in a straightforward way. In particular, assuming that the input alphabet is either fixed or the maximum rank of its symbols is small compared to N, the time complexity of the resulting, pruning, algorithm is $O\big(N^2 \log N \cdot n \cdot m\big)$, where m and n are the number of transitions and the number of states of M, respectively. The complexity of the less efficient, non-pruning version has not been formally investigated. It is likely exponential unless particular care is taken in the graph-exploration step, that is, in the computation of what is called *hull* in the present work and *expand* in [2].

4.4. Experiments

An implementation of both the pruning and the non-pruning version of the N-best algorithm can be found in the *BestTrees* GitHub repository[3]. The distribution consists of the Java source code along with a number of wta examples. There is also a runnable `.jar` file of the latest version for convenience.

The experiments were run on a family of wtas, language equivalent with the wta M of Figure 2. Here, $\Sigma = \Sigma_{(0)} \cup \Sigma_{(2)}$ with $\Sigma_{(0)} = \{a, b\}$ and $\Sigma_{(2)} = \{\circ\}$. Let $\|t\|_\sigma$ ($\sigma \in \Sigma_{(0)}$) denote the number of occurrences of σ in $t \in T_\Sigma$, and let $\|t\|$ denote the total number of leaves of t. Then we have

$$M(t) = \begin{cases} \|t\| + min(\|t\|_a, \|t\|_b) & \text{if } \|t\| \text{ is even} \\ \infty & \text{otherwise .} \end{cases}$$

Algorithm 1 was first run on M itself, allowing N to range between 0 and 80. The average run times are summarised in Fig. 3 and, with a logarithmic scale, in Fig. 4. As expected, the pruning version is considerably more efficient. In fact, without pruning the algorithm exhibits an exponential behaviour. For the

[2] In [2], the priority queue would not contain the trees $q\langle t\rangle$, but only the trees $t \in \mathbb{T}_\Sigma$. This difference does not affect the result because, as noted above, $minWeight(q\langle t\rangle)$ and $bestAncestor(q\langle t\rangle)$ are independent of q.

[3] `https://github.com/tm11ajn/besttrees`

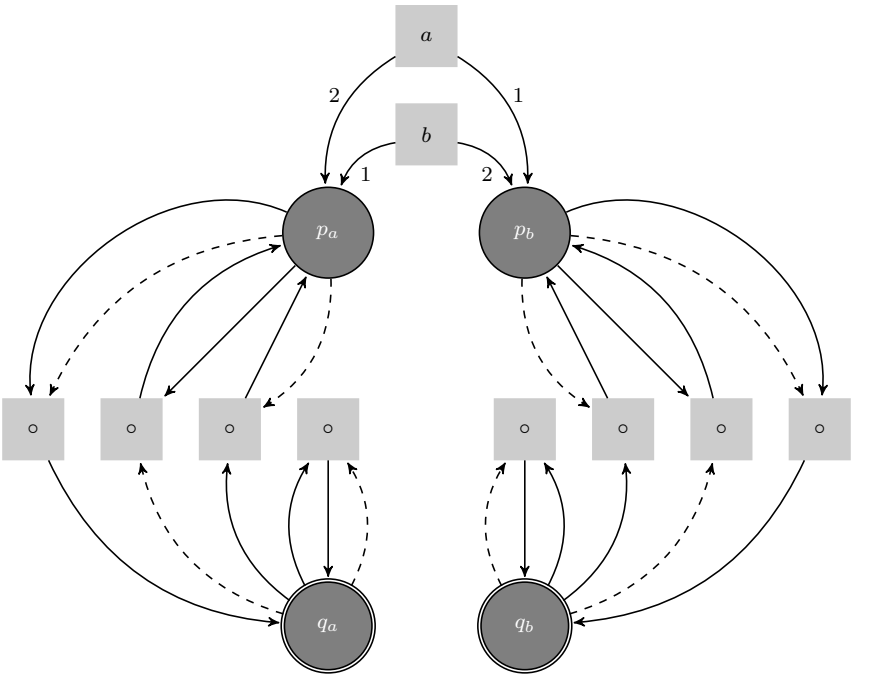


Figure 2: The input wta used in our experiments, taken from [2]. Round nodes (with double circles if final) represent states, and squares represent transitions. The consumed input symbols are shown inside the squares. Solid arcs point to the right-hand side of the transition in question and are labelled with the weight of the transition unless it is zero. In the case of input symbol ∘ the two states in the left-hand side of a transition are indicated by incoming solid and dashed arcs. Since the wta is symmetric, the latter distinction is, in fact, unnecessary.

pruning version, the run time remained below 10 seconds for the entire test series, whereas the running time of the non-pruning version exceeded a quarter of an hour for the higher values of N. While the theoretical worst case running time of the pruning version is $O(N^2 \log N)$ for a fixed wta, Fig. 4 indicates that it may be nearly linear in practice, at least in this particular case. This is strengthened by Fig. 5, which clearly shows the almost linear behaviour.

Further tests were then conducted on different-sized automata derived from M by introducing redundancy. More specifically, we added new states and transitions to count the height of subtrees up to a varying bound k. This means that for any two distinct automata M' and M'' thus obtained, the automaton for the smaller k is a subautomaton of the one for the larger k. The size of the largest of these automata, measured as $m + n$, was 1500. Again the value of N was varied, though now on the smaller range between 0 and 50, and the algorithm was run both with and without pruning. The results are shown in Figures 6 and 7. We see that the size of the wta does not matter significantly for the version of the algorithm that does not use pruning. This is not surprising since the same hulls are calculated during the execution of the

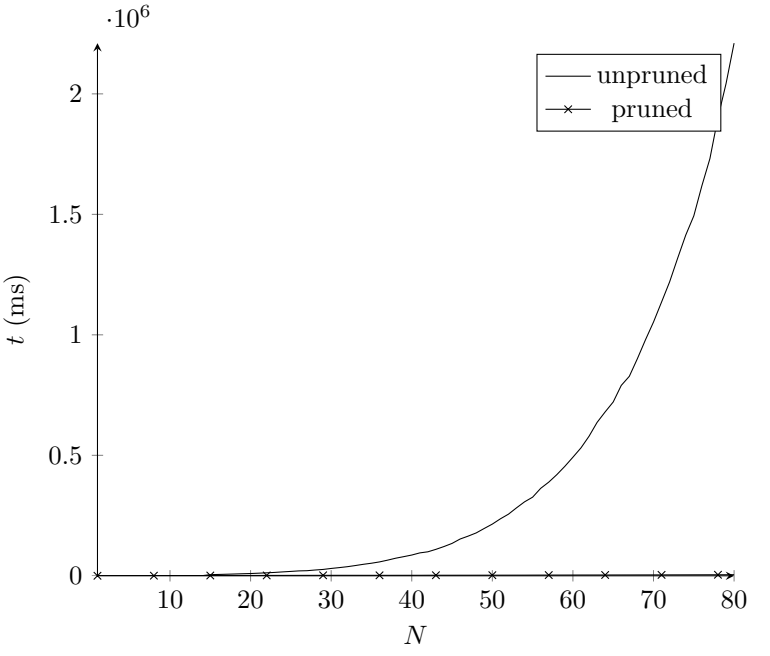


Figure 3: The running time of both versions of the algorithm applied to the wta in Fig. 2 for different values of N.

algorithm, regardless of the size of the wta (see footnote 2), and the number of iterations is determined solely by N. As with the initial tests, the run time for the pruning version was below 10 seconds on all instances, whereas the version without pruning took minutes when N was close to 50.

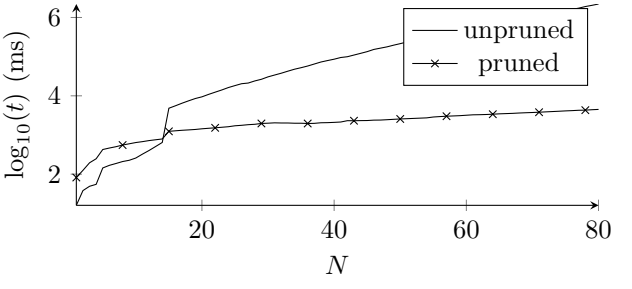


Figure 4: The running time of both versions of the algorithm applied to the wta in Fig. 2 for different values of N on a logarithmic scale.

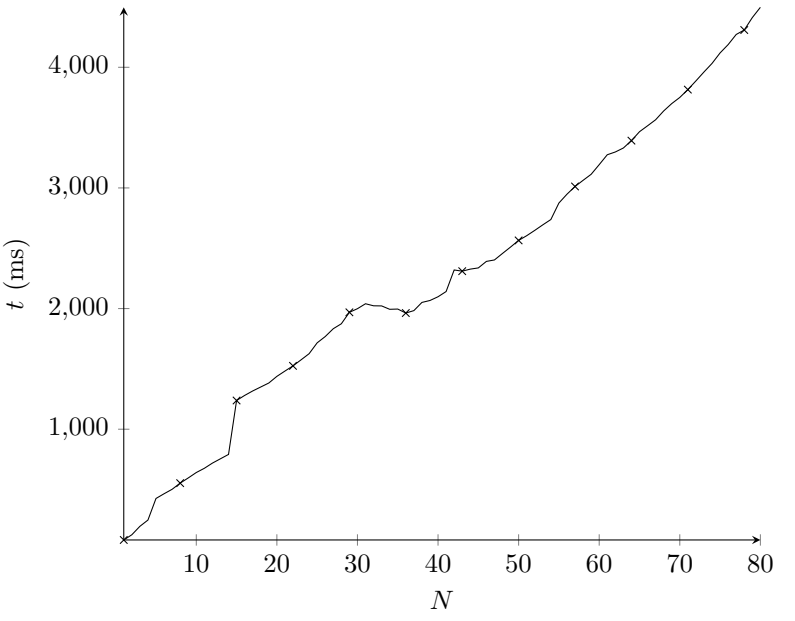


Figure 5: The running time of the pruning version of the algorithm applied to the wta in Fig. 2 for different values of N.

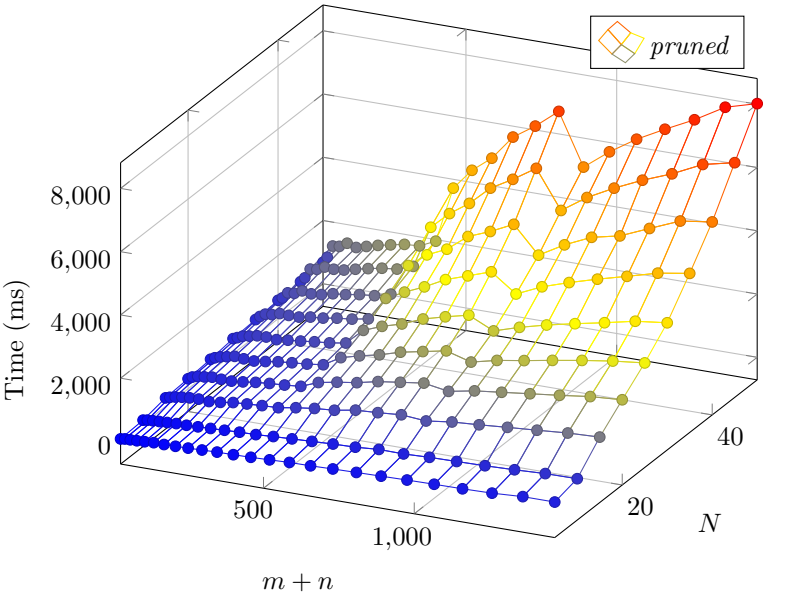


Figure 6: The running time of the pruned version of the algorithm for different values of N on different sized wtas (measured in $m+n$).

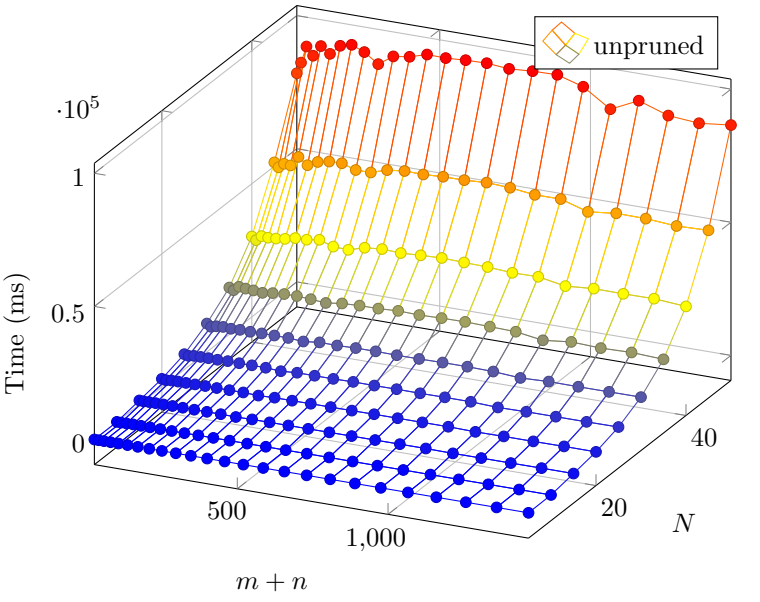


Figure 7: The running time of the unpruned version of the algorithm for different values of N on different sized wtas (measured in $m+n$).

5. Conclusion and Future Work

We have defined the N-best problem for layered graphs with edge-weights in a nice semiring. Whenever the graph representation permits the procedures *hull*, *minWeight*, *bestAncestor*, and *descendants* to be (efficiently) implemented, so can our N-best algorithm. As we have seen, this is the case for weighted tree automata over the min-plus semiring. Our initial experiments on this restricted domain indicated furthermore that the pruning technique proposed in [2], which carries over to the setting of the present paper, has the expected positive impact on the running time of the algorithm.

To understand the practical applicability of Algorithm 1 to problems in natural language processing, a study on real-world data is needed. Rather than using the non-pruning version of the algorithm as a baseline, it seems reasonable to use the state-of-the-art-approach, which is computing the N' best runs for some $N' \gg N$ and discarding duplicates. On the theoretical side, we would like to weaken the assumptions on the underlying graph and semiring. In particular, it would be interesting to see whether there is a class of semirings beyond the min-plus semiring which allows for an efficient implementation of the procedures *hull*, *minWeight*, *bestAncestor*, and *descendants* in the case of wta, thus yielding efficient solutions to the N-best trees problem based on Theorem 1.

A potentially rewarding line of future work is the instantiation of Algorithm 1 to work with less traditional types of objects, for example pictures. Suppose that we are given an algebra consisting of a domain of pictures and a set Σ

of operations on them.[4] Under suitable well-definedness conditions, a wta M over Σ then defines a weighted picture language, the weight of a picture P being the sum of all $\mathcal{W}_M(t)$ such that t evaluates to P. (Alternative definitions are possible as well, such as defining the weight of P to be equal to $\mathcal{W}_M(t)$, and requiring that different trees representing the same picture yield the same weight.) Then the nodes of the input graph to the N-best vertices problem would be pictures rather than trees, and the output would be a set of N best pictures. Future work could try to make this rough idea more precise, in order to discover solvable variants of the N-best pictures problem.

Acknowledgement

We thank Andreas Maletti and the participants at the *4th International Workshop on Trends in Tree Automata and Tree Transducers (TTATT 2016)*, where a preliminary and abridged version of the paper was presented, for their insightful comments that led to various improvements. Last but by no means least, we are grateful to the anonymous reviewers for pointing out flaws and suggesting significant improvements in various places.

[1] Johanna Björklund, Frank Drewes, and Niklas Zechner. An efficient best-trees algorithm for weighted tree automata over the tropical semiring. Report UMINF 14.22, Umeå University, 2014.

[2] Johanna Björklund, Frank Drewes, and Niklas Zechner. An efficient best-trees algorithm for weighted tree automata over the tropical semiring. In *Proc. 9th Intl. Conf. on Language and Automata Theory and Applications (LATA 2015)*, volume 8977 of *LNCS*, pages 97–108, 2015.

[3] Matthias Büchse, Daniel Geisler, Torsten Stüber, and Heiko Vogler. n-best parsing revisited. In *Proceedings of the 2010 Workshop on Applications of Tree Automata in Natural Language Processing*, pages 46–54, Uppsala, Sweden, July 2010. Association for Computational Linguistics.

[4] Frank Drewes. *Grammatical Picture Generation – A Tree-Based Approach.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[5] Donald E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6:1–5, 1977.

[6] Mehryar Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

[7] Mehryar Mohri and Michael Riley. An efficient algorithm for the n-best-strings problem. In *Proceedings of the Conference on Spoken Language Processing*, 2002.

[4] See [4] for various types of such picture algebras.

II

A Comparison of Two N-Best Extraction Methods for Weighted Tree Automata

Johanna Björklund, Frank Drewes, and Anna Jonsson( )

Department of Computing Science, Umeå University, Umeå, Sweden
{johanna,drewes,aj}@cs.umu.se

Abstract. We conduct a comparative study of two state-of-the-art algorithms for extracting the N best trees from a weighted tree automaton (wta). The algorithms are Best Trees, which uses a priority queue to structure the search space, and Filtered Runs, which is based on an algorithm by Huang and Chiang that extracts N best runs, implemented as part of the Tiburon wta toolkit. The experiments are run on four data sets, each consisting of a sequence of wtas of increasing sizes. Our conclusion is that Best Trees can be recommended when the input wtas exhibit a high or unpredictable degree of nondeterminism, whereas Filtered Runs is the better option when the input wtas are large but essentially deterministic.

1 Introduction

Data-driven language processing involves as a rule weighted language models. Rather than providing a definite answer as to whether a sentence belongs to a target language, these return a probability or a fitness score. This reflects the inherently ambiguous nature of human language and is convenient for statistical machine learning, but it often complicates downstream processing. When the output of a machine translation system is not limited to a small set of possible translations, but is a weighted device ranking the universe of all possible outputs, efficient algorithms are needed to find the highest-scoring solutions. This problem is known as the N-best problem. The input is a weighted automaton M and a natural number N, and task is to find N best-ranking elements with respect to M. The difficulty of the problem, and indeed whether there is a unique or several interchangeable solutions, largely depends on the type of automata at hand and the domain from which weights are taken.

We consider the N-best problem for weighted tree automata [3,4], which are useful in natural language processing, owing to their capability to rank parse trees of context-free languages. This makes them useful for syntax-based forms of processing, as demonstrated in, e.g., machine translation and program verification. Weighted tree automata [2] are typically defined over algebras that have at least as much structure as a semiring, but semifields or even fields are often used. The weight of a computation (called a run) of an automaton on an input tree is the semiring product of the weights of the rules applied, and the weight

© Springer International Publishing AG, part of Springer Nature 2018
C. Câmpeanu (Ed.): CIAA 2018, LNCS 10977, pp. 97–108, 2018.
https://doi.org/10.1007/978-3-319-94812-6_9

of the tree is the semiring sum of the weights of all runs on this tree. We restrict ourselves to the so-called tropical semiring, which means that the weight of a run is the (ordinary) sum of the weights of the applied rules and the weight of a tree is the minimum of the weights of its runs.

Our main contribution is an empirical evaluation of two N-best algorithms for weighted tree automata (wta) M over the tropical semiring. Both algorithms represent the state of the art, but operate in quite different ways. The first of these is an indirect method based on an N-best runs (or derivations) algorithm proposed by Huang and Chiang [5] and implemented in the wta toolkit Tiburon [7]. The algorithm computes N best runs in time $O(mN \log N)$, where m is the number of transitions of the wta. This can be used to compute N best trees by generating a list of N' best runs of M, for some large enough $N' \geq N$. These runs are evaluated to the corresponding trees and duplicates are discarded to obtain a list of N best trees. Henceforth, we refer to this method by Filtered Runs. Filtered Runs thus takes a heuristic approach in the sense that an unlucky user may request too large or small a number of best runs, either wasting time or not gathering enough runs to find N unique best trees.

The second algorithm that we evaluate is Best Trees [1], a generalisation of an N-best algorithm for string automata [8]. Intuitively, Best Trees implements a breadth-first search, while making extensive use of pruning to avoid a combinatorial explosion. The running time of Best Trees is $O(N^2 \cdot (mn \log N + n^3))$ [1, slightly simplified], where m and n are the number of transitions and states of the input wta. Hence, the algorithm is less efficient than the pure N-best runs algorithm by Huang and Chiang, even though both are polynomial. However, Best Trees is guaranteed to produce exactly the desired number of trees without the need to discard duplicates, whereas Filtered Runs may require the enumeration of a large number of runs. The latter can happen if the input wtas exhibit a high degree of nondeterminism, i.e. if the number of distinct trees among the N best runs grows slowly (logarithmically in the worst case) with increasing N. Filtered Runs is thus expected to run faster even on large wtas if there is no or very little nondeterminism, while the asymptotic advantage of Best Trees should become apparent as the degree of nondeterminism increases. We perform empirical evaluations in order to (a) confirm this expected behaviour and (b) get an idea about how the algorithms compare on different kinds of wta and varying amounts of nondeterminism.

To study this in a setting where the type, size, and amount of nondeterminism of input wtas can be varied in a controlled way, we run our experiments on a range of synthesized wtas designed for this purpose rather than using "real life" wtas. While the results mostly confirm the theoretical expectations, they show that the precise behaviour is not as simple as the theoretical worst case analysis suggests. In particular, the running time of Filtered Runs depends on aspects other than the pure amount of nondeterminism, such as the order in which the transition rules of the input wta are given.

2 Preliminaries

We write $\mathbb{N}$ for the set of nonnegative integers, $\mathbb{N}_+$ for $\mathbb{N}\setminus\{0\}$, and $\mathbb{R}_+$ for the set of non-negative reals; $\mathbb{N}^\infty$ and $\mathbb{R}_+^\infty$ denote $\mathbb{N}\cup\{\infty\}$ and $\mathbb{R}_+\cup\{\infty\}$, respectively. For $n\in\mathbb{N}$, $[n]=\{i\in\mathbb{N}\mid 1\le i\le n\}$. Thus, in particular, $[0]=\emptyset$ and $[\infty]=\mathbb{N}$. The cardinality of a (countable) set S is written $|S|$, and the powerset of S is denoted by $pow(S)$. The n-fold Cartesian product of a set S with itself is denoted by S^n. As usual, the set of all finite sequences over S is denoted by S^*, and the empty sequence by λ.

For a set A, an A-labelled *tree* is a partial function $t\colon \mathbb{N}_+^*\to A$ whose domain $dom(t)$ is a finite non-empty set that is prefix-closed and closed to the left: whenever $vi\in dom(t)$ for some $v\in\mathbb{N}_+^*$ and $i\in\mathbb{N}_+$, it holds that $v\in dom(t)$ (prefix-closedness) and $vj\in dom(t)$ for all $1\le j\le i$ (closedness to the left). The size of t is $|t|=|dom(t)|$. An element v of $dom(t)$ is called a *node of* t, and $|\{i\in\mathbb{N}_+\mid vi\in dom(t)\}|$ is the *rank of* v. The *subtree* of $t\in T_\Sigma$ rooted at v is the tree t/v defined by $dom(t/v)=\{u\in\mathbb{N}_+^*\mid vu\in dom(t)\}$ and $t/v(u)=t(vu)$ for every $u\in\mathbb{N}_+^*$. If $t(\lambda)=f$ and $t/i=t_i$ for all $i\in[k]$, where k is the rank of λ in t, then we denote t by $f[t_1,\dots,t_k]$, which may be identified with f if $k=0$.

A *ranked alphabet* is a disjoint union of finite sets of symbols, $\Sigma=\bigcup_{k\in\mathbb{N}}\Sigma_{(k)}$. For $f\in\Sigma$, the $k\in\mathbb{N}$ such that $f\in\Sigma_{(k)}$ is the *rank* of f, denoted by $rank(f)$. The set T_Σ of ranked trees over Σ consists of all Σ-labelled trees t in which the rank of every node $v\in dom(t)$ equals the rank of $t(v)$. For a set T of trees we denote by $\Sigma(T)$ the set of trees which have a symbol from Σ in their root, with direct subtrees in T, i.e., $\{f[t_1,\dots,t_k]\mid k\in\mathbb{N}, f\in\Sigma_{(k)},\text{ and } t_1,\dots,t_k\in T\}$.

In the following, let $\Box\notin\Sigma$ be a special symbol of rank 0. The set of *contexts over* Σ is the set C_Σ of trees $c\in T_{\Sigma\cup\{\Box\}}$ containing exactly one node $v\in dom(c)$ with $c(v)=\Box$. The *substitution* of another tree t into c results in the tree $c[\![t]\!]$ given by $dom(c[\![t]\!])=dom(c)\cup\{vu\mid u\in dom(t)\}$ and

$$c[\![t]\!](w)=\begin{cases} c(w) \text{ if } w\in dom(c)\setminus\{v\}, \text{ and}\\ t(u) \text{ if } w=vu \text{ for some } u\in dom(t)\end{cases}$$

for all $w\in dom(c[\![t]\!])$.

Recall that the domain of the *tropical semiring* is $\mathbb{R}_+^\infty$, with min serving as addition and real-valued addition as multiplication. A *weighted tree language* over the tropical semiring is a mapping $L\colon T_\Sigma\to\mathbb{R}_+^\infty$, where Σ is a ranked alphabet. Such languages can be specified by a *weighted tree automaton with final states* (wta). A wta is a system $M=(Q,\Sigma,R,Q_\mathrm{f})$ consisting of

- a finite set Q of symbols of rank 0 called *states*;
- a ranked alphabet Σ of *input symbols* disjoint with Q;
- a finite set R of *transition rules* $f[q_1,\dots,q_k]\xrightarrow{w}q$, where $q,q_1,\dots,q_k\in Q$, $f\in\Sigma_{(k)}$, and $w\in\mathbb{R}_+$; and
- a set $Q_\mathrm{f}\subseteq Q$ of *final states*.

A transition rule $r\colon f[q_1,\dots,q_k]\xrightarrow{w}q$ will also be viewed as a symbol of rank k, so that R becomes a ranked alphabet. In addition, we view every state $q\in Q$

as a symbol of rank 0. We define the set $runs_M^q \subseteq T_{R \cup Q}$ of *q-runs of* M, their resulting trees $result_M(\rho)$, and their weights $wt_M(\rho)$ (for $\rho \in runs_M^q$) inductively, as follows:

1. For every state $q \in Q$, $q \in runs_M^q$ with $result_M(q) = q$ and $wt_M(q) = 0$.
2. For every $r\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$ in R and all $\rho_1 \in runs_M^{q_1}, \ldots, \rho_k \in runs_M^{q_k}$, $\rho = r[\rho_1, \ldots, \rho_k] \in runs_M^q$ with $result_M(\rho) = f[result_M(\rho_1), \ldots, result_M(\rho_k)]$, and $wt_M(\rho) = w + \sum_{i \in [k]} wt_M(\rho_i)$.

The set of *accepting* runs is $runs_M = \{runs_M^q \mid q \in F\}$.

Now, the weighted tree language $M\colon T_\Sigma \to \mathbb{R}_+^\infty$ *recognised by* M is given by

$$M(t) = \min\{wt_M(\rho) \mid \rho \in runs_M^q \text{ is accepting and } result_M(\rho) = t\}$$

for all $t \in T_\Sigma$ (where, by convention, $\min \emptyset = \infty$). In other words, $M(t)$ is the minimal weight of an accepting run of t. Note that we, by a slight abuse of notation, denote by M both the wta and the weighted tree language it computes.

For a wta M and an $N \in \mathbb{N}^\infty$ as input, the *N-best runs problem* is the problem to compute a sequence of N accepting runs of minimal weight according to M. More precisely, an algorithm solving the problem outputs a sequence $\rho_1, \rho_2, \ldots$ of N pairwise distinct accepting runs such that there are no $i \in [N]$ and $\rho \in runs_M \setminus \{\rho_1, \ldots, \rho_i\}$ with $wt_M(\rho) < wt_M(\rho_i)$. (If the total number N' of accepting runs is smaller than N, the algorithm only outputs N' runs.)

Similarly, the *N-best trees problem* asks to compute pairwise distinct trees $t_1, t_2, \ldots$ in T_Σ of minimal weight, i.e., such that there are no $i \in [N]$ and $t \in T_\Sigma \setminus \{t_1, \ldots, t_i\}$ with $M(t) < M(t_i)$.

3 Previous Work

The difference between the two N-best problems is that, in the nondeterministic case, distinct runs may result in the same tree. The wta toolkit Tiburon provides an implementation of the N-best runs algorithm by Huang and Chiang [5]. This yields an obvious procedure for solving the N-best trees problem: one simply computes N' best runs $\rho_1, \rho_2, \ldots$ for large enough N', and outputs those $result_M(\rho_i)$ for which $result_M(\rho_i) \notin \{result_M(\rho_1), \ldots, result_M(\rho_{i-1})\}$. This procedure is guaranteed to produce the desired result because any given tree has at most an exponential number of runs, which means that the next tree will be encountered after at most exponentially many steps. (If there are no more accepting runs one can simply continue to enumerate arbitrary ones of the remaining trees, whose weight will by definition be ∞.)

The N-best trees algorithm developed in [1] avoids the detour via N' best runs. We now give a short summary of the reasoning that leads to this algorithm. Let the size parameters of the input wta M be the following:

- m is the number $|R|$ of transition rules of M,
- n is the number of states, and

– r is the maximum rank of symbols.

The algorithm explores its search space by maintaining a priority queue K of trees that are candidates of output trees. The trees in the queue mark the frontier of the search space, the priority being determined primarily by the minimal value of $M(c[\![t]\!])$, where c ranges over all possible contexts. To determine this value, note that the definition of $M(t)$ works also for trees $t \in T_{\Sigma \cup Q}$. In particular, t can be of the form $c[\![q]\!]$, where c is a context and $q \in Q$. This is useful because of the following. For a run of the form $\rho[\![\rho']\!]$ where $\rho' \in runs_M^q$ we clearly have $wt_M(\rho[\![\rho']\!]) = wt_M(\rho) + wt_M(\rho')$. Hence, if we denote by M^q the wta obtained from M by replacing its set of final states by $\{q\}$, then

$$M(c[\![t]\!]) = \min_{q \in Q} (M(c[\![q]\!]) + M^q(t))$$

for all contexts c and all trees t.

As $M(c[\![p]\!])$ is independent of t, a context c that minimises it can be calculated in advance. Such a context c, which we call a *cheapest context*[1] of q and which is henceforth denoted by c_q, is thus a cheapest context into which a subtree t can be embedded in order to reach a final state, once the state q has been reached at the root of t. As was shown in [1], a family $(c_q)_{q \in Q}$ of cheapest contexts can efficiently be computed given M.

To solve the N-best trees problem, when looking at a tree t in the frontier of our search space we are, intuitively, interested in the tree $c[\![t]\!]$ that has the least possible weight. The smaller this weight is, the higher should the priority of t be. Clearly, when comparing trees with regard to this, c can be assumed to be one of the cheapest contexts c_p. Thus, our aim has to be to determine the state q that minimises the weight of $c_q[\![t]\!]$. We call such a state an *optimal state* for t, and denote it by $opt(t)$ (breaking ties arbitrarily). In the algorithm, optimal states can efficiently be computed in an incremental way as trees are assembled from subtrees, provided that a small amount of bookkeeping information is stored along with each tree.

Now, the N-best trees algorithm maintains data structures T and K, where

– T is a set of trees that have already been processed and
– K is a priority queue of trees in $\Sigma(T)$, the frontier of the search space.

The queue K initially contains the trees in Σ_0. Its priority order $<_K$ is defined by $t <_K t' \iff \Delta(t) < \Delta(t')$, where $\Delta(s) = M(c_{opt(s)}[\![s]\!])$ for all $s \in T_\Sigma$.

We reproduce the pseudocode of the base algorithm from [1] in Algorithm 1. As discussed in detail in [1], the set of trees enqueued in line 13 can be pruned because for every state q at most N trees for which q is an optimal state may ever become relevant. An additional optimisation was used in the implementation used for the experiments of this paper: once the algorithm has outputted $i \leq N$ trees, it suffices to keep $N - i$ rather than N trees in K for each optimal state. Hence, the queue shrinks as progress is made. While this does not affect the asymptotic running time, it does yield a significant improvement in practise.

[1] In [1] the term *smallest completion* was used.

Algorithm 1. Compute $N \in \mathbb{N}^\infty$ trees of minimal weight according to a wta M

```
1: procedures Best Trees(M, N)
2:     compute cheapest contexts for all states
3:     T ← ∅; K ← ∅
4:     enqueue(K, Σ_0)
5:     i ← 0
6:     while i < N ∧ K is nonempty do
7:         t ← dequeue(K)
8:         T ← T ∪ {t}
9:         if M(t) = Δ(t) then
10:            output(t)
11:            i ← i + 1
12:        end if
13:        enqueue(K, expand(T, t))
14:    end while
15: end procedures
```

4 Experiments

In this section, we experimentally verify the time complexity of Best Trees, and then compare its performance with the indirect method Filtered Runs.

It is easy to construct worst-case scenarios in which Best Trees works exponentially faster than Filtered Runs. For example, let $\Sigma = \Sigma_{(0)} \cup \Sigma_{(1)}$ with $\Sigma_{(0)} = \{a\}$ and $\Sigma_{(1)} = \{f\}$, and consider the wta M with two states q_1, q_2, and the rules $a \xrightarrow{0} q_i$ and $f[q_i] \xrightarrow{1} q_j$ for all $i, j \in [2]$, where q_1 is final. Then $M(t) = |t|$ for every tree $t \in T_\Sigma$ and thus Best Trees simply enumerates trees by size. However, each tree t has $2^{|t|}$ accepting runs, all of weight $|t|$, and thus Filtered Runs needs to generate $2^{|t|} - 1$ best runs to discover t.

In the following, we conduct experiments on synthesized sets of wtas which, rather than triggering this kind of worst-case behaviour, are designed to shed light on particular aspects of the algorithms in less extreme (and thus perhaps practically more relevant) cases. An annotated collection containing all of these wta sets is available on the project web page[2], along with the measured running times for each wta. Due to space restrictions, we limit ourselves for the present to brief descriptions of the data.

The experiments were run on a computer with 8 Intel i7 processors, each at a speed of 3.6 GHz and with 16 GB memory allocated for the JVM. The efficiency results are based on repeated experimentation, and each reported running time is a mean value of five runs. As both Best Trees and Filtered Runs are deterministic algorithms, the existing (but relatively low) variance in the running times is due to variations in the execution environment, e.g., overall system load. All plots show the running times in milliseconds as a function of N or wta size, which is why we exclude the y axis labels from the figures.

[2] http://people.cs.umu.se/aj/besttrees_experiments/.

4.1 Data

Below follows a short presentation of the wta sets used in this paper. Each set consists of a sequence of language-equivalent wtas of increasing sizes. The relatively small wta sizes are due to the limitations of Tiburon: increasing the number of states causes out-of-memory errors when adding more nondeterminism.

Basic Example. A sequence of 20 wtas, starting with the example wta in [1]. Each subsequent automaton was derived from its predecessor by mirroring an existing run for some tree t by the addition of new states and rules resulting in an alternative run on t. Thus, the numbers of states and transition rules increase at the same rate, which allows us to check the running time as a function of the number of states and rules on the one hand, and of N on the other hand. The amount of nondeterminism, however, does not significantly increase as only t gets one more accepting run.

Different Weights. A sequence of 16 wtas over a, b of rank 0 and 0, resp., and states $q_1, \ldots, q_4$, in which all of the transition rules have different weights. All states are final. The transition rules have the weight of the index of their target state. The weight of transition rule $b[q_i, q_j] \to q_k$ is $ijk/100$, where ijk is interpreted as a number in decimal notation. The ℓ-th wta ($\ell \in [16]$) consists of the first 4ℓ of these rules if ordered according to decreasing weight. As a consequence, the degree of nondeterminism is moderate throughout, but is changing as the wta sizes grow, as the later rules result in the best runs. In particular, the degree of nondeterminism of the final wta including all transition rules is low because only the rules $a \to q_1$ and $b[q_1, q_1] \to q_1$ result in cheapest runs.

Modified Different Weights. Similar to Different Weights, but transition rules on b are added in a different order, starting with $b[q_1, q_1] \to q_1$ (see the project web page (see footnote 1) for details). Hence, the best runs in all 16 wtas are those which assign q_1 to all nodes, which means that the degree of nondeterminism is small and grows moderately with growing wta sizes.

Equal Weights. The wta set Equal Weights is also similar to the set Different Weights, but the weights are equal (all 0) apart from the rules on a, which have weight 4. Thus, this example has the highest possible degree of nondeterminism.

4.2 Running Time of Best Trees

Let us first compare the measured running times of Best Trees with the theoretical bound $O(N^2 \cdot (mn \log N + n^3))$ derived in [1].[3]

A somewhat unexpected outcome of our experiments was that our implementation of the computation of the cheapest contexts in line 2 of Algorithm 1, which makes use of an algorithm by Knuth [6], turned out to be slightly less efficient

[3] For the sake of clarity, the expression is slightly simplified. In particular, the maximum rank r of symbols is taken to be constant, as it is typically small in practise.

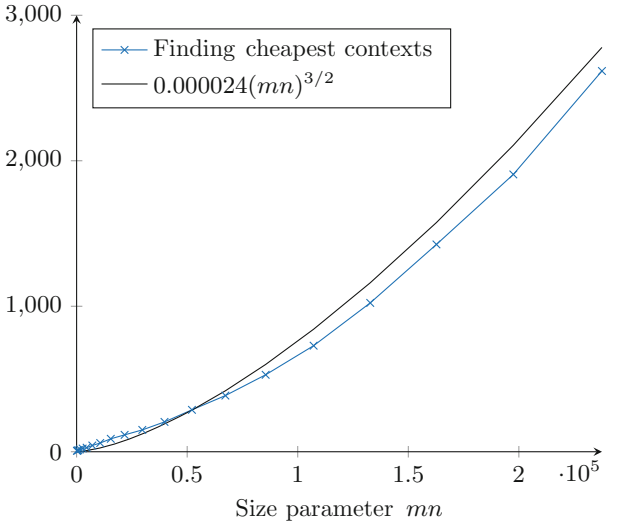
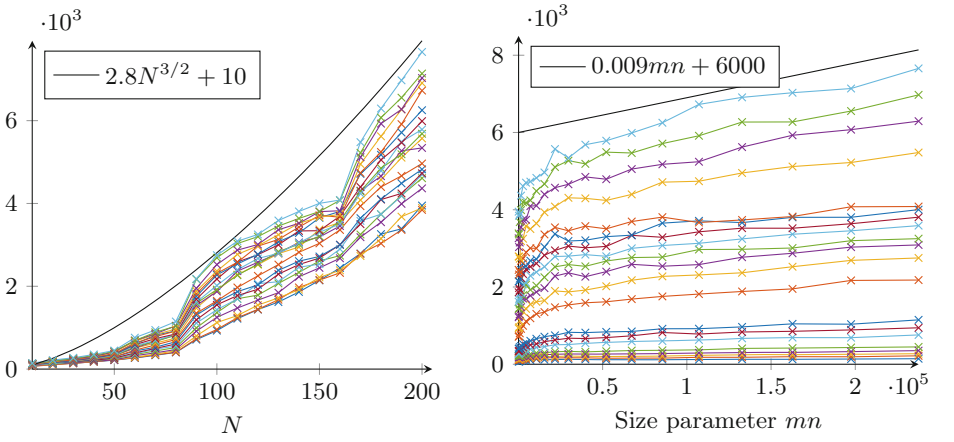


Fig. 1. Running time of finding the cheapest contexts for wtas of increasing size in the set Basic Example.

than the theoretical upper bound $O(mn \log n)$. Plotted against the parameter mn, $O(mn \log n)$ should basically become linear when finding the cheapest contexts for Basic Example. As can be seen in Fig. 1, the practical running time appears to instead be proportional to $(mn)^{3/2}$ on this type of input automata. Further experiments revealed that when m was increased while n was kept constant (suggesting a running time of $\Theta(m)$), we instead acquired figures resembling $\Theta(m^2)$ very closely. Optimising the computation of the cheapest contexts could therefore give a certain gain when N is small, but since the computation time is independent of N, its influence vanishes as N grows. Due to this not being optimised, we from here on disregard the time for finding the cheapest contexts when presenting running times for Best Trees. This, however, only affects the runs on Basic Example significantly.

The numbers of rules and states increase at a similar rate in the wta sequence Basic Example; this behaviour allows us to vary the two parameters N and mn. Based on the theoretical upper bounds, the running times should be in $O(N^2 \log N)$ and $O((mn)^{3/2})$ (the latter because of the term n^3 in the theoretical estimation, which equals $(mn)^{3/2}$ for $m = n$). This is confirmed by our results, which are visualised in Fig. 2a for increasing N, and in Fig. 2b for increasing mn. As may be expected, these running times are slightly better than the theoretical worst-case estimations.

As N varies, the theoretical worst-case running time of $O(N^2 \log N)$ is only reached when the priority queue used by the algorithm is at its maximal length throughout most of the execution. Simply put, this only happens when most trees can reach most states. In practise, the running time is therefor likely to be lower. This is for instance the case for the Basic Example test set.



(a) Running time depending on N; the multiple lines represent the increasingly large wtas of Basic Example.

(b) Running time depending on the size of wtas; the multiple lines represent $N \in \{10, 20, ..., 200\}$.

Fig. 2. Running time of Best Trees on the Basic Example set.

Here, the algorithm exhibits the behaviour shown in Fig. 2a, which is roughly proportional to $N^{3/2}$.

The waviness of the plot is explained by the pattern of the language recognised by the input wtas: it only contains trees with an even number of symbols of rank 0. By there only being a binary symbol in addition to the rank 0 symbols, all of the trees in the language have an odd number of binary symbols. When all of the trees with $2i-1$ binary symbols have been found, the next accepted tree is amongst the trees with $2i+1$ binary symbols. However, all of the possibilities for $2i$ binary symbols have to be processed as well since a rule application adds at most one binary symbol to the resulting tree. Thus, the jumps in the plot represent going from finding the trees with $2i-1$ binary symbols to finding the trees with $2i+1$ binary symbols.

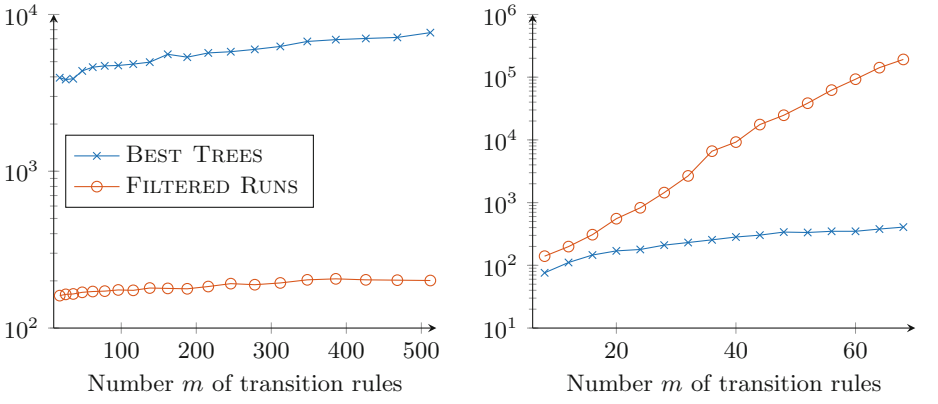
The measured running time as a function of the size of wtas is also lower than the theoretical upper bound $O((mn)^{3/2})$ in this example, namely linear.

4.3 Comparison of Best Trees and Filtered Runs

In these experiments, we used Tiburon v.1.0. We ran Tiburon on the input wta to get d_N runs where d_N is the smallest number of runs that produces N distinct trees. The number d_N was found manually for each combination of input wta and value of N. Then, the list of runs was filtered using a Python script such that only the N best trees remained. The filtering was done by going through the list top-down and collecting only the unseen trees (by comparing each to the previously collected ones) until N trees were gathered. The time spent on

filtering is included in the reported running times but negligible compared to the running time of Tiburon.

Figure 3 shows how the running times of Filtered Runs and Best Trees compare on the two extreme example sets. Even though Best Trees is polynomial on Basic Example (as confirmed above), it is much less efficient than Filtered Runs, which can be explained with the insignificant degree of nondeterminism of the wtas in Basic Example. Turning to Equal Weights instead, the situation changes: as seen in Fig. 3b, Best Trees remains polynomial whereas Filtered Runs appears to exhibit the expected exponential behaviour as it is sensitive to the high (and growing) degree of nondeterminism in Equal Weights.



(a) Comparison of Best Trees and Filtered Runs when run on the Basic Example wta set for $N = 200$.

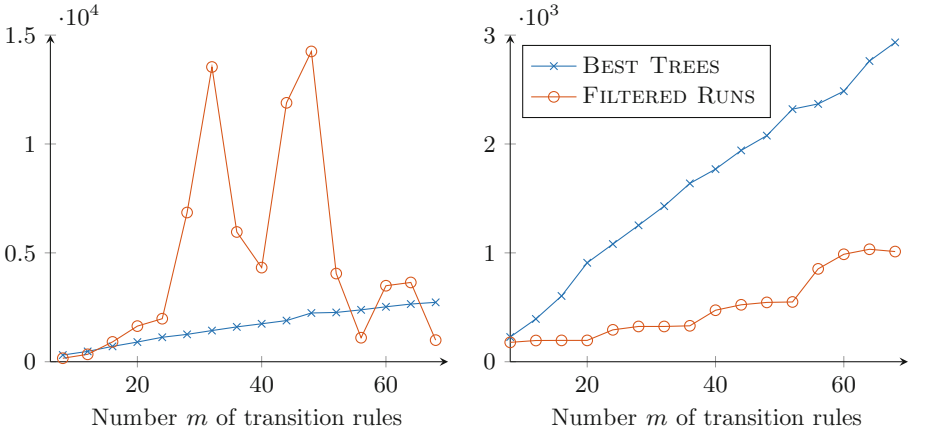
(b) Comparison of Best Trees and Filtered Runs when run on the Equal Weights wta set for $N = 30$.

Fig. 3. Best Trees and Filtered Runs when run on Basic Example and Equal Weights for fixed N.

The results for the less extreme sets Different Weights and Modified Different Weights are shown in Fig. 4. On the former, the running time of Filtered Runs is quite erratic, which can be explained with the changing degree of nondeterminism. In particular, the running time drops towards the end because the degree of nondeterminism does. Because of the low, and slowly but steadily growing degree of nondeterminism of Modified Different Weights, Filtered Runs runs faster on this set and the behaviour is much smoother as wta sizes increase.

4.4 Discussion

An obvious advantage of Best Trees is that its argument N is simply the number of best trees desired. In contrast, using Filtered Runs with the current



(a) Comparison of Best Trees and Filtered Runs when run on the Different Weights wta set.

(b) Comparison of Best Trees and Filtered Runs when run on the Modified Different Weights wta set.

Fig. 4. Best Trees and Filtered Runs when run on Different Weights and Modified Different Weights for $N = 100$.

interface of Tiburon makes it necessary to guess the number d_N of runs needed to produce sufficiently many best runs. To avoid a trial and error procedure, one would have to compute an appropriate – and thus exponentially large – upper bound N' on d_N from N, resulting in a running time that is guaranteed to be exponentially less efficient than Best Trees even in cases where the actual d_N equals N. However, note that this is not an intrinsic weakness of Filtered Runs because the best runs algorithm of [5] allows for a lazy implementation that outputs runs one by one upon request. It should thus not be difficult to modify the Tiburon implementation so as to allow for $N' = \infty$, resulting in an infinite (lazy) list of runs that can be inspected in Filtered Runs to extract any number of best trees. The more serious limitation of Filtered Runs is that its running time is directly related to the number N' of best runs required, which is difficult to predict. The running times reported in our experiments are therefore indicative of how many runs had to be computed for each value of N.

Overall, Best Trees shows a more predictable and smoother behaviour than Filtered Runs, allowing us to predict its time consumption more reliably if the structure of input wtas is not well known (see Fig. 4a).

During the experiments, the memory usage of Tiburon became too high when running it on the nondeterministic wta sets for $N > 10$, forcing us to increase the memory allocated for the JVM from 2–4 GB to 16 GB. In contrast, Best Trees did not encounter such problems and thus seems to use less memory.

5 Conclusion

We have experimentally validated the running time of Best Trees, the N-best algorithm for wtas proposed in [1], and the practical results were in line with the theoretical predictions. We then continued to compare Best Trees with Filtered Runs. Whereas Best Trees can be said to take a direct approach, Filtered Runs is indirect in the sense that it computes a large number of best runs, and then discards those that duplicate previously outputted trees.

As it is easy to create "artificial" examples that make Filtered Runs exponentially less efficient than Best Trees, we used more benign input automata in our experiments. The experimental results confirm that the degree of nondeterminism has a decisive influence on the relative efficiency of both methods. The greater simplicity of Filtered Runs makes it preferable if the degree of nondeterminism is low and the input wtas are large. Conversely, if the degree of nondeterminism is high in comparison to the size of the wta, Best Trees is the more efficient algorithm.

Another, more general, advantage of Best Trees is that it provides guarantees, and that it avoids unpredictable behaviour such as the one seen in Fig. 4a, which may be important in applications where the structure of the input automata is varied or unclear. Using the currently available implementation provided by Tiburon, one can also add that it is inconvenient to be forced to guess the number of runs needed to get N distinct trees.

References

1. Björklund, J., Drewes, F., Zechner, N.: Efficient enumeration of weighted tree languages over the tropical semiring. J. Comput. Syst. Sci. (2017)
2. Fülöp, Z., Vogler, H.: Weighted tree automata and tree transducers. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01492-5_9
3. Gécseg, F., Steinby, M.: Tree Automata. Akadémiai Kiadó (1984). https://arxiv.org/abs/1509.06233
4. Gécseg, F., Steinby, M.: Tree languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 3, Chap. 1, pp. 1–68. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-642-59126-6_1
5. Huang, L., Chiang, D.: Better k-best parsing. In: Proceedings of the Conference on Parsing Technology 2005, pp. 53–64. Association for Computational Linguistics (2005)
6. Knuth, D.E.: A generalization of Dijkstra's algorithm. Inf. Process. Lett. **6**, 1–5 (1977)
7. May, J., Knight, K.: Tiburon: a weighted tree automata toolkit. In: Ibarra, O.H., Yen, H.-C. (eds.) CIAA 2006. LNCS, vol. 4094, pp. 102–113. Springer, Heidelberg (2006). https://doi.org/10.1007/11812128_11
8. Mohri, M., Riley, M.: An efficient algorithm for the n-best-strings problem. In: Proceedings of the Conference on Spoken Language Processing (2002)

III

Faster Computation of N-Best Lists for Weighted Tree Automata[1]

Johanna Björklund[a], Frank Drewes[a], Anna Jonsson[a]

[a]*Department of Computing Science, Umeå University, Sweden*

Abstract

We show that a previously proposed algorithm for the N-best trees problem – not to be confused with the easier N-best runs problem – can be made more efficient by changing how it arranges and explores the search space. Given an integer N and a weighted tree automaton (wta) M over the tropical semiring, the algorithm computes N trees of minimal weight with respect to M. Compared to the original algorithm, the modifications increase the laziness of the evaluation strategy, which makes the new algorithm asymptotically more efficient than its predecessor.

1. Introduction

Tree automata are used in natural language processing to represent the derivation processes of context-free rewrite systems. They are often equipped with weights to assign a likelihood or score to their input rather than just making a binary decision. For example, given a probabilistic context-free grammar (pcfg) G and a string w, we can construct a weighted tree automaton (wta) that processes the parse trees of w according to G, assigning to each its probability with respect to G.[2] That way, the computation of the N most likely parses for w with respect to G becomes an instance of the N best trees problem for wta.

Socher et al. [2] recently improved the Stanford Parser by pairing a pcfg with a recurrent neural network (RNN) that learns vector representations. Intuitively, each nonterminal in the pcfg is associated with a continuous vector space. The vector space induces an unbounded refinement of the category represented by the nonterminal into subcategories, and the RNN computes transitions between such vectors. For efficiency reasons, the device is not applied to the input sentence directly. Instead, the $N = 200$ highest-scoring parse trees with respect to the pcfg are computed, whereupon the RNN is used to rerank these to find the best parse tree. The work has raised interest in hybrid finite-state continuous-state approaches (see, for example, [3]), and underlines the importance of the N-best problem in the context of language processing.

[1]Declarations of interest: none.

[2]This is an instance of *parsing by intersection*, using the well-known technique by Bar-Hillel et al. [1] to intersect a context-free language with a regular one (the latter being $\{w\}$ in this case).

More formally, the input to the N-best trees problem is a nondeterministic wta M over an ordered domain of weights, and the output is a set of N trees that have minimal weight in the weighted tree language recognised by M. The N-best trees problem is similar to that of computing the N best runs of M, and for deterministic automata, the two problems are one and the same. In the nondeterministic case, the N-best trees problem poses the additional difficulty that there may be many different runs on a single input tree, and it is the combined weight of these runs which determines the weight and thus the ranking of the tree.

In the two examples mentioned above, an algorithm solving the N-best runs problem would in fact suffice because pcfgs result in deterministic wtas. However, suppose now that we are interested only in parse trees having a particular structure, say those in which at least one rule is used twice. Using a guess-and-verify technique, we can check this property by a nondeterministic (unweighted) tree automaton M, and combining the original wta with M using a product construction yields a nondeterministic wta in which each parse tree has exponentially many runs. Now, using an N-best runs algorithm would likely return a list of N best parse trees containing duplicates. The problem could be circumvented by determinising M, but that would increase its size exponentially, and is thus not a good strategy.

The computational difficulty of the N-best trees problem depends heavily on the algebraic domain from which weights are taken. Here we assume this domain to be the frequently used tropical semiring. Hence, the weight of a tree t is the minimal weight of any run on t, and the weight of a run is the sum of the weights of the transitions used in the run, which are non-negative real numbers. The tropical semiring is particularly common in speech and text processing [4], since probabilistic devices can be modelled using negative log likelihoods. Moreover, it has the advantage of being extremal: the sum of two elements a and b always equals one of a and b. As a consequence, it is not necessary to consider all runs of an automaton on an input tree to find the weight of the tree, as its weight is equal to the weight of the optimal run.

In previous work [5], Björklund et al. generalised an N-best algorithm by Mohri and Riley [6] from strings to trees. Intuitively, the N-best trees algorithm performs a lazy implicit determinisation and uses a priority queue to output N best trees in the right order. The running time of the algorithm was shown to be in $\mathcal{O}(\max(Nmn \cdot (Nr + r \log r + N \log N), N^2n^3, mr^2))$, where m and n are the numbers of transitions and states of M, respectively, and r is the maximum number of children (the *rank*) of symbols in the input alphabet.

This line of research was continued empirically in [7], a comparative study of our algorithm and the N best runs algorithm by Huang and Chiang [8] implemented in the Tiburon wta toolkit [9], which represents the state of the art. Büchse et al. [10] prove that the algorithm by Huang and Chiang works for cyclic input wta and generalise it by extending it to structured weight domains, but the core idea of the algorithm remains the same. Tiburon can be used to compute N best trees indirectly by first computing a large enough set of best runs and then removing duplicate trees. The conclusion of [7] was that the direct solution is faster if the input wtas exhibit a high degree of nondeterminism, whereas the indirect approach using Tiburon is the better option when the input wtas are large but essentially deterministic.

We now improve the algorithm in [5] by organising the explored search space in a

different way. In the original algorithm, all assembled trees were kept in a single queue. In this paper, we split the queue into as many queues as there are transitions in the input automaton. The queue K_τ of transition τ contains trees which are instantiations of τ, i.e., trees with a run that applies τ at the root. This makes it possible to improve the strategy that was used to prune the queue in [5], avoiding pruning altogether. The intuition is simple: To assemble N distinct output trees, at most N instantiations of any one transition may be needed. We furthermore assemble the instantiations of τ in a lazy fashion, constructing an instantiation explicitly only when it is dequeued from K_τ. We formally prove the correctness of the new algorithm and derive an upper bound on its running time, namely $\mathcal{O}(Nm(\log(m) + r^2 + r\log(Nr)))$.

2. Preliminaries

We write $\mathbb{N}$ for the set of nonnegative integers, $\mathbb{N}_+$ for $\mathbb{N} \setminus \{0\}$, and $\mathbb{R}_+$ for the set of non-negative reals; $\mathbb{N}^\infty$ and $\mathbb{R}_+^\infty$ denote $\mathbb{N} \cup \{\infty\}$ and $\mathbb{R}_+ \cup \{\infty\}$, respectively. For $n \in \mathbb{N}$, $[n] = \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$. Thus, in particular, $[0] = \emptyset$ and $[\infty] = \mathbb{N}$. The cardinality of a (countable) set S is written $|S|$. The n-fold Cartesian product of a set S with itself is denoted by S^n.

As usual, the set of all finite sequences over S is denoted by S^*, and the empty sequence by λ. Given a sequence $\sigma = s_1 \cdots s_n$ of n elements $s_i \in S$, we denote its length n by $|S|$. Given an integer $i \in [n]$, we write σ_i for the i-th element s_i of σ. For simplicity of notation, we occasionally use sequences as if they were sets, e.g., writing $s \in \sigma$ to express that s occurs in σ or $S \setminus \sigma$ to denote the set of all elements of S that do not occur in σ.

For a set A, an A-labelled *tree* is a partial function $t\colon \mathbb{N}_+^* \to A$ whose domain $dom(t)$ is a finite non-empty set that is prefix-closed and closed to the left: whenever $vi \in dom(t)$ for some $v \in \mathbb{N}_+^*$ and $i \in \mathbb{N}_+$, it holds that $v \in dom(t)$ (prefix-closedness) and $vj \in dom(t)$ for all $1 \leq j \leq i$ (closedness to the left). The size of t is $|t| = |dom(t)|$. An element v of $dom(t)$ is called a *node of* t, and $|\{i \in \mathbb{N}_+ \mid vi \in dom(t)\}|$ is the *rank of* v. The *subtree* of $t \in T_\Sigma$ rooted at v is the tree t/v defined by

$$dom(t/v) = \{u \in \mathbb{N}_+^* \mid vu \in dom(t)\}$$

and $t/v(u) = t(vu)$ for every $u \in \mathbb{N}_+^*$. If $t(\lambda) = f$ and $t/i = t_i$ for all $i \in [k]$, where k is the rank of λ in t, then we denote t by $f[t_1, \ldots, t_k]$, which may be identified with f if $k = 0$.

A *ranked alphabet* is a disjoint union of finite sets of symbols, $\Sigma = \bigcup_{k\in\mathbb{N}} \Sigma_{(k)}$. For $f \in \Sigma$, the $k \in \mathbb{N}$ such that $f \in \Sigma_{(k)}$ is the *rank* of f, denoted by $rank(f)$. The set T_Σ of ranked trees over Σ consists of all Σ-labelled trees t in which the rank of every node $v \in dom(t)$ equals the rank of $t(v)$. For a set T of trees we denote by $\Sigma(T)$ the set of trees which have a symbol from Σ at their root, with direct subtrees in T, more precisely, $\{f[t_1, \ldots, t_k] \mid k \in \mathbb{N}, f \in \Sigma_{(k)}, \text{ and } t_1, \ldots, t_k \in T\}$.

In the following, let $\square \notin \Sigma$ be a special symbol of rank 0. The set of *contexts over* Σ is the set C_Σ of trees $c \in T_{\Sigma\cup\{\square\}}$ containing exactly one node $v \in dom(c)$ with $c(v) = \square$. We define the *depth* of c to be $depth(c) = |v|$, i.e., the depth of c is the

distance of $\Box$ from the root of c. The *substitution* of another tree t into c results in the tree $c[\![t]\!]$ given by $dom(c[\![t]\!]) = dom(c) \cup \{vu \mid u \in dom(t)\}$ and

$$c[\![t]\!](w) = \begin{cases} c(w) & \text{if } w \in dom(c) \setminus \{v\} \\ t(u) & \text{if } w = vu \text{ for some } u \in dom(t) \end{cases}$$

for all $w \in dom(c[\![t]\!])$.

Recall that the domain of the *tropical semiring* is $\mathbb{R}_+^\infty$, with $\min$ serving as semiring addition and ordinary plus as semiring multiplication. A *weighted tree language* over the tropical semiring is a mapping $L\colon T_\Sigma \to \mathbb{R}_+^\infty$, where Σ is a ranked alphabet. Such languages can be specified by a *weighted tree automaton* (wta). A wta is a system $M = (Q, \Sigma, R, \omega, q_\mathrm{f})$ consisting of:

- a finite set Q of symbols of rank 0 called *states*;
- a ranked alphabet Σ of *input symbols* disjoint with Q;
- a finite set $R \subseteq \bigcup_{k\in\mathbb{N}} Q^k \times \Sigma_{(k)} \times Q$ of *transition rules*;
- a mapping $\omega\colon R \to \mathbb{R}_+$; and
- a final state $q_\mathrm{f} \in Q$.

From here on, we write $f[q_1, \ldots, q_k] \xrightarrow{w} q$ to denote that $\tau = (q_1, \ldots, q_k, f, q) \in R$ and $\omega(\tau) = w$, and consider R to be the set of these weighted rules, thus dropping the component ω from the definition of M.

A transition rule $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$ will also be viewed as a symbol of rank k, so that R becomes a ranked alphabet. We use $tar(\tau)$ to denote the target state q of τ and let $rank(\tau) = rank(f)$. In addition, we view every state $q \in Q$ as a symbol of rank 0.

We define the set $runs_M \subseteq T_{R\cup Q}$ of *runs ρ of M*, their input trees $input_M(\rho)$, their *intrinsic weights* $wt_M(\rho)$ and their *target state* $tar(\rho)$ inductively, as follows:

1. For every $q \in Q$, we have $q \in runs_M$ with $input_M(q) = q$, $wt_M(q) = 0$, and $tar(q) = q$.
2. For every transition rule $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$ in R and all runs $\rho_1, \ldots, \rho_k \in runs_M$ such that $tar(\rho_i) = q_i$ for all $i \in [k]$, we let $\rho = \tau[\rho_1, \ldots, \rho_k] \in runs_M$ with

$$\begin{aligned} input_M(\rho) &= f[input_M(\rho_1), \ldots, input_M(\rho_k)], \\ wt_M(\rho) &= w + \textstyle\sum_{i\in[k]} wt_M(\rho_i), \text{ and} \\ tar(\rho) &= q. \end{aligned}$$

The *weight* of a run $\rho \in runs_M$ is

$$M(\rho) = \begin{cases} wt_M(\rho) & \text{if } tar(\rho) = q_\mathrm{f} \\ \infty & \text{otherwise.} \end{cases}$$

Now, the weighted tree language $M\colon T_\Sigma \to \mathbb{R}_+^\infty$ *recognised by* M is given by

$$M(t) = \min\{M(\rho) \mid \rho \in \mathit{runs}_M \text{ and } \mathit{input}_M(\rho) = t\}$$

for all $t \in T_\Sigma$ (where, by convention, $\min \emptyset = \infty$). In other words, $M(t)$ is the minimal weight of any run resulting in t. Note that we, by a slight abuse of notation, denote by M both the wta and the weight assignments to runs and trees it computes. Moreover, for $q \in Q$ we define the mapping $M_q\colon C_\Sigma \to \mathbb{R}_+^\infty$ by $M_q(c) = M(c[\![q]\!])$ for every $c \in C_\Sigma$.

Throughout the rest of the paper, we will generally drop the subscript M in runs_M, input_M, and wt_M, because the wta in question will always be clear from the context.

Given a wta M and an integer $N \in \mathbb{N}^\infty$, the *N-best runs problem* consists in computing a sequence of N runs of minimal weight according to M. More precisely, an algorithm solving the problem will output a sequence $\rho_1, \rho_2, \ldots$ of N pairwise distinct runs such that there do not exist $i \in [N]$ and $\rho \in \mathit{runs} \setminus \{\rho_1, \ldots, \rho_i\}$ with $M(\rho) < M(\rho_i)$. We allow $N = \infty$ since it enables us to enumerate the set of all trees over the input alphabet in ascending order until some given criterion is fulfilled.

General assumption. To make sure that the N-best runs problem always possesses a solution, and to simplify the presentation of our algorithms, we assume from now on that all considered wtas M have infinitely many runs ρ such that $\mathit{tar}(\rho) = q_\mathrm{f}$. In particular, T_Σ is assumed to be infinite. Apart from simplifying some technical details, this assumption does not affect any of the reasonings in the paper.

Similarly to the N-best runs problem, the *N-best trees problem* for the wta M consists in computing a sequence of pairwise distinct trees $t_1, t_2, \ldots$ in T_Σ of minimal weight. In other words, we seek a sequence of trees such that there do not exist $i \in [N]$ and $t \in T_\Sigma \setminus \{t_1, \ldots, t_i\}$ with $M(t) < M(t_i)$. Note that the N-best trees problem always has a solution because we assume that T_Σ is infinite.

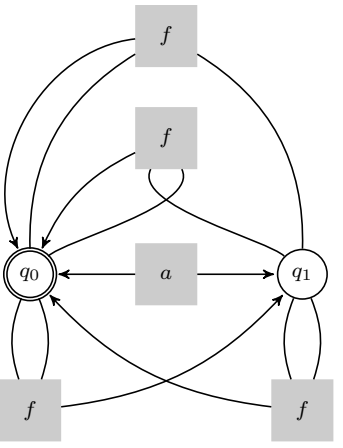
Example 2.1. *Figure 1 shows an example wta, and Table 1 contains a side-by-side comparison of the input trees of the* 10 *best runs and the* 10 *best trees of the example wta. Since the weight of every transition rules is* 1*, the weight of every run on an input tree t is its size $|t|$. Looking at the rules, we get the following recursive equations for the number $\#_i(t)$ of runs ρ on a tree t ending in state $\mathit{tar}(\rho) = q_i$:*

$$\begin{aligned}
\#_0(a) &= 1\\
\#_1(a) &= 1\\
\#_0(f[t_0, t_1]) &= \#_0(t_0)\#_1(t_1) + \#_1(t_0)\#_0(t_1) + \#_1(t_0)\#_1(t_1)\\
\#_1(f[t_0, t_1]) &= \#_0(t_0)\#_0(t_1)
\end{aligned}$$

Hence, every tree t will occur $\#_0(t)$ times in an N-best list based on best runs (provided that N is large enough).

3. Algorithm

In this section, we explain how the algorithm in [5] can be made lazier, and hence more efficient, by exploring the search space with respect to transitions rather than states. From here on, let $M = (Q, \Sigma, R, q_\mathrm{f})$ be a wta with m transition rules, n states, and a maximum rank of r among the symbols in Σ.



$$a \xrightarrow{1} q_i \quad \text{for } i \in \{0,1\}$$
$$f[q_i, q_i] \xrightarrow{1} q_{1-i} \text{ for } i \in \{0,1\}$$
$$f[q_i, q_j] \xrightarrow{1} q_0 \quad \text{for } \{i,j\} = \{0,1\}$$

Figure 1: An example wta. The input alphabet is $\Sigma_{(0)} \cup \Sigma_{(2)}$, where $\Sigma_{(0)} = \{a\}$ and $\Sigma_{(2)} = \{f\}$. Circles represent states (double circles indicate the final state, i.e., $q_\mathrm{f} = q_0$), and squares represent transitions (all of weight 1). The consumed input symbols are shown inside the squares. Undirected edges connect states in left-hand sides to consumed input symbols in counter-clockwise order starting at noon. Directed arcs point from the consumed symbol to the right-hand side state of the transition in question.

Best runs		**Best trees**	
Input tree	Weight	Tree	Weight
a	1	a	1
$f[a,a]$	3	$f[a,a]$	3
$f[a,a]$	3	$f[f[a,a],a]$	5
$f[a,a]$	3	$f[a,f[a,a]]$	5
$f[a,f[a,a]]$	5	$f[f[a,a],f[a,a]]$	7
$f[f[a,a],a]$	5	$f[f[f[a,a],a],a]$	7
$f[a,f[a,a]]$	5	$f[f[a,f[a,a]],a]$	7
$f[f[a,a],a]$	5	$f[a,f[f[a,a],a]]$	7
$f[a,f[a,a]]$	5	$f[a,f[a,f[a,a]]]$	7
$f[f[a,a],a]$	5	$f[f[a,f[a,a]],f[a,a]]$	9

Table 1: The input trees of 10 best runs (left) in comparison with 10 best trees (right) for the wta in Figure 1.

3.1. Best Trees

We first recall the approach of [5] in a summary form. The algorithm maintains two data structures: an initially empty set T that collects all processed trees, and a priority queue K of trees in $\Sigma(T)$ that will be examined next. The priority of a tree t in K is determined by the minimal value in the set of all $M(c[\![t]\!])$, where c ranges over all possible contexts. Let M^q denote the wta obtained from M by making q its final state. Then, for every context c and all trees t,

$$M(c[\![t]\!]) = \min_{q \in Q} \left(M_q(c) + M^q(t) \right) \quad . \tag{1}$$

Since $M_q(c)$ is independent of t, it is possible to compute in advance a *best context* $\mathbb{c}_q$ that minimises it [5]. The way in which this can be done will be discussed in Section 3.2.

Definition 3.1. *A* best context *of a state $q \in Q$ is a context $\mathbb{c}_q$ with*

$$M_q(\mathbb{c}_q) = \min_{c \in C_\Sigma} M_q(c).$$

The value $M_q(\mathbb{c}_q)$ is denoted by $\mathbb{w}_q$.

The algorithm uses the best contexts to compute an *optimal state* $opt(t)$ for each tree t that it encounters: $opt(t) = \operatorname{argmin}_{q \in Q} \mathbb{c}_q[\![t]\!]$. Throughout the paper, we shall make use of the following weight functions derived from M, where $t \in T_\Sigma$:

$$\begin{aligned} \Delta_q(t) &= \mathbb{w}_q + M^q(t) \\ \Delta(t) &= M(\mathbb{c}_{opt(t)}[\![t]\!]) \ . \end{aligned}$$

Note that $\Delta(t) = \min_{c \in C_\Sigma} M(c[\![t]\!]) = \min_{q \in Q} \Delta_q(t)$.

The priority queue K is initialised with the trees in Σ_0. Its priority order $<_K$ is defined as follows, for all trees t and t' in K:

$$\begin{aligned} t <_K t' \quad \Longleftrightarrow \quad & \Delta(t) < \Delta(t') \text{ or} \\ & \Delta(t) = \Delta(t') \text{ and } t <_{lex} t' \ . \end{aligned}$$

Here, $<_{lex}$ is any lexical order that orders trees first by size and then by viewing them as strings to be compared alphabetically from left to right.

We can now reproduce the pseudocode of the base algorithm from [5] in Algorithm 1. Given a wta M and $N \in \mathbb{N}^\infty$, it solves the N-best trees problem. After outputting $i \in [N]$ trees, the set of trees enqueued in line 22 is pruned so that for every $q \in Q$, at most $N - i$ trees are kept for which q is an optimal state. The function $expand(T, t)$, which computes the trees to be enqueued in each step, returns the set of all trees in $\Sigma(T)$ such that the "new" tree t occurs at least once among the direct subtrees of the root.

The correctness of this approach is formally proved in [5].

3.2. Computation of Best Contexts

We now recall the computation of best contexts $\mathbb{c}_q$, $q \in Q$, to the extent needed to understand the improved algorithm. This computation consists of two phases: first, a 1-best tree $\mathbb{t}_q$ is computed for each state $q \in Q$. The desired property of $\mathbb{t}_q$ is that it is a 1-best tree of M^q, that is, it is a tree $t \in T_\Sigma$ that minimises $M^q(t)$. After that, the second phase computes the actual best context $\mathbb{c}_q$ for every $q \in Q$, i.e., a context $c \in C_\Sigma$ with $M_q(c) = \mathbb{w}_q$.

The first phase can be accomplished using a dynamic programming algorithm by Knuth [11] that computes 1-best runs. (Note that, if ρ is a 1-best run, then $input(\rho)$ is a 1-best tree.) The algorithm maintains a min-priority queue of all transition rules and collects, in $|R|$ iterations, the desired best runs $\mathbb{p}_q$. Initially, $\mathbb{p}_q$ is undefined for every $q \in Q$. The value determining the priority of a transition rule $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$

Algorithm 1 Compute $N \in \mathbb{N}^\infty$ trees of minimal weight according to a wta M

procedure BEST TREES V.1(M, N)
  compute best contexts as
  $T \leftarrow \emptyset$
  $K \leftarrow \emptyset$

  for $f \in \Sigma_0$ **do**
    $\mathrm{enq}(K, f)$
  end for

  $c \leftarrow 0$
  while $c < N$ **do**
    $t \leftarrow \mathrm{deq}(K)$

    $T \leftarrow T \cup \{t\}$

    if $M(t) = \Delta(t)$ **then**
      $\mathrm{output}(t)$
      $c \leftarrow c + 1$
    end if
    $\mathrm{enq}(K, \mathit{expand}(T, t))$

  end while
end procedure

Algorithm 2 Compute $N \in \mathbb{N}^\infty$ trees of minimal weight according to a wta M

procedure BEST TREES V.2(M, N)
  compute best contexts
  output $\mathbb{t}_{q_\mathrm{f}}$
  $T_q \leftarrow \mathrm{mkList}(\mathbb{t}_q)$ for every $q \in Q$
  for $(\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q) \in R$ **do**
    $U \leftarrow \begin{cases} \mathit{inc}(1^{\mathit{rank}(\tau)}) & \text{if } \tau = \mathbb{p}_q(\lambda) \\ \{1^{\mathit{rank}(\tau)}\} & \text{otherwise} \end{cases}$
    $K_\tau \leftarrow \mathrm{mkQeueue}(U)$
  end for
  $K' \leftarrow \mathrm{mkQueue}(\{K_\tau \mid \tau \in R\})$
  $c \leftarrow 0$
  while $c < N$ **do**
    $K_\tau \leftarrow \mathrm{deq}(K')$
    $u \leftarrow \mathrm{deq}(K_\tau)$
    if $\left|T_{\mathit{tar}(\tau)}\right| < N \wedge \tau[u] \notin T_{\mathit{tar}(\tau)}$ **then**
      append $\tau[u]$ to $T_{\mathit{tar}(\tau)}$
    end if
    if $\mathit{tar}(\tau) = q_\mathrm{f} \wedge \tau[u]$ is new **then**
      $\mathrm{output}(\tau[u])$
      $c \leftarrow c + 1$
    end if
    $\mathrm{enq}(K_\tau, \mathit{inc}(u))$ [a]
    $\mathrm{enq}(K', K_\tau)$
  end while
end procedure

[a] We assume that $\mathrm{enq}(K_\tau, U)$ enqueues all $u \in U$ in K_τ except those already in it, thus skipping duplicates.

is ∞ if any $\mathbb{p}_{q_i}$ ($i \in [k]$) is still undefined. Otherwise, it is $w + \sum_{i \in [k]} \mathit{wt}(\mathbb{p}_{q_i})$, i.e., the weight of the run $\tau[\mathbb{p}_{q_1}, \ldots, \mathbb{p}_{q_k}]$. The algorithm repeatedly dequeues the highest priority element $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$ from the queue. If $\mathbb{p}_q$ is still undefined, it sets $\mathbb{p}_q = \tau[\mathbb{p}_{q_1}, \ldots, \mathbb{p}_{q_k}]$ and $\mathbb{t}_q = f[\mathbb{t}_{q_1}, \ldots, \mathbb{t}_{q_k}]$. It then updates the priorities of transition rules having q among the states in their right-hand sides and repeats. We note that the trees $\mathbb{t}_q$ are discovered by the algorithm in the order of ascending weight. This observation will soon become important for the initialisation phase of Algorithm 2.

As a side remark, we note the set of 1-best runs and 1-best trees determined by this algorithm are *subtree closed*, meaning that every subtree of $\mathbb{p}_q$ and $\mathbb{t}_q$ is itself one of the trees $\mathbb{p}_{q'}$ and $\mathbb{t}_{q'}$, respectively. It follows that the entire set of these trees can be stored as a maximally shared structure in space $O(|M|)$, where

$$|M| = \sum_{(f[q_1,\ldots,q_k] \xrightarrow{w} q) \in R} (k+1) \ .$$

The second phase applies Dijkstra's shortest paths algorithm [12] to M, where M

is viewed as a weighted edge-labelled graph. More precisely, consider the graph with node set Q such that, for every transition rule $(\tau\colon f[q_1,\ldots,q_k] \xrightarrow{w} q) \in R$ and every $q' \in \{q_1,\ldots,q_k\}$, there is an edge $e = (q,\tau,q')$ from q to q' with label τ. The weight of such an edge is given by

$$wt(e) = w - wt(\mathbb{p}_{q'}) + \sum_{i\in[k]} wt(\mathbb{p}_{q_i}) \ .$$

The intuition behind this weight assignment is that the edge represents the possibility that $\mathbb{c}_{q'} = \mathbb{c}_q[\![f[\mathbb{t}_{q_1},\ldots,\mathbb{t}_{q_{i-1}},\Box,\mathbb{t}_{q_{i+1}},\ldots,\mathbb{t}_{q_k}]]\!]$ (where $q' = q_i$), in which case

$$\begin{aligned}\mathbb{w}_{q'} &= \mathbb{w}_q + w + \sum_{j\in[k]\setminus\{i\}} wt(\mathbb{p}_{q_j}) \\ &= \mathbb{w}_q + w - wt(\mathbb{p}_{q'}) + \sum_{i\in[k]} wt(\mathbb{p}_{q_i}) \ .\end{aligned}$$

Thus, $wt(e)$ is is the weight that needs to be added to $\mathbb{w}_q$ to obtain $\mathbb{w}_{q'}$ (under the assumption that, indeed, $\mathbb{c}_{q'} = \mathbb{c}_q[\![f[\mathbb{t}_{q_1},\ldots,\mathbb{t}_{q_{i-1}},\Box,\mathbb{t}_{q_{i+1}},\ldots,\mathbb{t}_{q_k}]]\!]$).

Now, having computed the paths of minimal weight in this graph using Dijkstra's algorithm, consider the edge sequence π of least weight from q_f to a state q'. Then $\mathbb{c}_{q'} = (\pi)$, where (π) is is defined recursively as follows:

1. If $\pi = \lambda$ then $q' = q_\mathrm{f}$ and we set $(\pi) = \Box$.
2. If $\pi = \pi' e$ where $e = (q,\tau,q')$ for some transition rule $\tau\colon f[q_1,\ldots,q_k] \xrightarrow{w} q$, we choose some $i \in [k]$ with $q_i = q'$, and set

$$(\pi) = (\pi')[\![f[\mathbb{t}_{q_1},\ldots,\mathbb{t}_{q_{i-1}},\Box,\mathbb{t}_{q_{i+1}},\ldots,\mathbb{t}_{q_k}]]\!] \ .$$

We note here that this way of computing best contexts results in contexts of a very peculiar kind: every such context $\mathbb{c}_q$ consists of a "spine" leading to the node v such that $\mathbb{c}_q(v) = \Box$, and all subtrees that branch out from this spine are of the form $\mathbb{t}_{q'}$ for the required states q'.

3.3. Transition-Based Best Trees

The improved algorithm for computing N best trees hinges on the observation that to generate N best trees, at most N distinct instantiations of each transition rule are needed. Here, an instantiation of a rule $\tau = f[q_1,\ldots,q_k] \xrightarrow{w} q$ is a tree $f[t_1,\ldots,t_k]$. Furthermore, the algorithm creates these instantiations in a lazy fashion.

To this end, a separate priority queue K_τ is kept for each transition rule $\tau \in R$. These priority queues do not store the trees themselves directly. Instead, for a rule τ as above, sequences of k-tuples of indices are stored, each index i_j at position j pointing to the i_j-th element of a sequence of previously dequeued trees leading to state q_j. Hence, each such tuple can be understood as an abstract instruction of how to instantiate τ by previously dequeued trees. As outlined in Section 3.5, an efficient implementation of the algorithm can make this assembly "just in time", so as to avoid unnecessary work.

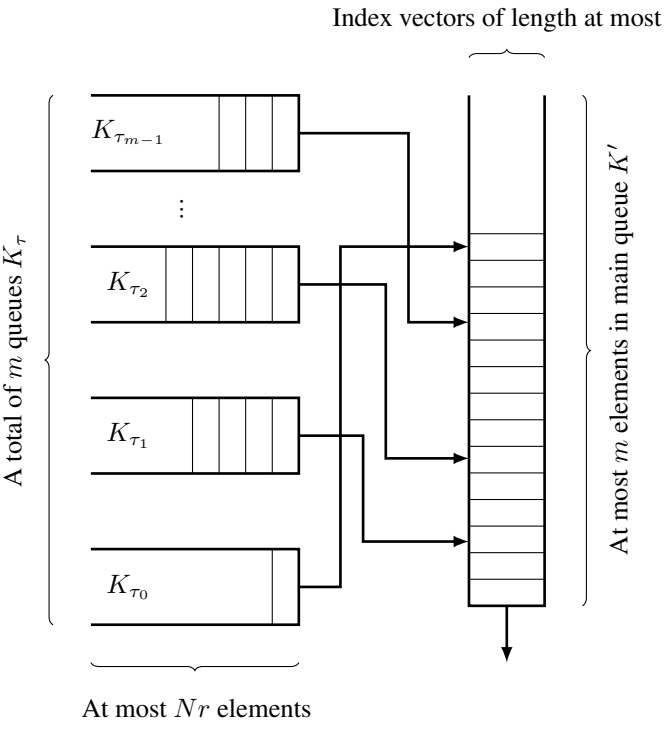


Figure 2: The algorithm keeps a dedicated priority queue K_τ for each transition rule τ, and the collection of these queues is itself arranged in a main priority queue K' (drawn on the right).

In each iteration, the algorithm chooses the highest-priority element across all of the queues K_τ, where the priority order (to be described later) is similar to $<_K$, but improved by replacing the use of the lexical order by a more goal-oriented component. To efficiently pick the highest-priority element across all queues, we organise the queues in a meta-queue K', where the priority of every K_τ in K' is given by the priority of its highest-priority element. This organisation is schematically illustrated in Figure 2.

The algorithm itself is outlined in Algorithm 2. In addition to the aforementioned queues it maintains, for each state $q \in Q$, a sequence T_q of at most N trees in T_Σ. These are the previously dequeued best instantiations of rules τ with $tar(\tau) = q$. In other words, T_q is a prefix of a solution of the N-best problem for the wta M^q.

The priority queue K of Algorithm 1 is, as previously mentioned, divided into a set of queues $\{K_\tau \mid \tau \in R\}$ in Algorithm 2. More precisely, for every $\tau \in R$, the queue K_τ contains index tuples in $\mathbb{N}^{rank(\tau)}$. These tuples represent trees as follows: Given a transition rule $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$ in R and a tuple $u = (i_1, \ldots, i_k) \in \mathbb{N}^k$, we denote by $\tau[u]$ the tree $f[(T_{q_1})_{i_1}, \ldots, (T_{q_k})_{i_k}]$ if $i_j \leq \left|T_{q_j}\right|$ for every $j \in [k]$, and let $\tau[u]$ be undefined otherwise.[3]

[3]Recall that $(T_{q_j})_{i_j}$ denotes the i_j-th element of the sequence T_{q_j}.

The function inc returns, for every tuple $u = (i_1, \ldots, i_k) \in \mathbb{N}^k$ ($k \in \mathbb{N}$), the set of all its successors obtained by increasing exactly one of i_j, $j \in [k]$, by 1.

Definition 3.2 (The inc function)**.** *Let $k \in \mathbb{N}$. The function inc maps every tuple $u = (i_1, \ldots, i_k) \in \mathbb{N}^k$ to the set of its immediate successors, as follows:*

$$inc(u) = \{(i_1, \ldots, i_{j-1}, i_j + 1, i_{j+1}, \ldots i_k) \mid j \in [k]\} \ .$$

Each queue K_τ in Algorithm 2 is a min-priority queue in which the least priority is assigned to elements u such that $\tau[u]$ is undefined. This reflects that we do not yet know the weight of the instantiation of τ with respect to u, and that every instantiation for which we do know the weight of the resulting tree is preferable.

To define the priority used in K_τ, we follow the previous approach, using a variant of $<_K$, but in addition to the necessary adaptations, we shall replace the lexical component by a better choice resulting in a more goal-oriented search. The priority of every element u of K_τ is mainly determined by a variant of Δ, denoted by $\Delta_\tau(u)$. Recall that, for $q \in Q$, $\Delta_q(t)$ is the least weight of all runs on trees of the form $c[\![t]\!]$, where only runs are considered whose target state at the root of the subtree t is q. Thus, $\Delta_q(t) \geq \Delta(t)$, where equality holds for $q = opt(t)$. Now, Δ_τ specialises this further by assuming that the specific transition rule applied at the root of the subtree t is τ. Moreover, since we only apply this weight function to trees of the form $\tau[u]$, we let its argument be u rather than $\tau[u]$. Formally, consider a transition rule $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$. If $\tau[u]$ is undefined, we simply put $\Delta_\tau(u) = \infty$. If $\tau[u]$ is defined, we define

$$\Delta_\tau(u) = \mathbb{w}_q + w + \sum_{i=1}^{k} M^{q_i}((T_{q_i})_{u_i}) \ .$$

This expression may look a bit complicated, but the intuition behind it is actually straightforward. As we shall see, q_i is an optimal state for $(T_{q_i})_{u_i}$ for $i \in [k]$. Thus, if ρ_i is the lowest-weight run on $(T_{q_i})_{u_i}$ with $input(\rho_i) = (T_{q_i})_{u_i}$ and $tar(\rho_i) = q_i$, and we set $\rho = \tau[\rho_1, \ldots, \rho_k]$, then the expression $w + \sum_{i=1}^{k} M^{q_i}((T_{q_i})_{u_i})$ is simply the definition of $wt(\rho)$, which is equal to $M^q(\tau[u])$. Thus, adding $\mathbb{w}_q$, $\Delta_\tau(u)$ turns out to be the minimum of all $M(\rho)$, where ρ ranges over all runs with $\rho(v) = \tau$ and $input(\rho/v) = \tau[u]$ for some $v \in dom(\rho)$. Thus, $\Delta_\tau(u) \geq \Delta_q(\tau[u]) \geq \Delta(\tau[u])$.

Finally, to complete the definition of the priority used in K_τ ($\tau \in R$) and in K', let $\Lambda_\tau(u) \in \mathbb{N}^\infty \times \mathbb{N}$ be given by

$$\Lambda_\tau(u) = (\Delta_\tau(u), \delta_{tar(\tau)}) \tag{2}$$

where $\delta_q = depth(\mathbb{c}_q))$ for every $q \in Q$. We order $(i, j), (i', j') \in \mathbb{N}^\infty \times \mathbb{N}$ as usual, i.e., $(i, j) < (i', j')$ if $i < i'$ or $i = i'$ and $j < j'$.

Now, let $q \in Q$ and $(\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q) \in R$. Having computed best contexts (and, in the process, also best trees) for all states, T_q is initialised to contain only the best tree $\mathbb{t}_q$ for q. The queue K_τ is initialised to contain only 1^k if $\tau \neq \mathbb{p}_q(\lambda)$, since the latter means that τ was not used to build $\mathbb{t}_q$, which means that the first instantiation of τ is still waiting to be added to T_q at a suitable position. Otherwise, K_τ is initialised to

contain all successors of 1^k because, by the way in which $\mathbb{t}_q$ was constructed and the definition of $\tau[u]$, we have $(T_q)_1 = \mathbb{t}_q = \tau[1^k]$ and are now looking for the next best instantiation of τ.

After initialising T_q and K_τ, the algorithm enters the main loop. This loop is executed until N trees have been outputted. The first step in the loop is to extract a minimal u from the set of all queues K_τ, $\tau \in R$ (by first dequeuing K_τ from K' and then u from K_τ). Thanks to our assumption that M possesses infinitely many runs ending in q_f, it can be shown that $\Delta_\tau(u) \in \mathbb{N}$, i.e., the tree $\tau[u]$ is defined. The tree is appended at the end of the list of the best (smallest-weighted) trees that reach the target state $tar(\tau)$ of τ. If $tar(\tau) = q_\mathrm{f}$ and $\tau[u]$ is "new", i.e., has not been outputted before, then $\tau[u]$ is outputted now, and the counter c tracking the number of output trees is increased. Note that, in contrast to Algorithm 1, we have to check whether $\tau[u]$ was outputted before, because some $\tau'[u']$ outputted earlier may actually have been equal to $\tau[u]$. However, this can only be the case if $\tau \neq \tau'$, and can thus only happen m times for every tree.

The above-mentioned fact that there always exist τ and u such that $\tau[u]$ is defined and thus $\Delta_\tau(u) \neq (\infty, 0)$, is proved in the following section.

3.4. Correctness

Let us now show that Algorithm 2 is correct. To simplify the reasoning, we shall first consider a variant of the algorithm, referred to as Algorithm 2′, obtained by removing the inequality $\left|T_{tar(\tau)}\right| < N$ from the condition on line 14. In the following lemma, we say that $T_q = t_1 \cdots t_m$ is *ordered according to weight* if $M^q(t_1) \leq \cdots \leq M^q(t_m)$.

Lemma 3.3. *Consider a run of Algorithm 2′. During every execution of the main loop the following statements hold:*

(1) When the loop is entered, each T_q ($q \in Q$) is ordered according to weight.

(2) When line 13 has been executed with $tar(\tau) = q$, there do not exist any $q' \in Q$ and $t \in T_\Sigma \setminus T_{q'}$ such that $\Delta_{q'}(t) < \Delta_\tau(u)$ (where τ and u denote the values of the corresponding variables in Algorithm 2′ at that point).

Proof. We use (1) as a loop invariant. Since it holds before the first execution of the loop (as each T_q is a singleton), we need to show that, under the condition that (1) holds when the loop is entered, statement (2) holds as well and, when line 21 has been executed, (1) still holds.

We first show (2). Assume for a contradiction that t did actually exist and let $s = \mathbb{c}_{q'}[\![t]\!]$, where $\mathbb{c}_{q'}(v_0) = \Box$, i.e., v_0 is the node such that $s/v_0 = t$. By the definition of $\Delta_{q'}$, there is a run ρ with $input(\rho) = s$, $tar(\rho/v_0) = q'$, and $wt(\rho) = \Delta_{q'}(t)$. For every $v \in dom(s)$, let $q_v = tar(\rho/v)$ be the state the run is in after having processed s/v. There is at least one node $v \in dom(s)$ such that $s/v \notin T_{q_v}$ (namely $v = v_0$). Now, choose $v \in dom(s)$ with $s/v \notin T_{q_v}$ in such a way that $|s/v|$ is minimal, and suppose that $s/v = g[s_1, \ldots, s_\ell]$ and $\rho(v) = (\tau' \colon g[p_1, \ldots, p_\ell] \xrightarrow{w} q_v)$. (Thus, $p_i = q_{vi}$ for all $i \in [\ell]$.) By the minimality of $|s/v|$, $s_1 \in T_{p_1}, \ldots, s_\ell \in T_{p_\ell}$,

and thus there is $u' = (j_1, \ldots, j_\ell) \in \mathbb{N}^\ell$ such that $(T_{p_i})_{j_i} = s_i$ for all $i \in [\ell]$. With $p = opt(s/v)$ it follows that

$$\begin{aligned}\Delta_{\tau'}(u') &= \mathbb{w}_p + w + \sum_{i=1}^{\ell} M^{p_i}((T_{p_i})_{j_i}) \\ &= \mathbb{w}_p + wt(\rho/v) \\ &\leq wt(\rho) \\ &= \Delta_{q'}(t) \\ &< \Delta_\tau(u) \ .\end{aligned}$$

Since $s/v \notin T_{q_v}$, the tuple u' has not yet been dequeued from $K_{\tau'}$. Thus, while u' itself may not yet be in $K_{\tau'}$, $K_{\tau'}$ must contain some $u'' = (j''_1, \ldots, j''_\ell)$ with $j''_i \leq j_i$ for all $i \in [\ell]$. This is because when an element is dequeued on line 13 then all of its direct successors will be enqueued on line 21. In particular, $K_{\tau'}$ cannot be empty at the start of an iteration, as long as there is some $u' \in \mathbb{N}^\ell$ that has not yet been dequeued from $K_{\tau'}$ (which will always be the case if $\ell > 0$).

Note that $j''_i \leq j_i$ for all $i \in [\ell]$ implies that $\Delta_{q_v}(u'') \leq \Delta_{q_v}(u')$ since $T_{p_1}, \ldots, T_{p_\ell}$ are ordered according to weight. Hence, the inequality $\Delta_{\tau'}(u'') \leq \Delta_{\tau'}(u') < \Delta_\tau(u)$ contradicts the assumption that u was dequeued on line 13.

We have thus proved (2). Since $\tau[u]$ is appended to T_q on line 15, it remains to be shown that $M^q(\tau[w]) \leq M^q(\tau[u])$ for all trees $\tau[w]$ that have been appended to T_q during earlier iterations. Choosing q as q', $\tau[u]$ as t, and w as u in (2) we get

$$\begin{aligned}M^q(\tau[u]) &= \Delta_q(\tau[u]) - \mathbb{w}_q \\ &\geq \Delta_\tau(w) - \mathbb{w}_q \\ &\geq \Delta_q(\tau[w]) - \mathbb{w}_q \\ &= M^q(\tau[w]),\end{aligned}$$

where the first inequality holds by (2) and the second holds because $tar(\tau) = q$. Thus, T_q is still ordered according to weight when $\tau[u]$ has been appended to it. □

Lemma 3.4. *Algorithm 2′ computes a solution to the N-best trees problem.*

Proof. We first observe that, due to the output condition on line 17 of Algorithm 2′, the sequence of trees outputted by the algorithm does not contain repetitions.

Next, whenever a tree $s = \tau[u]$ is outputted on line 18, we show that $M(t) \geq M(s)$ for all trees $t \in T_\Sigma$ that have not yet been outputted. Let $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$. Then $q = q_\mathrm{f}$ and thus $\mathbb{c}_q = \square$, $\mathbb{w}_q = 0$, and $M(s) = \Delta_\tau(u)$. Now, consider any $t \in T_\Sigma$ that has not yet been outputted. Then we have $t \in T_\Sigma \setminus T_{q_\mathrm{f}}$. Consequently, $M(t) = \Delta_{q_\mathrm{f}}(t) \geq \Delta_\tau(u)$ by Lemma 3.3, as required.

To complete the proof, we have to show that there cannot be an infinite number of iterations without any tree being outputted. We show the following, stronger statement:

> *Claim 1.* Let $\delta = \max_{q \in Q} \delta_q$, where δ_q is defined as in Equation 2 for all $q \in Q$. At any point in time during the execution of Algorithm 2′, it takes at most δ iterations until the selected transition rule τ satisfies $tar(\tau) = q_\mathrm{f}$.

To see this, let K_τ be dequeued on line 12 and let $q = tar(\tau)$. If $q = q_\mathrm{f}$, the statement holds. Otherwise, the context $\mathbb{c}_q$ has the form

$$\mathbb{c}_{q'}[\![g[\mathbb{t}_{p_1}, \ldots, \mathbb{t}_{p_{i-1}}, \Box, \mathbb{t}_{p_{i+1}}, \ldots, \mathbb{t}_{p_\ell}]]\!]$$

for a transition rule $\tau' \colon g[p_1, \ldots, p_\ell] \xrightarrow{w'} q'$ with $p_i = q$. The tree $t = \tau[u]$ is appended to T_q, say at position n. It follows that $t' = \tau'[u']$ with

$$u' = (\underbrace{1, \ldots, 1}_{i-1}, n, \underbrace{1, \ldots, 1}_{\ell-i-1})$$

becomes defined, where $t' = g[\mathbb{t}_{p_1}, \ldots, \mathbb{t}_{p_{i-1}}, t, \mathbb{t}_{p_{i+1}}, \ldots, \mathbb{t}_{p_\ell}]$. We also know that

1. there is no u'' in any of the queues $K_{\tau''}$ with $\Delta_{\tau''}(u'') < \Delta_\tau(u)$ (Lemma 3.3(2)),
2. $\Delta_{\tau'}(u') = M(\mathbb{c}_q[\![t]\!]) = M(\mathbb{c}_{q'}[\![t']\!]) = \Delta_\tau(u)$, and
3. $\delta_{q'} = \delta_q - 1$.

It follows that the queue $K_{\tau''}$ dequeued on line 12 at the start of the next iteration satisfies $\delta_{tar(\tau'')} = \delta_q - 1$, thus bounding the number of iterations until this quantity reaches 0 from above by δ.

Now, to finish the proof, note that $\tau[u] \neq \tau[u']$ whenever $u \neq u'$ because the sequences T_q, $q \in Q$, do not contain repetitions. Since, furthermore, no tuple u is enqueued twice in K_τ, line 18 will be reached after at most δ iterations. □

We finally show that Algorithm 2 is correct as well.

Theorem 3.5. *Algorithm 2 computes a solution to the N-best trees problem.*

Proof. Consider an execution of Algorithm 2′ and assume that we assign every tree a colour, red or black, where black is the default colour. (The colour attribute of a subtree may differ from that of the tree itself.) Suppose that, in an iteration of the main loop, we have $\tau \colon f[q_1, \ldots, q_k] \xrightarrow{w} q$ and $u = (i_1, \ldots, i_k)$, and thus $\tau[u] = f(t_1, \ldots, t_k)$ where $t_j = (T_{q_j})_{i_j}$ for all $j \in [k]$. If $|T_q| \geq N$ on line 14 and we append $\tau[u]$ to it on line 15, we colour $\tau[u]$ red while its subtrees $t_1, \ldots, t_k$ keep their colours as given by their positions in $T_{q_1}, \ldots, T_{q_k}$.

Now, assume that some T_q contains a tree $t = f[t_1, \ldots, t_k]$ such that t_j is red for some $j \in [k]$. We show that this implies that t is red. We know that t was appended to T_q as a tree of the form $\tau[u]$ for some $u = (i_1, \ldots, i_k)$ with $i_j > N$ (because of the assumption that t_j is red). We know also that earlier iterations have dequeued all tuples from K_τ of the form $u^i = (i_1, \ldots, i_{j-1}, i, i_{j+1}, \ldots, i_k)$ for $i = 1, \ldots, N$. (This is an immediate consequence of Lemma 3.4 because, by Lemma 3.3(1), $\Delta_\tau(u^i) < \Delta_\tau(u)$ for all $i \in [k]$.) Since the $\tau[u^i]$ are pairwise distinct (as they differ in the j-th direct subtree) this means that $|T_q| \geq N$ before $\tau[u]$ was enqueued, thus proving that t is red, as claimed.

Since Algorithm 2′ terminates when $|T_{q_\mathrm{f}}| = N$, none of the trees in T_{q_f} will ever be red. By the above, this implies that all subtrees of in trees in T_{q_f} are black as well. As subtrees inherit their colour from the T_q they are taken from, this shows that no red

tree occurring in an execution of Algorithm 2′ can ever have an effect on the output of the algorithm. Hence, the output of Algorithm 2 is the same as that of Algorithm 2′ and the result follows from Lemma 3.4. □

3.5. Time complexity

Recall that the input $M = (Q, \Sigma, R, q_f)$ is assumed to be a wta with m transition rules, n states, and a maximum rank of r among its symbols.

In the complexity analysis, we consider an efficient implementation of Algorithm 2 along the lines illustrated in Figure 2, with priority queues based on heaps [13]. This enables us to implement the following details efficiently:

- Consider a transition $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$. At a given stage of the algorithm, some of the elements $u = (u_1, \ldots, u_k)$ in K_τ may contain elements u_i such that $|T_{q_i}| < u_i$, i.e., $\tau[u]$ is still undefined and hence $\Delta_\tau(u) = \infty$. Thus, $\Delta_\tau(u)$ must be decreased from ∞ to its final value in $\mathbb{R}_+^\infty$ when $|T_{q_i}|$ has reached u_i for all $i \in [k]$. For this, we record for every $p \in Q$ and for $|T_p| < j < |N|$ a list of all $u \in K_\tau$ such that τ is as above, $q_i = p$ for some $i \in [k]$, and $u_i = j$. When $|T_p|$ reaches the value j, this list is used to adjust the priority of each u on that list to the new value of $\Delta_\tau(u)$ (which is either still ∞ or has reached its final value).

- The queue K' that contains the individual queues K_τ as elements is implemented in the straightforward way, also using priority queues based on heaps. As described earlier, the priority between K_τ and $K_{\tau'}$ is given by comparing their top-priority elements: if these elements are u and u', respectively, then K_τ takes priority over $K_{\tau'}$ if $(\Delta_\tau(u), \delta_{tar(\tau)}) < (\Delta_{\tau'}(u'), \delta_{tar(\tau')})$. If one of the queues K_τ runs empty (which can only happen if $rank(\tau) = 0$), then K_τ is removed from K.

We now establish an upper bound on the running time of the algorithm.

Theorem 3.6. *Algorithm 2 runs in time*

$$O(Nm(\log(m) + r^2 + r\log(Nr))) \ .$$

Proof. For the proof, we look at the maximum number of instantiations that we encounter during a run of the algorithm.

Since K' contains (at most) the m queues K_τ, enqueuing into K' is in $O(\log(m))$. Furthermore, each rule is limited to N instantiations, due to the fact that the sequences T_q do not contain repetitions and thus $\tau[u] \neq \tau[u']$ for $u \neq u'$. This implies that the maximum number of iterations of the main loop is Nm, yielding an upper bound of $O(Nm\log(m))$ for the management of K'.

Next, we have the rule-specific queues K_τ. Each time a tuple $u \in \mathbb{N}^k$ is dequeued from K_τ, at most $|inc(u)| = k \leq r$ new tuples are enqueued. In total, the creation of these k tuples of size k each takes $k^2 \leq r^2$ operations. Thus, there will be at most Nr elements in K_τ for any τ, which gives us a time bound of $O(\log(Nr))$ per queue operation, a total time of $O(N(r^2 + r\log(Nr)))$ for the management of K_τ, and thus a total of $O(Nm(r^2 + r\log(Nr)))$ for the m queues K_τ altogether.

Summing up the upper bounds for the management of the two queue types yields

$$O\left(Nm(\log(m) + r^2 + r\log(Nr))\right) \ ,$$

as claimed.

To complete the analysis, we have to argue that the time that needs to be spent to check whether $\tau[u]$ has been outputted before, can be made negligible. We do this by implementing the forest of outputted trees in such a way that equal subtrees are shared. Hence, trees are equal if (and only if) they have the same address in memory. Assuming a good hashing function, the construction of $\tau[u]$ from the (already previously constructed) trees referred to by u, can essentially be done in constant time. Now, if we maintain with every previously constructed tree a flag ✓ indicating whether that tree had already been outputted once, the test boils down to constructing $\tau[u]$ (which would return the already existing tree if it did exist) and checking the flag ✓. □

The running time of Algorithm 2 should be contrasted with the running time

$$O\left(\max(Nmn \cdot (Nr + r\log r + N\log N), N^2n^3, mr^2)\right)$$

of Algorithm 1 [5]. By handling instantiations of transition rules rather than states, we reduce running times roughly by a factor of $O(Nn)$.

Even though the algorithm of Huang and Chiang [8] solves the easier N-best *runs* problem, it could be interesting to note that the upper bound derived by Huang and Chiang is $O(m\log n + dN\log N)$ where d is the size of the largest run among the N results. If we for Algorithm 2 consider r a constant (as is done in [8]), the two upper bounds differ roughly by a factor m. As mentioned in the introduction, Büchse et al. [10] extend the algorithm of Huang and Chiang to structured weight domains. This does not seem to be possible for Algorithm 2 (nor for Algorithm 1). The reason is that the computation and use of best contexts requires that the semiring is extremal, which is not the case for structured weight domains in general. Further comparisons are left to future work.

In practical applications, we expect to see Algorithm 2 used in two ways. The first is, as discussed in the introduction, the situation in which N best trees are computed for a relatively small value of N, for example, $N = 200$ as suggested by Socher et al. [2]. Here, based on a comparison of the upper bounds on the running time, and assuming that they are reasonably tight, Algorithm 2 will outperform Algorithm 1 if N is larger than $\sqrt{m}$, which we believe is the common case. In the second scenario, Algorithm 2 is invoked with a very large N to enumerate the trees recognised by the input automaton, outputting them in ascending order by weight. In this scenario, our exploration by transition rule is even more valuable, as it saves redundant computation.

4. Conclusion

We have presented an improved version of the algorithm in [5] that solves the N-best trees problem for weighted tree automata over the tropical semiring. The main novelty lies in the exploration of the search space with a focus on instantiations of transition rules rather than on states, and the lazy assembly of these instantiations. We

have proved the new algorithm to be correct and have derived an upper bound on its running time which is smaller than that of the previous one. We believe that this speed-up makes it superior for usage in typical language-processing applications.

Acknowledgements

We are thankful to the reviewers for their constructive comments on the original version of this article.

[1] Y. Bar-Hillel, M. Perles, E. Shamir, On Formal Properties of Simple Phrase Structure Grammars, in: Y. Bar-Hillel (Ed.), Language and Information: Selected Essays on their Theory and Application, chap. 9, 116–150, 1964.

[2] R. Socher, J. Bauer, C. D. Manning, A. Y. Ng, Parsing with compositional vector grammars, in: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, vol. 1, 455–465, 2013.

[3] Y. Zhao, L. Zhang, K. Tu, Gaussian Mixture Latent Vector Grammars, in: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Volume 1: Long Papers, 1181–1189, URL `https://aclanthology.info/papers/P18-1109/p18-1109`, 2018.

[4] J. Benesty, M. M. Sondhi, Y. Huang (Eds.), Springer Handbook of Speech Processing, Springer, 2008.

[5] J. Björklund, F. Drewes, N. Zechner, Efficient enumeration of weighted tree languages over the tropical semiring, Journal of Computer and System Sciences 104 (2019) 119–130.

[6] M. Mohri, M. Riley, An Efficient Algorithm for the n-Best-Strings Problem, in: Proceedings of the Conference on Spoken Language Processing (ICSLP 02, 1313–1316, 2002.

[7] J. Björklund, F. Drewes, A. Jonsson, A Comparison of Two N-Best Extraction Methods for Weighted Tree Automata, in: 23rd International Conference on the Implementation and Application of Automata (CIAA 2018), Lecture Notes in Computer Science, 197–108, 2018.

[8] L. Huang, D. Chiang, Better K-best Parsing, in: Proc. of the Conference on Parsing Technology 2005, Association for Computational Linguistics, 53–64, 2005.

[9] J. May, K. Knight, Tiburon: A weighted tree automata toolkit, in: International Conference on Implementation and Application of Automata, Springer, 102–113, 2006.

[10] M. Büchse, D. Geisler, T. Stüber, H. Vogler, n-Best Parsing Revisited, in: Proceedings of the 2010 Workshop on Applications of Tree Automata in Natural Language Processing, Association for Computational Linguistics, Uppsala, Sweden, 46–54, 2010.

[11] D. E. Knuth, A Generalization of Dijkstra's Algorithm, Information Processing Letters 6 (1977) 1–5.

[12] E. W. Dijkstra, A note on two problems in connexion with graph, Numerische Mathematik 1 (1959) 269–271.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, 3rd edn., 2009.

IV

A Comparative Evaluation of the Efficiency of N-Best Algorithms on Language Data

Johanna Björklund*
Umeå University

Frank Drewes**
Umeå University

Anna Jonsson†
Umeå University

The N-best extraction problem consists in selecting the N highest ranking hypotheses from a set of hypotheses, with respect to a given ranking system. In our setting, the hypotheses and ranking are jointly represented by a weighted tree automaton (wta) over the tropical semiring: the hypotheses are trees, or runs on trees, and the ranking is decided by the weight assigned to them. In previous work, we presented an algorithm for N-best extraction that combines techniques to restrict the search space, and proved it to be correct and efficient. The algorithm is now implemented in the software BETTY*, allowing us to complement the deductive study with an empirical investigation. In particular, we compare our algorithm to the state-of-the-art algorithm for extracting the N best runs, implemented in in the software toolkit* TIBURON*. The data sets used in the experiments are wtas resulting from real-world natural language processing tasks, as well as artificially created wtas with varying degrees of nondeterminism. We find that* BETTY *outperforms* TIBURON *on all tested data sets.*

1. Introduction

Trees are standard in natural language processing (NLP) to represent linguistic analyses of sentences. Similarly, tree automata provide a compact representation for a set of such analyses. *Bottom-up tree automata* act as recognizing devices and process their input trees in a step-wise fashion, working upwards from the leaf nodes towards the root of the tree. For memory, they have a finite set of *states*, some of which are said to be *accepting*, and their internal logic is represented as a finite set of *transition rules*. A *run* of a tree automaton M on an input tree t is a mapping from the nodes of t to the states, that is compatible with the transition rules. In general, M can have several distinct runs on t, and *accepts* t if one of these runs maps the root of t to an accepting state. By equipping the transition rules with weights, M can be made to associate t with a likelihood or score: The weight of a run of M on t is the product of the weights of the transition rules used in the run, and the weight of t is the sum of the weights of all runs on t. This type of automaton is called a weighted-tree automaton (wta) and is popular in, e.g., dependency parsing and machine translation.

A central task is the extraction of the highest ranking trees with respect to a weighted tree automaton. For example, when the automaton represents a large set of intermediate solutions, it may be desirable to prune these down to a more manageable number before continuing the

* E-mail: johanna@cs.umu.se.
** E-mail: drewes@cs.umu.se.
† E-mail: aj@cs.umu.se.

computation. This problem is known as the *best trees problem*. The additive operator $\oplus$ that combines weights of runs is *extremal* if $a \oplus b \in \{a, b\}$ for all weights a and b. When the operator has this property, the weight of a tree coincides with the weight of one of its runs, and there is a *natural order* on the weights, given by $a \leq b$ if and only if $a \oplus b = a$. This is for example the case when performing a Viterbi analysis: the transition rules are equipped with probabilities, which are multiplied over a given run to determine the probability of that run, and weights of alternative runs on a tree t are combined using $\oplus = \max$ to obtain the weight of t. An equivalent setting is the one in which transition weights are summed up to obtain the weight of a run, and the run with the minimal weight determines the weight of t.

In these cases, the best trees problem can be solved by computing a list of $N \cdot c$ best runs for M, for some constant c, and discarding runs made on previously outputted trees. A complicating factor is that M can have exponentially many runs on a single tree, so c may have to be very large to guarantee that the output contains N distinct trees.

We note that the best trees problem has applications also outside of NLP. In fact, whenever we are considering a set of objects, each of which can be expressed uniquely by an expression in some particular algebra, and the set of expressions is the language of a wta, then the best trees algorithm can be used to produce the best objects. For instance, Björklund, Drewes, and Ericson (2016) propose a restricted class of hypergraphs that are uniquely described by expressions in a certain graph algebra, so the best trees algorithm makes it possible to find the optimal such graphs with respect to a wta. The reason why the representation needs to be unique is that otherwise we will have to check equivalence between objects as an added step. In the case of graphs this would mean deciding graph isomorphism, which is in general not known to be tractable.

In this work, we report on an empirical comparison of a recent algorithm capable of extracting both the best trees and best runs, and a state-of-the-art algorithm by Huang and Chiang (2005) for best runs. This adds to a line of research that spans two decades. It originates with an algorithm by Mohri and Riley (2002) to compute the N best strings with respect to a weighted string automaton. To reduce the amount of redundant computation, consisting in the exploration of alternative runs on one and the same substring, they incorporate the N shortest paths algorithm by Dijkstra (1959). Moreover, Mohri and Riley work with a *determinized* version of the input automaton M, which can be exponentially larger than M, but has at most one run on each input string. Independently, Huang and Chiang (2005) published an algorithm that computes the best runs for tree automata. The algorithm combines dynamic programming and lazy evaluation to keep the number of intermediate computations small.

The algorithm by Huang and Chiang (2005) is implemented in the widely referenced Tiburon toolkit[1] (May and Knight 2006). From here on, we refer to this implementation simply as TIBURON, even though the best runs procedure is only one out of many that the toolkit has to offer. The initial application for the implementation was as part of a machine-translation pipeline, to extract the N best trees from a weighted tree automaton. In connection with this work, Knight and Graehl (2005) noted that there was no known efficient algorithm to solve this problem directly. As an alternative way forward, Knight and Graehl thus enumerate the best runs and discard duplicate trees, until sufficiently many unique trees have been found. The algorithm by Huang and Chiang (2005) is later generalized by Büchse et al. (2010) to allow a linear pre-order on the weights, as opposed to a total order. Büchse et al. (2010) also prove that the algorithm is correct on cyclic input hypergraphs, provided that the Viterbi algorithm for finding an optimal run (Jurafsky and Martin 2009) is replaced by Knuth's algorithm (Knuth 1974)[2].

1 https://github.com/isi-nlp/tiburon

2 This replacement had already been done in TIBURON but it had not been further commented upon.

Also Finkel, Manning, and Ng (2006) remark on the lack of sub-exponential algorithms for the best trees problem. They propose an algorithm that approximates the solution when the automaton is expressed as a cascade of probabilistic tree transducers. The authors model the cascade as a Bayesian network and consider every step as a variable. This allows them to sample a set of alternative labels from each prior step, to propagate in the current step. The approximation algorithm runs in polynomial time in the size of the input device and the number of samples, but the convergence rate to the exact solution is not analyzed. To explain why their approach is preferable to finding N best runs, they extract the $N = 50$ best runs from the Stanford parser and observe that about half of the output trees are actually duplicates – enough to affect the outcome of the processing pipeline. Thus, they argue, extracting the highest ranking trees rather than the highest ranking runs is not only theoretically better, but is also of practical significance.

Björklund, Drewes, and Zechner (2019) generalize the best strings algorithm by Mohri and Riley (2002), resulting in the algorithm BEST TREES V.1 that solves the best trees problem in polynomial time. The algorithm includes an on-the-fly determinization that serves to keep the object automaton small, and is presented together with formal correctness and complexity proofs. In contrast to the current paper, which focuses mainly on the N-best runs problem, Björklund, Drewes, and Jonsson (2018) compare BEST TREES V.1 with TIBURON on the N-best trees problem, letting TIBURON enumerate runs until N distinct trees have been found. The conclusion is that TIBURON performs better when the input automaton is largely deterministic, that is, when there is a small number of runs on each input tree, but when the input automaton shows a higher degree of nondeterminism, meaning that number of runs grows rapidly in the size of the input tree, BEST TREES V.1 is faster.

Björklund, Drewes, and Jonsson (2019) continue to improve BEST TREES V.1 by queuing intermediate results with respect to transition rules rather than states, leading to the algorithm BEST TREES V.2. Henceforth, we will refer to the latter by the name BEST TREES. In addition to solving the best trees problem, BETTY can also solve the best runs problem. In this paper, we make use of this possibility in order to compare BEST TREES, implemented as BETTY[3], with TIBURON with respect to the computation of best runs. For our experiments, we use both largely deterministic wta from a machine translation project and from a grammatical framework, and more nondeterministic wta that were artificially created to expose the algorithms to challenging instances. Our results show that BETTY is generally more efficient than TIBURON, despite the fact that the former is more general as it can also compute best trees (with almost the same efficiency as that it computes best runs with).

2. Preliminaries

In this paper, $\mathbb{N}$ denotes the set of nonnegative integers (including zero), and $\mathbb{R}_+$ stands for the set of non-negative reals. We furthermore let $\mathbb{R}_+^\infty$ denote $\mathbb{R}_+ \cup \{\infty\}$. A sequence of 1's of length l is denoted by 1^l. For $n \in \mathbb{N}$, the set $\{i \in \mathbb{N} \mid 1 \leq i \leq n\}$ is abbreviated by $[n]$; in particular, $[0] = \emptyset$. As usual, the set of all strings (or: finite sequences) over a set A is denoted by A^*, and the empty string by λ.

A *ranked alphabet* is a finite set Σ of symbols partitioned into disjoint subsets $\Sigma_{(k)}$, $k \in \mathbb{N}$, i.e., $\Sigma = \biguplus_{k\in\mathbb{N}} \Sigma_{(k)}$. The *rank* of $f \in \Sigma$, denoted by $\mathrm{rank}(f)$, is the unique $k \in \mathbb{N}$ for which $f \in \Sigma_{(k)}$. Given such a ranked alphabet Σ and any set T, we denote by $\Sigma(T)$ the set of formal expressions

$$\Sigma(T) = \{f[t_1, \ldots, t_k] \mid k \in \mathbb{N}, f \in \Sigma_{(k)}, \text{ and } t_1, \ldots, t_k \in T\} \ .$$

3 https://github.com/tm11ajn/betty

The set of *trees over* Σ is the smallest set T_Σ such that $T_\Sigma = \Sigma(T_\Sigma)$.

Throughout the paper, let $\Box \notin \Sigma$ be a special symbol of rank 0. The set C_Σ of *contexts over* Σ is the set of trees in $T_{\Sigma\cup\{\Box\}}$ containing exactly one occurrence of $\Box$. We define the *depth* of $c \in C_\Sigma$, denoted by $\mathrm{depth}(c)$, and the *substitution* of a tree t for $\Box$ in c, denoted by $c[\![t]\!]$, as follows: If $c = \Box$, then $\mathrm{depth}(c) = 0$ and $c[\![t]\!] = t$. Otherwise, if $c = f[t_1, \ldots, t_{i-1}, c', t_{i+1}, \ldots, t_k]$ with $c' \in C_\Sigma$, then $\mathrm{depth}(c) = \mathrm{depth}(c') + 1$ and $c[\![t]\!] = f[t_1, \ldots, t_{i-1}, c'[\![t]\!], t_{i+1}, \ldots, t_k]$.

A *weighted tree language* (over the tropical semiring) is a mapping $L\colon T_\Sigma \to \mathbb{R}_+^\infty$. Such languages can be specified by *weighted tree automata*. Here, we take a slightly different but equivalent view corresponding to the one of Huang and Chiang (2005) and Büchse et al. (2010) in order to facilitate comparison.

Recall that ordinary finite automata can be represented as state-transition diagrams, i.e., directed edge-labeled graphs whose nodes and edges correspond to states and transitions, respectively. A path from a (node representing an) initial state to a final one corresponds to an accepting run of the automaton on the string of edges labels on that path. If the string automaton is weighted, additional weights are assigned to the edges, and those are summed up along a path, yielding the weight of the run. For a given string, the minimal weight of a run on that string is the weight of the string (and ∞ if there is no such run).

In a similar way, (weighted) tree automata can be represented as hypergraphs whose hyperedges generalize ordinary binary edges in that they may have any number of source nodes, but still only one target node. A hyperedge labeled with an input symbol of rank k must have k source nodes. The formal definition reads as follows.

Definition 1
Let Σ be a ranked alphabet. A *weighted hypergraph* over Σ and $\mathbb{R}_+$ (for short: hypergraph) is a triple $G = (V, E, \mathrm{wt})$ where:

- V is a finite sets of *nodes*,
- $E \subseteq V^* \times \Sigma \times V$ is the set of *hyperedges*, and
- $\mathrm{wt}\colon E \to \mathbb{R}_+$ assigns a weight to every hyperedge.

For $e = (v_1 \cdots v_k, f, v) \in E$, we define $\mathrm{rank}(e) = k$ (thus turning E into a ranked alphabet) and require that $\mathrm{rank}(e) = \mathrm{rank}(f)$. We denote the three components of e by $\mathrm{tail}(e)$, $\mathrm{lab}(e)$, and $\mathrm{head}(e)$, i.e., $\mathrm{tail}(e) = v_1 \cdots v_k$, $\mathrm{lab}(e) = f$ and $\mathrm{head}(e) = v$. The homomorphisms $\mathrm{wt}\colon T_E \cup C_E \to \mathbb{R}_+$ and $\mathrm{result}\colon T_E \cup C_E \to T_\Sigma$ are given as follows:

1. $\mathrm{wt}(\Box) = 0$ and $\mathrm{result}(\Box) = \Box$.
2. For $\rho = e[\rho_1, \ldots, \rho_k] \in T_E \cup C_E$, where $e \in E$,

$$\mathrm{wt}(\rho) = \mathrm{wt}(e) + \sum_{i\in[k]} \mathrm{wt}(\rho_i) \text{ and } \mathrm{result}(\rho) = \mathrm{lab}(e)[\mathrm{result}(\rho_1), \ldots, \mathrm{result}(\rho_k)] \ .$$

Next, we define *runs*, which in the hypergraph setting become trees over hyperedges.

Definition 2
Let $G = (V, E, \mathrm{wt})$ be a hypergraph. For $v \in V$, the sets $\mathrm{runs}_G(v)$ of *runs arriving at* v are the smallest subsets of T_E such that, for every edge $e \in E$ with $\mathrm{tail}(e) = v_1 \cdots v_k$, if $\rho_1 \in \mathrm{runs}_G(v_1), \ldots, \rho_k \in \mathrm{runs}_G(v_k)$ then $e[\rho_1, \ldots, \rho_k] \in \mathrm{runs}_G(\mathrm{head}(e))$.

For a final node[4] $v_{\mathrm{f}} \in V$, the *weighted tree language accepted at* v_{f} *in* G is given by

$$L_G^{v_{\mathrm{f}}(t)} = \min\{\mathrm{wt}(\rho) \mid \rho \in \mathrm{runs}_G(v_{\mathrm{f}}) \text{ and } \mathrm{result}(\rho) = t\}$$

for every tree $t \in T_\Sigma$.

Given a hypergraph G as above, a final node $v_{\mathrm{f}} \in V$, and some $N \in \mathbb{N}$, the *N-best runs problem* asks to compute a sequence of N runs of smallest possible weight in $\mathrm{runs}_G(v_{\mathrm{f}})$. More precisely, the goal is to compute a sequence of pairwise distinct runs $\rho_1, \rho_2, \ldots, \rho_N \in \mathrm{runs}_G(v_{\mathrm{f}})$ such that there do not exist $i \in [N]$ and $\rho \in \mathrm{runs}_G(v_{\mathrm{f}}) \setminus \{\rho_1, \ldots, \rho_i\}$ with $\mathrm{wt}(\rho) < \mathrm{wt}(\rho_i)$. Similarly, the *$N$-best trees problem* asks to compute a sequence of pairwise distinct trees $t_1, t_2, \ldots, t_N$ in T_Σ of smallest possible weight, i.e., a sequence of trees such that there do not exist $i \in [N]$ and $t \in T_\Sigma \setminus \{t_1, \ldots, t_i\}$ with $L_G^{v_{\mathrm{f}}(t)} < L_G^{v_{\mathrm{f}}(t_i)}$. To make sure that the two problems always have a solution, we assume in the following that $\mathrm{runs}_G(v_{\mathrm{f}})$ is infinite. Further, we assume that every $e \in E$ occurs in some run of $\mathrm{runs}_G(v_{\mathrm{f}})$. In automata terminology, this latter condition means that the automaton does not contain useless states, and it can be ensured in the usual way.

We end this section by discussing the choice of our particular weight structure, the tropical semiring. In general, a commutative semiring is a structure $(\mathbb{D}, \oplus, \otimes, 0, 1)$ such that both $(\mathbb{D}, \oplus, 0)$ and $(\mathbb{D}, \otimes, 1)$ are commutative monoids, $\otimes$ distributes over $\oplus$ from both left and right, and 0 is an annihilator for $\otimes$, i.e., $0 \otimes d = 0 = d \otimes 0$ for all $d \in \mathbb{D}$. With this, Definitions 1 and 2 can be generalized to weighted hypergraphs over arbitrary commutative semirings, and their resulting weighted tree languages, simply by replacing $+$ and $\max$ by $\oplus$ and $\otimes$, respectively.

The semiring used in Definitions 1 and 2 is the *tropical semiring* $(\mathbb{R}_+^\infty, \min, +, \infty, 0)$, which is frequently used in NLP. Equally popular is the *Viterbi semiring* $([0,1], \max, \cdot, 0, 1)$ that acts on the unit interval of probabilities, with maximum and standard multiplication as operations. In the setting discussed here, both semirings are equivalent. To see this, transform a hypergraph G over the Viterbi semiring to a hypergraph G over the tropical semiring by simply mapping every hyperedge weight p to $-\ln p$, i.e., taking negative logarithms everywhere. Since $-\ln p + -\ln p' = -\ln(p \cdot p')$ and $-\ln p < -\ln p' \iff p > p'$ it holds that $L_G^{v_{\mathrm{f}}(t)}$ (now calculated using the Viterbi semiring operations) is equal to $\exp(-L_H^{v_{\mathrm{f}}(t)})$ for all trees t, and trees $t_1, \ldots, t_N$ form an N-best list according to $L_G^{v_{\mathrm{f}}}$ (now looking for trees with *maximal* weights) if and only if they form an N-best list according to $L_H^{v_{\mathrm{f}}}$ in the sense defined above.

3. Algorithms

In this section we relate BEST TREES to the algorithms of Huang and Chiang (2005) and Büchse et al. (2010).

3.1 The algorithm BEST TREES

Let us first describe the algorithm BEST TREES by Björklund, Drewes, and Jonsson (2019) in the hypergraph setting. Its pseudocode is outlined in Algorithm 1. As input it takes a triple (G, v_{f}, N), where $G = (V, E, \mathrm{wt})$ is a hypergraph, $v_{\mathrm{f}} \in V$ is a final node, and $N \in \mathbb{N}$. The objective is to solve the N-best trees problem or, for the sake of comparison with the TIBURON implementation of the algorithm by Huang and Chiang (2005), the N-best runs problem. From the point of view of computational complexity, the latter is the easier problem of the two. Essentially, the difference is that an algorithm solving the former must discard duplicate trees

4 We use this terminology because v_{f} corresponds to the *final state* of a tree automaton represented by G.

Algorithm 1 Compute $N \in \mathbb{N}$ best trees given a hypergraph $G = (V, E, \mathrm{wt})$ and a final node $v_\mathrm{f} \in V$ of final nodes

procedure BEST TREES(G, v_f, N)
  compute $(c_v, t_v^{\mathrm{best}})$ for all $v \in V$
  $T_v \leftarrow$ LIST-MAKE(t_v^{best}) for all $v \in V$
  for $e \in E$ **do**
    $U \leftarrow \begin{cases} \mathrm{inc}(\langle i_1, \ldots, i_{\mathrm{rank}(e)} \rangle) & \text{if } e = c_{\mathrm{head}(e)}(\lambda) \\ \{1^{\mathrm{rank}(e)}\} & \text{otherwise} \end{cases}$
    $K_e \leftarrow$ HEAP-MAKE(U) for all $e \in E$
  end for
  $K \leftarrow$ HEAP-MAKE($\{K_e \mid e \in E\}$)
  output($t_{v_\mathrm{f}}^{\mathrm{best}}$)
  repeat
    $K_e \leftarrow$ HEAP-EXTRACT-MIN(K)
    $(i_1, \ldots, i_{\mathrm{rank}(e)}) \leftarrow$ HEAP-EXTRACT-MIN(K_e)
    HEAP-INSERT(K, K_e)
    $t_i \leftarrow$ LIST-GET($T_{\mathrm{head}(e)}, i_j$) for $j \in [\mathrm{rank}(e)]$
    $t \leftarrow \mathrm{lab}(e)[t_1, \ldots, t_{\mathrm{rank}(e)}]$
    if t is not yet in $T_{\mathrm{head}(e)}$ **then**
      LIST-ADD($T_{\mathrm{head}(e)}, t$)
    end if
    if $\mathrm{head}(e) = v_\mathrm{f}$ **then**
      if t has not yet been outputted **then**
        output(t)
      end if
    end if
    HEAP-INSERT($K_e, \mathrm{inc}(\langle i_1, \ldots, i_{\mathrm{rank}(e)} \rangle)$)
    HEAP-UPDATE-PRIORITIES(K)
  until N trees have been outputted
end procedure

encountered, that is, equal results of distinct runs. For the sake of efficiency, this should be done as early as possible in the computation.

A central fact underlying Algorithm 1 is that, for every $v \in V$, there is a *best context* $c_v \in C_E$ such that

1. $c_v[\![\rho]\!] \in \mathrm{runs}_G(v_\mathrm{f})$ for every run $\rho \in \mathrm{runs}_G(v)$ and
2. among all contexts that satisfy the preceding condition, c_v minimizes $(\mathrm{wt}(c_v), \mathrm{depth}(c_v))$ with respect to the lexicographic order[5].

Suitable contexts c_v can be efficiently computed by a procedure based on an algorithm by Knuth (1974). Here, the role of Knuth's algorithm is to compute, for every node $v \in V$, a best tree t_v^{best}, that is, a tree t which minimizes $L_G^{v(t)}$. In fact, Algorithm 1 does not make much use of the contexts c_v themselves but mainly uses the pairs $(\omega_v, \delta_v) = (\mathrm{wt}(c_v), \mathrm{depth}(c_v))$ for every $v \in V$. Intuitively, c_v is the most economical way to extend any run ρ arriving at v to a run ρ' arriving at v_f, minimizing the weight of ρ'. Note that this depends only on v and not on ρ itself. As we will see below, we can use the pairs (ω_v, δ_v) to guide the search for best trees.

To understand Algorithm 1, let us first consider the easier N-best runs problem. For every $v \in V$, let $T_v(1), T_v(2), \ldots$ be an enumeration (without repetition) of $\mathrm{runs}_G(v)$ according

5 A pair $(w, n) \in \mathbb{R}_+ \times \mathbb{N}$ is lexicographically less than another such pair (w', n') if $w < w'$ or $w = w'$ and $n < n'$.

to increasing weight. Then, for a hyperedge $e \in E$ with $\mathrm{tail}(e) = v_1 \cdots v_k$, every run $\rho = e[\rho_1, \ldots, \rho_k]$ can uniquely be specified by e and the tuple $(i_1, \ldots, i_k)$ such that $\rho_j = T_{v_j}(i_j)$ for all $j \in [k]$. We write $\rho = e\langle i_1, \ldots, i_k\rangle$. Now, in order to *construct* the lists T_v run by run, we keep, for every hyperedge e as above, a queue K_e of those tuples $(i_1, \ldots, i_k)$ for which $e\langle i_1, \ldots, i_k\rangle$ is not yet in $T_{\mathrm{head}(e)}$. For a given $(i_1, \ldots, i_k)$ in K_e, $e\langle i_1, \ldots, i_k\rangle$ may not yet be available because some T_{v_j} does not yet contain i_j runs. Call those which *are* available the *current candidates*, and consider, for every run $\rho \in \mathrm{runs}_G(v)$, the pair $\Pi(\rho) = (\omega_v + \mathrm{wt}(\rho), \delta_v)$, where ω_v and δ_v are given by the best context c_v for v as discussed above. It turns out that if we, among the set of all current candidates ρ, always choose the one that minimizes $\Pi(\rho)$ and append it to the appropriate T_v, then this ρ is the one that minimizes $\Pi(\rho)$ among *all* runs not yet in any T_v. Hence, ρ is indeed an appropriate next element of T_v. In addition, if $v = v_\mathrm{f}$, ρ is written to the output.

To make the procedure efficient, we make two central observations (see (Björklund, Drewes, and Jonsson 2019) for details). First, in order to find N best runs, no more than the first N elements of each T_v need to be stored; any additional ones can be discarded should they be encountered as candidates. Second, $i_1 \leq j_1, \ldots, i_k \leq j_k$ implies $\mathrm{wt}(e\langle i_1, \ldots, i_k\rangle) \leq \mathrm{wt}(e\langle j_1, \ldots, j_k\rangle)$, which means that at each moment in time only the minimal still unused elements of $\mathbb{N}^k$ need to be kept in K_e, for each $e \in E$ of rank k. (Here, *minimal* refers to the partial order given by $(i_1, \ldots, i_k) \leq (j_1, \ldots, j_k)$ if and only if $i_\ell \leq j_\ell$ for all $\ell \in [k]$.)

Consider a hyperedge e with $\mathrm{tail}(e) = v_1 \cdots v_k$ and $\mathrm{head}(e) = v$. We need quick access to a tuple $(i_1, \ldots, i_k)$ in K_e that determines a current candidate $\rho = e\langle i_1, \ldots, i_k\rangle$ which minimizes $\Pi(\rho) = (\omega_v + \mathrm{wt}(\rho), \delta_v)$. We do this by defining the priority of $(i_1, \ldots, i_k)$ in K_e to be $\Pi(\rho)$ if $|T_{v_j}| \geq i_j$ for all $j \in [k]$ (i.e., if ρ is indeed a current candidate). Otherwise, the priority of $(i_1, \ldots, i_k)$ is $(\infty, 0)$, in effect disabling this potential candidate until it eventually becomes available. Finally, to be able to access the run $\rho = e\langle i_1, \ldots, i_k\rangle$ that, across all K_e, minimizes $\Pi(\rho)$, we also need quick access to the queue K_e whose highest-priority element has the highest priority across all K_e. To facilitate this, we keep a *main queue* K whose elements are the individual queues K_e, the priority of each K_e in K being the priority of its highest priority element $(e, \langle i_1, \ldots, i_k\rangle)$.

To obtain an algorithm computing best trees rather than best runs, we discard runs resulting in duplicate trees as soon as possible. That is, a candidate run ρ is appended to T_v at position $i + 1 \leq N$ only if $\mathrm{result}(\rho) \neq \mathrm{result}(T_v(j))$ for all $j \in [i]$. Moreover, $\mathrm{result}(\rho)$ is written to the output only if $v = v_\mathrm{f}$ and $\mathrm{result}(\rho)$ has not been outputted before (as the result of another run arriving at v_f).

The pseudocode in Algorithm 1 shows the main steps of the procedure, using min-heaps to implement priority queues. The initialization phase prior to the main loop computes a best context c_v and a best tree t_v^{best} for every node v. The tree t_v^{best} is inserted in T_v. Every queue K_e initially either contains the element $1^{\mathrm{rank}(e)}$ or, if e happens to be the edge used to arrive at v in the best tree for v, all of the successors of $1^{\mathrm{rank}(e)}$. (See the end of Section 3.3 for an alternative that allows us to avoid the initialization of each T_v with t_v^{best}, instead initializing T_v as the empty list and K_e as the queue containing only $1^{\mathrm{rank}(e)}$ for all $v \in V$ and $e \in E$.)

We note that line 13 re-inserts queue K_e into K, which is needed because K_e has just been extracted from K as its top-most element, and must thus be re-inserted with a lower priority as its own top-most element $(e, \langle i_1, \ldots, i_{\mathrm{rank}(e)}\rangle)$ was extracted. Line 24 inserts the set of all $(e, \langle j_1, \ldots, j_{\mathrm{rank}(e)}\rangle$ into e such that $j_\ell = i_\ell + 1$ for exactly one $\ell \in [\mathrm{rank}(e)]$ and $j_{\ell'} = i_{\ell'}$ for all $\ell' \in [\mathrm{rank}(e)] \setminus \{\ell\}$.[6] Finally, line 25 increases the priorities of all elements of K (i.e., of all

6 To avoid duplicates to appear in K_e, we implement inc in such a way that it returns only those $\langle j_1, \ldots, j_{\mathrm{rank}(e)}\rangle$ which are not yet present in K_e.

K_e) in which the priority of the top-most element has changed due to the addition of t to $T_{\mathrm{head}(e)}$ or due to the insertion of new tuples in line 24.

Algorithm 1 solves the N-best runs problem instead of the N-best trees problem if we replace $\mathrm{lab}(e)$ by e on line 15 and make lines 17 and 21 unconditional.

3.2 The algorithm BEST RUNS

We now recall the algorithm by Huang and Chiang (2005), which we henceforth will refer to as BEST RUNS. The type of hypergraph used as input to BEST RUNS differs slightly from the one used in the previous section in that the admissible weight structures are not restricted to the tropical semiring. Instead, each hyperedge e is equipped with a weight function $\mathrm{wt}_e \colon \mathbb{R}_+^{\mathrm{rank}(e)} \to \mathbb{R}_+$. The definition of the weight of a run $\rho = e[\rho_1, \ldots, \rho_k]$ is then changed to $\mathrm{wt}(\rho) = \mathrm{wt}_e(\mathrm{wt}(\rho_1), \ldots, \mathrm{wt}(\rho_k))$.

For the algorithm to work, these weight functions wt_e are required to be monotonic: $\mathrm{wt}_e(w_1, \ldots, w_k) \geq \mathrm{wt}_e(w'_1, \ldots, w'_k)$ whenever $w_i \geq w'_i$ for all $i \in [k]$. Definition 1, which BEST TREES is based on corresponds to the special case where each of these weight functions is of the form $\mathrm{wt}_e(w_1, \ldots, w_k) = w + \sum_{i \in [k]} w_i$ for a constant w. In other words, the weight functions BEST TREES can work with are a restriction of those BEST RUNS works on. This will be discussed in Section 3.3.

The input to BEST RUNS is a triple (G, v_{f}, N), where the components are as for BEST TREES. The algorithm is outlined in Algorithm 2. Line 2 is a preprocessing step that can be performed in $O(|E|)$ time using the Viterbi algorithm, given that the rank of the alphabet used is considered a constant. A list $res[v]$ is used to store, for each node $v \in V$, the at most N discovered best runs arriving at v. The search space is represented by an array of heaps, here denoted $cands$. For each node v, $cands[v]$ holds a heap storing the (at most N) best, so far unexploited, runs arriving at v. That is, if we have already picked the M best runs arriving at a node, the heap lets us pick the next best unpicked candidate efficiently when so requested by the recursive call.

To expand the search space, the M-th best run ρ is used as follows: if $\mathrm{tail}(e) = v_1 \cdots v_k$, we obtain a new candidate by replacing the i-th direct subtree of ρ with the next (and thereby minimally worse) run in res arriving at v_i.

3.3 Comparison

A major difference between BEST RUNS and BEST TREES is that the former solves the N-best runs problem whereas the latter solves the N-best trees problem: on lines 16 and 20 of Algorithm 1, duplicate trees are discarded. If these conditions are removed, BEST TREES solves the best runs problem. (Strictly speaking, a further change required for this is that the tree t constructed on line 15 must be defined as $e[t_1, \ldots, t_{\mathrm{rank}(e)}]$ rather than as $\mathrm{lab}(e)[t_1, \ldots, t_{\mathrm{rank}(e)}]$.)

Following Huang and Chiang (2005), let us assume that the ranked alphabet Σ is fixed. The running time of BEST TREES is $O(N\,|E|\,(\log|E| + \log N))$ (Björklund, Drewes, and Jonsson 2019). Since $|E| \in O(|V|^{r+1})$, where r is the maximal rank of symbols in Σ, it follows that $\log|E| \in O(\log|V|)$, so the second expression simplifies to $N\,|E|\,(\log|V| + \log N) \leq |E| \log|V|\, N \log N$. As shown by Huang and Chiang (2005), the worst case running time of BEST RUNS is $O(|E| \log|V| + mN \log N)$ where m is the size of the largest run among the

Algorithm 2 Compute $N \in \mathbb{N}$ runs arriving at $v_\mathrm{f} \in V = [|V|]$ of minimal weight according to a hypergraph $G = (V, E, \mathrm{wt})$ (Huang and Chiang 2005).

```
 1: procedure BEST RUNS(G, v_f, N)
 2:     Compute 1-best run ρ_v^best arriving at v, for every v ∈ V
 3:     res[v] ← LIST-EMPTY() for all v ∈ V
 4:     LIST-ADD(res[v], ρ_v^best) for all v ∈ V
 5:     BEST RUNS_0(v_f, N)
 6:     for ρ ∈ res[v_f] do
 7:         output(ρ)
 8:     end for
 9: end procedure
10:
11: procedure BEST RUNS_0(v, N)
12:     if |res[v]| ≥ N then
13:         return
14:     end if
15:     if cands[v] is undefined then
16:         tmp ← SORT-BY-WEIGHT({e[ρ_{v_1}^best, ..., ρ_{v_k}^best] | e ∈ E and v_1, ..., v_k ∈ V where
                                                    head(e) = v and v_1 ··· v_k = tail(e)})
17:         cands[v] ← HEAP-IFY(tmp[1] ... tmp[N])
18:         LIST-ADD(res[v], HEAP-EXTRACT-MIN(cands[v]))
19:     end if
20:     while LIST-SIZE(res[v]) < N and HEAP-SIZE(cands[v]) > 0 do
21:         s ← LIST-SIZE(res[v])
22:         ρ ← LIST-GET(res[v], s)                          ▷ the s-best run arriving at v
23:         NEXT RUNS(cands[v], ρ)                           ▷ build new candidates from ρ
24:         LIST-ADD(res[v], HEAP-EXTRACT-MIN(cands[v]))
25:     end while
26: end procedure
27:
28: procedure NEXT RUNS(vcands, ρ = (e[ρ_1, ..., ρ_rank(e)]))
29:     for i ← 1, ..., rank(e) do
30:         list ← res[root(ρ_i)]
31:         N' ← index of ρ in res[root(ρ)]
32:         BEST RUNS_0(root(ρ_i), N')                       ▷ make sure list has N' ≤ N elements
33:         if N' ≤ LIST-SIZE(list) then
34:             ρ'_i ← LIST-GET(list, N')
35:             ρ' ← e[ρ_1, ..., ρ_{i-1}, ρ'_i, ρ_{i+1}, ..., ρ_rank(e)]
36:             if ρ' ∉ vcands then
37:                 HEAP-INSERT(vcands, ρ')
38:             end if
39:         end if
40:     end for
41: end procedure
```

N results.[7] Assuming that N is large enough to make the second term the dominating one, the difference between both running times is thus a factor of $(|E| \log |V|)/m$ (assuming for the sake of the comparison that the given bounds are reasonably tight). A further comparison between the running times does not appear to be all that meaningful because m depends on both N and the structure of G, and the algorithms are specialized on different problems.

7 In fact, the running time obtained by Huang and Chiang (2005) is $O(|E| + mN \log N)$, but this assumes G to be acyclic. When G is cyclic, line 2 must be implemented by using Knuth's algorithm, resulting in an additional factor $\log |V|$ in the first term.

A conceptual comparison of the two algorithms may be more insightful. The algorithms differ mainly in two ways. The first difference is that, while BEST RUNS enumerates candidates of best runs using one priority queue per node of G, BEST TREES uses a more fine-grained approach maintaining one priority queue per hyperedge. Since the upper bound on the length of queues is $O(N)$ in both cases, in total BEST TREES may need to handle $O(N\,|E|)$ candidates whereas BEST RUNS needs only $O(N\,|V|)$. This ostensible disadvantage of BEST TREES is not a real one as it occurs only when solving the N-best trees problem. The difference vanishes if the algorithm is used to compute best runs (by changing lines 16 and 20). To see this, consider a given node $v \in V$. Each time a queue element $(i_1, \ldots, i_{\mathrm{rank}(e)})$ with $\mathrm{head}(e) = v$ is dequeued from a queue K_e, the corresponding run is appended to the list T_v on line 17, and $\mathrm{rank}(e)$ queue elements are inserted on line 24. As this can only happen at most N times per node v, in total only $O(N\,|V|)$ queue elements are ever created in the worst case. In other words, if BEST TREES is set to solve the N-best runs problem, the splitting of queues does not result in a disadvantage compared to BEST RUNS.

The second conceptual difference between the algorithms is that BEST TREES adopts an optimization technique known from the N-best strings algorithm by Mohri and Riley (2002). It precomputes and uses (the weight and depth of) a best context c_v for every node v in order to explore the search space in a more goal-oriented fashion.[8] (Recall the discussion in the second paragraph of Section 3.1.) Let ℓ be the maximum of the distances of nodes in V to v_f, i.e.,

$$\ell = \max_{v \in V} \mathrm{dist}(v, v_\mathrm{f}),$$

where $dist(v, v_\mathrm{f})$ denotes the length of the shortest sequence of hyperedges $e_0 \cdots e_n$ such that $v \in [\mathrm{tail}(e_0)]$, $\mathrm{head}(e_{i-1}) \in [\mathrm{tail}(e_i)]$ for all $i \in [n]$, and $v_\mathrm{f} = \mathrm{head}(e_n)$. Then, at every point in time at most ℓ further loop executions are made before the next best run is outputted. To see this, consider an execution of the main loop, in which a run $\rho = e[\cdots]$ arriving at a node v is constructed. The priority of the corresponding queue element on line 12 is given by $(\mathrm{wt}(\rho) + w, d)$, where w and d are the weight and depth of c_v. If $v \neq v_\mathrm{f}$, then $c_v \neq \Box$, and thus it is of the form $c_u[\![e'[\rho_1, \ldots, \rho_{i-1}, \Box, \rho_{i+1}, \ldots, \rho_k]]\!]$, where the ρ_j are the 1-best runs arriving at their respective nodes (see Figure 1).

It follows that line 24 inserts the element into K_e that represents the run $\rho' = e'[\rho_1, \ldots, \rho_{i-1}, \rho, \rho_{i+1}, \ldots, \rho_k]$. Since each of the runs ρ_i, arriving at some node u_i, is already in the respective list T_{u_i}, the run ρ' immediately becomes a current candidate arriving at u. The corresponding pair $(w_u, d_u) = (\mathrm{wt}(c_u), \mathrm{depth}(c_u))$ satisfies $w_u + \mathrm{wt}(\rho') = w + \mathrm{wt}(\rho)$ and $d_u = d - 1$. Hence, ρ' (or another run with the same priority) will be picked in the next loop execution, meaning that after at most ℓ steps the node arrived at by the constructed run will be v_f, i.e., the run will be outputted. This also shows that queues whose elements are not needed for generating output trees, will never become filled beyond their initial element.

The reader may have noticed that the discussion above also explains why the best trees t_v^{best}, $v \in V$, are inserted into each T_v prior to the start of the main loop, rather than letting T_v be empty and initializing K_e to $1^{\mathrm{rank}(e)}$ for all $v \in V$ and $e \in E$: the latter would not guarantee that at most ℓ iterations are made until a run is outputted because some ρ_i may still not be in $T_{\mathrm{head}(e)}$. If this happens, edges of weight 0 may repeatedly be picked because e' in the reasoning above is not yet enabled. However, with a trick the initialization can nevertheless be simplified in the way mentioned. The idea is to delay the execution of line 25 for every queue K_e until e has actually appeared in an output tree. Thus, until this has happened, K_e is

8 The possibility to use this optimization depends on the use of the tropical semiring as weight structure. That does not seem to be possible in the more general setting of Büchse et al. (2010).

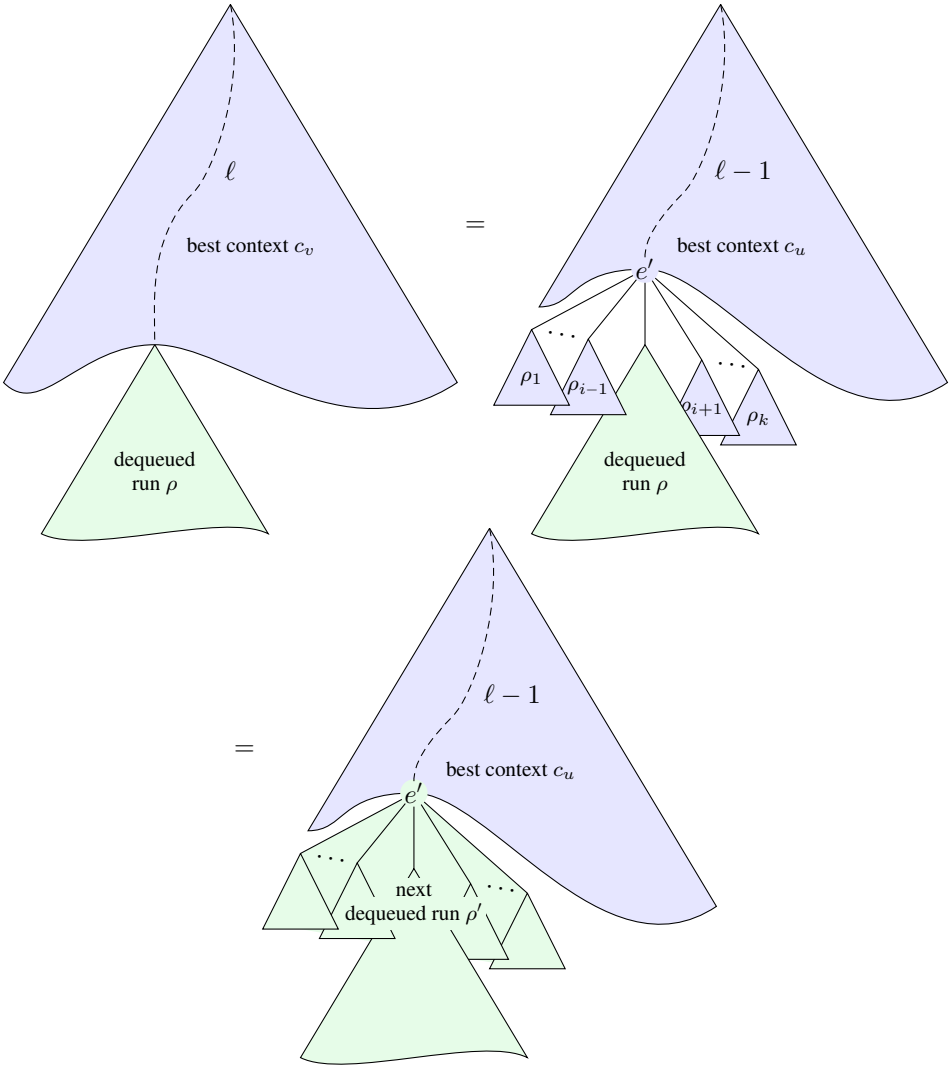


Figure 1
When a run ρ reaching node v is dequeued with $\ell > 0$ (top left), then its best context c_v is of the form $c_u[\![e'[\rho_1, \ldots, \rho_{i-1}, \square, \rho_{i+1}, \ldots, \rho_k]]\!]$, where c_u is the best context for $u = \mathrm{head}(e')$ and the ρ_i are 1-best trees reaching the remaining tail nodes of e' (top right). Hence, $\rho' = e'[\rho_1, \ldots, \rho_{i-1}, \rho, \rho_{i+1}, \ldots, \rho_k]$ (or a run which likewise is less than ℓ steps away from an output run) will be the next run dequeued (bottom).

disabled from contributing another tree to $T_{\mathrm{head}(e)}$. This variant turned out to have efficiency advantages in practice and has therefore been implemented in BETTY. Another slight advantage of this variant is that BETTY can handle a *set* of final nodes rather than only a single one. This is not conveniently possible when using the initialization of Algorithm 1. The reason is that line 9 would have to be generalized to outputting the best trees of all final nodes, which cannot be done while maintaining the correctness of the algorithm, because the second best tree for a node v may have a lesser weight than the best tree for another node v'.

4. Experiments

For each problem instance (i.e., combination of input file and value of N), we perform a number of test runs, measure the elapsed time, and compute the average over the test runs. To avoid noise in our data caused by, e.g., garbage collection, we only measure the time used by the actual application by timing its thread. Moreover, we disregard the time it takes to read the input files.

The number of test runs that are performed per problem instance is decided by the relationship between the mean μ and the variance σ of the recorded times – these values are computed every fifth test run, and to finish the testing of the current problem instance, we require that $\sigma < 0.01\mu$. However, five test runs per problem instance has shown to be sufficient to fulfill the requirement for practically all instances seen.

All test scripts are written in Python. We run the experiments on a computer with a 3.60 Hz Intel Core i7-4790 processor.

4.1 Corpora

The corpora that were used in our experiments (see the list below) contain both real-world and synthetic data. The first corpus is derived from an actual machine-translation system and is thus representative for real-world usage. The second corpus consists of manually engineered grammars for a set of natural languages, used in a range of research and industry applications. The last two corpora are artificially created for the purpose of investigating the effect of increasing degrees of nondeterminism. Let us now present each in closer detail.

`MT-data` Tree automata resulting from an English-to-German machine-translation task, described in the doctoral thesis of Quernheim (2017). The data consists of 927 files, each file containing a wta corresponding to one sentence – the files are indexed from 0 to 926. The smallest file has 24 lines and the largest one has 338 937 lines; all lines but the first hold a transition rule. Moreover, these wtas have a large number of states and are practically deterministic in the sense that each state corresponds to a particular part-of-sentence structure and input node.

`GF-data` Unweighted context-free grammars from the Grammatical Framework (GF) by Ranta (2011). GF is, among other things, a programming language and processing platform for multilingual grammar applications. It provides combinatory categorial grammars (Steedman 1987) for more than 60 natural languages, out of which we export a subset to context-free grammars in Backus–Naur form with the help of the built-in export tool. The number of production rules varies between languages: The Latin grammar has, for example, 1.6 million productions, whereas the Italian grammar has 5.8 million. (These differences are due to the level of coverage chosen by the grammars' designer, and do not reflect inherent complexities of the languages.)

`PolyNonDet` A family of grammars of increasing size indexed by i. The rules for member i follow (the final state is q_i).

- $a \xrightarrow{0} q_j$ for $j = 0, \ldots, i$
- $f[q_j, q_j] \xrightarrow{1} q_j$ for $j = 0, \ldots, i$
- $f[q_j, q_{j-1}] \xrightarrow{1} q_{j-1}$ for $j = 1, \ldots, i$

There are then $\Theta(n^i)$ runs for a tree of size n (and the number of rules grows linearly). Therefore, we call these *polynomially nondeterministic*.

`ExpNonDet` A family of grammars of increasing size, also indexed by i. The final state is q_f for all members. The rules for member i of the family for $j, k = 0, \ldots, i$ and $j \neq k$ are:

- $a \xrightarrow{0} q_j$

- $f[q_j, q_k] \xrightarrow{1} q_j$
- $f[q_k, q_j] \xrightarrow{1} q_j$
- $q_j \xrightarrow{0} q_f$

Thus, the number of rules grows quadratically in i and the number of runs for a tree of size n is $\Omega((i+1)^n)$, and we say that these are *exponentially nondeterministic*.

The variables in all result-displaying plots in this paper are either N (the number of trees or runs to output), m (the number of hyperedges or rules in the input file) or both. For the artificially created corpora, `PolyNonDet` and `ExpNonDet`, we exchange m for the variable i with which they are indexed. Note that for the former, m and i are exchangeable, but for the latter, m grows quadratically with increasing i.

4.2 Comparison with BEST TREES V.1

First, we verify that the algorithm BEST TREES is at least as efficient as its predecessor BEST TREES V.1. Therefore, we run experiments on the `ExpNonDet` data set for both of the algorithms – both solving the best trees task. The `ExpNonDet` data set was chosen since it has the largest degree of nondeterminism achievable, which should challenge both of the algorithms maximally. The results are presented in Figures 2 and 3 for varying N and i, respectively. Note that both of these plots have logarithmic y axes. We see that BEST TREES outperforms BEST TREES V.1 for both increasing N and increasing i.

More interestingly, in Figure 2, BEST TREES display a step-like behavior at $N \approx 100$, 200 and 600. The nature of the `ExpNonDet` corpus is the explanation of this: when we have found all of the distinct trees of size s (all of the same weight), then we must discard all of the duplicates of size s that are still in the queue, before arriving at a tree of size $s+1$ with higher weight.

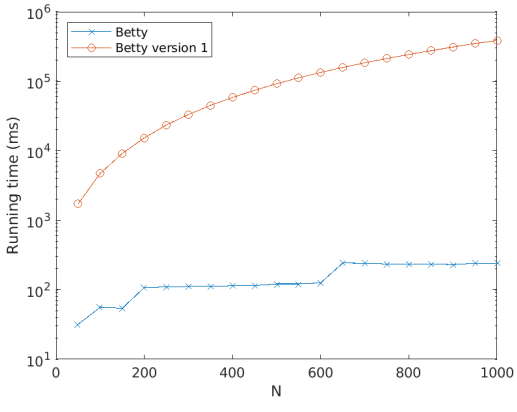


Figure 2
Comparison on the `ExpNonDet` file with $i = 7$.

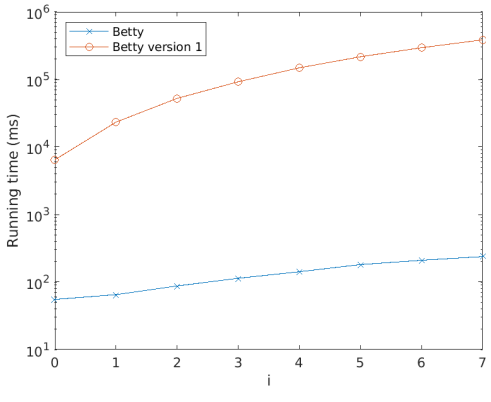


Figure 3
Comparison for $N = 1000$ on the `ExpNonDet` corpus.

4.3 Comparison with TIBURON

Now we compare BETTY with TIBURON for all data sets, starting with the `MT-data` corpus. Figure 4 shows the running times for finding the N best runs using TIBURON for every file of the `MT-data` corpus and $N = 1\,000, 2\,000, \ldots, 20\,000$. The corresponding results for BETTY can be seen in Figure 5. When we instead let BETTY solve the best trees problem for the same

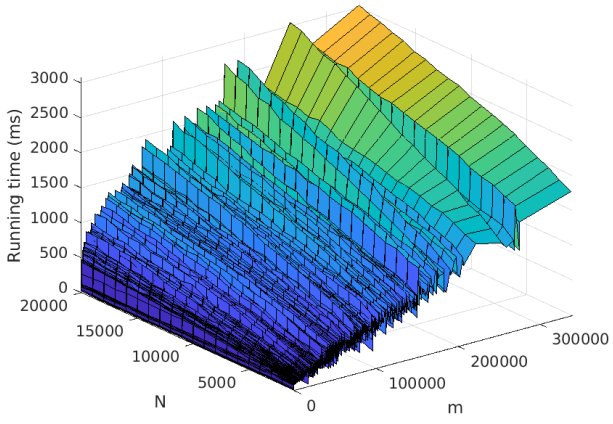


Figure 4
TIBURON solves the best runs problem for the `MT-data` corpus.

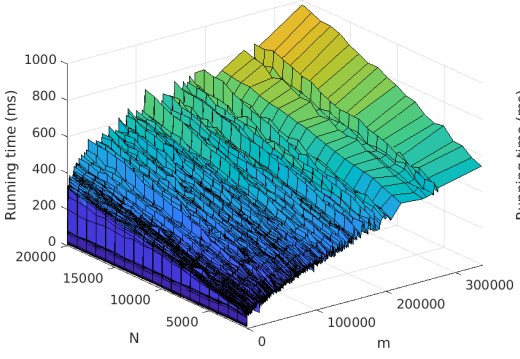


Figure 5
BETTY solves the best runs problem for the `MT-data` corpus.

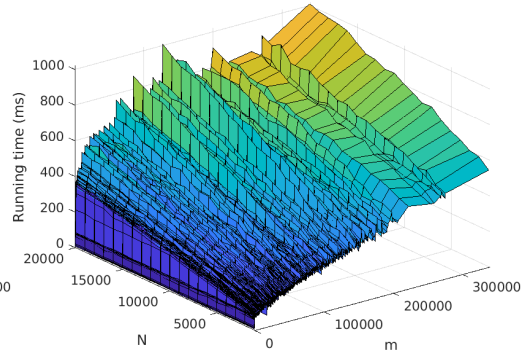


Figure 6
BETTY solves the best trees problem for the `MT-data` corpus.

input, we achieve the result in Figure 6. It seems that TIBURON is slower than BETTY for both tasks, but to be certain, we perform a statistical test. Since we have paired results from two populations, we run a Wilcoxon signed rank test on the results. We prepare both result sets for statistical testing by splitting them into 9 vectors of size 103 for each value of N. Let v_{BETTY} and v_{TIBURON} be such result vectors from the test runs of BETTY and TIBURON, respectively. Our null hypothesis H_0 is that $v_{\text{DIFF}} := v_{\text{BETTY}} - v_{\text{TIBURON}}$ comes from a distribution with median 0, and our alternative hypothesis H_1 is that v_{DIFF} comes from a distribution with median less than 0. Table 1 shows the result of a Wilcoxon signed rank test on our data. We only report on the outcome of the test comparing BETTY on the best trees task with TIBURON, since the best runs task yields nearly identical results. Each table value corresponds to one test for two paired vectors v_{BETTY} and v_{TIBURON}. For our statistical test, we use a significance level of 0.01 and can therefore conclude that H_0 is rejected for all partitions of our data (note that each table value is of order 10^{-18}). This implies that BETTY is statistically significantly faster than TIBURON on the `MT-data` corpus for both tasks. For the rest of the experiments, we omit the statistical tests.

Table 1
Probabilities of, under the null hypothesis H_0 that there is no difference between BETTY and TIBURON, observing values as or more extreme than the data, with the alternative hypothesis H_1 that BETTY on the best trees task is faster than TIBURON on the best runs task for `MT-data`.

N	Increasing file sizes, groups of 103 files $[\cdot 10^{-18}]$								
1 000	0.6514	0.6327	0.9530	0.7767	0.6514	0.6326	0.6327	0.6327	0.6326
2 000	0.6326	0.6326	0.6326	0.7218	0.6326	0.6327	0.6327	0.6327	0.6326
3 000	0.6326	0.6326	0.6327	0.6708	0.6327	0.6326	0.6326	0.6326	0.6327
4 000	0.6326	0.6327	0.6327	0.6709	0.6326	0.6326	0.6326	0.6327	0.6327
5 000	0.6327	0.6326	0.6325	0.6709	0.6326	0.6326	0.6327	0.6327	0.6327
6 000	0.6326	0.6326	0.6326	0.6708	0.6327	0.6327	0.6327	0.6326	0.6327
7 000	0.6327	0.6324	0.6326	0.6709	0.6326	0.6326	0.6326	0.6326	0.6327
8 000	0.6327	0.6327	0.6327	0.6709	0.6326	0.6326	0.6327	0.6326	0.6327
9 000	0.6327	0.6327	0.6326	0.6709	0.6327	0.6327	0.6326	0.6326	0.6327
10 000	0.6327	0.6327	0.6326	0.6709	0.6326	0.6326	0.6327	0.6326	0.6326
11 000	0.6327	0.6327	0.6327	0.6709	0.6327	0.6327	0.6326	0.6327	0.6327
12 000	0.6326	0.6327	0.6327	0.6707	0.6326	0.6327	0.6327	0.6326	0.6327
13 000	0.6327	0.6326	0.6327	0.6709	0.6326	0.6326	0.6327	0.6326	0.6327
14 000	0.6327	0.6327	0.6327	0.6709	0.6326	0.6326	0.6327	0.6327	0.6327
15 000	0.6327	0.6327	0.6327	0.6709	0.6326	0.6325	0.6327	0.6327	0.6327
16 000	0.6326	0.6326	0.6326	0.6708	0.6326	0.6327	0.6327	0.6326	0.6327
17 000	0.6327	0.6327	0.6326	0.6709	0.6326	0.6327	0.6515	0.6326	0.6327
18 000	0.6327	0.6327	0.6327	0.6709	0.6515	0.6326	0.6327	0.6326	0.6327
19 000	0.6326	0.6327	0.6327	0.6709	0.6327	0.6327	0.6327	0.6327	0.6327
20 000	0.6327	0.6327	0.6327	0.6515	0.6327	0.6327	0.6327	0.6327	0.6327

Next, we apply the three algorithms to the `GF-data` corpus. We run tests for 42 different languages, and present a selection of typical and atypical results. The typical results are shown in Figures 7, 8 and 9, and concern the languages Persian, Thai, and Somali. In all of these plots, TIBURON displays a slower performance than BETTY, regardless of what task is solved. The single atypical result is found in Figure 10, which is the result from the Latin file. Here we see that TIBURON is faster than BEST TREES for all values of N tested, although the slope of the TIBURON curve indicates that it will eventually surpass both BETTY curves with increasing N. We conjecture that this depends on certain characteristics of the Latin corpus: The Latin and the Somali file are approximately the same size, but the Latin output trees are very small and many represent one-word utterances, whereas the Somali output trees are larger and represent more complex sentence structures. The combination of a large file and small output trees makes the computation of the best contexts a large overhead, which is also visible in the plot. Even though this is a combination that we rarely see in practice, it is an interesting example of when computing the best contexts could be considered superfluous.

Now we have arrived at the artificial corpora which were created to investigate the effect of nondeterminism on the algorithms. Running TIBURON on the `PolyNonDet` corpus yields the running times in Figure 11, and using BETTY to solve the best runs and the best trees problems results in the numbers in Figures 12 and 13, respectively. We observe that TIBURON's running times are significantly larger than those of BETTY. None of the algorithms seems to be affected by increasing i, which means that they can all handle increasing degrees of polynomial nondeterminism well. By inspection of Figure 14 that compares the algorithms with respect to N, it is possible to conclude that they are all in $O(N)$ on this particular example.

When we instead use the `ExpNonDet` corpus to expose the algorithms to exponential nondeterminism, the best runs algorithms still do the job well, with BETTY (Figure 16) being

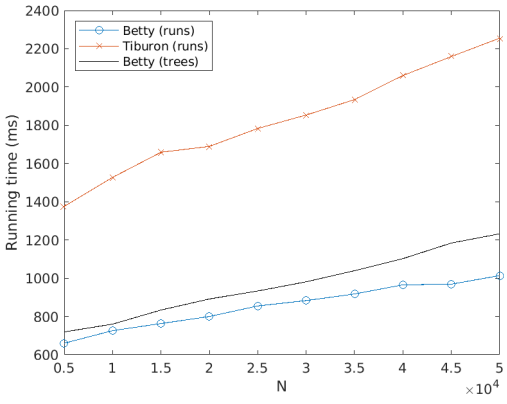


Figure 7
Comparison on the Persian file from `GF-data`.

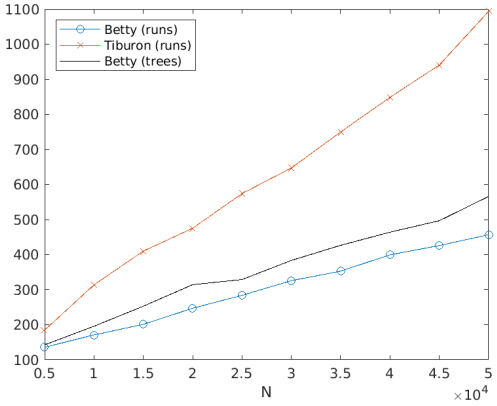


Figure 8
Comparison on the Thai file from `GF-data`.

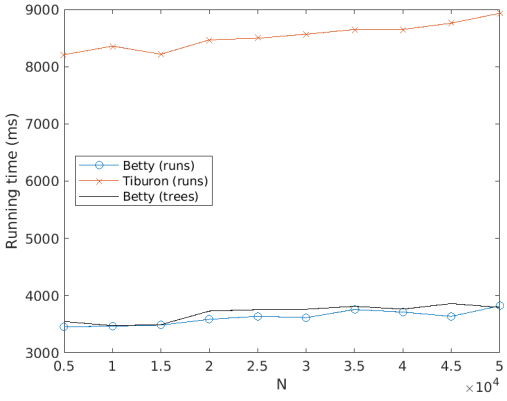


Figure 9
Comparison on the Somali file from `GF-data`.

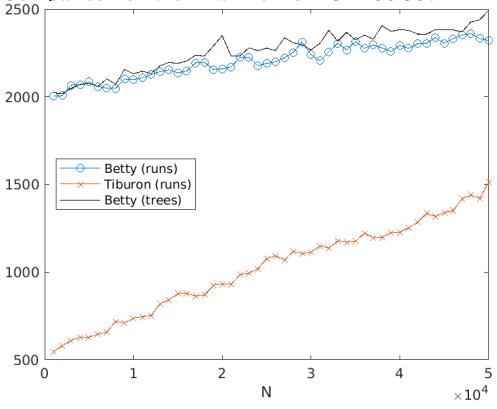


Figure 10
Comparison on the Latin file from `GF-data`.

slightly faster than TIBURON (Figure 15). However, the best trees problem is no longer that easily solvable. As can be seen in Figure 17, we had to restrict the intervals for both i and N to make the task feasible. This outcome is expected since there is now an exponential number of runs on each tree, and these duplicates have to be discarded before a larger tree can be found: The plateaus in Figure 17 indicate where we go from including trees of sizes smaller than s to including trees of size s. Asymptotic testing verifies that the best trees curve grows quadratically with i (meaning that it is linear in m) and is not worse than $N \log N$ for the second dimension. Figures 18 and 19 make a closer comparison of TIBURON and BETTY for the best runs task with respect to the variables N and i, respectively. The effect of increasing N is linear as for the `PolyNonDet` data, but when considering i, the previously constant behavior is now linear.

5. Conclusion

We have complemented previous theoretical work on the BEST TREES algorithm (Björklund, Drewes, and Jonsson 2019) with an experimental evaluation. Since BEST TREES can be easily modified to produce the best runs instead of the best trees, we considered both tasks in our evaluation. To achieve a large coverage, we used two types of data: data from real-world language processing tasks and artificially created data. The real-world data consists of machine translation

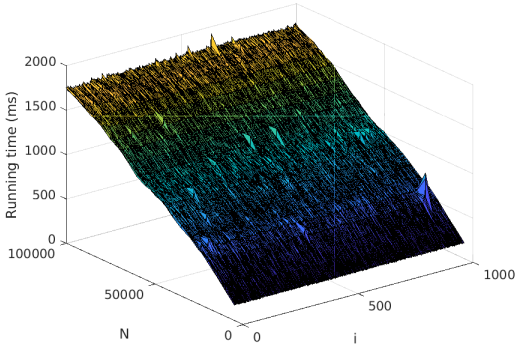


Figure 11
TIBURON solves the best runs problem for the `PolyNonDet` corpus.

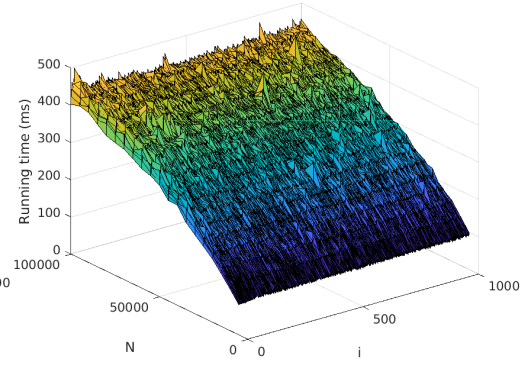


Figure 12
BETTY solves the best runs problem for the `PolyNonDet` corpus.

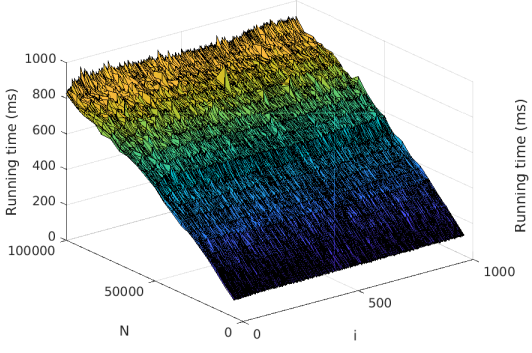


Figure 13
BETTY solves the best trees problem for the `PolyNonDet` corpus.

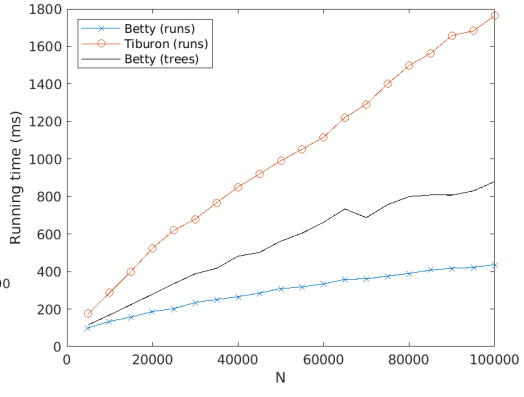


Figure 14
Comparison on the `PolyNonDet` file with $i = 999$.

output and corpus-based rule sets for natural languages; these corpora are meant to display the kind of behavior one can expect when applying our algorithm to language processing tasks. The artificial corpora were designed to expose BEST TREES to its worst-case scenario for the best trees task: the case when we have an exponential number of duplicate trees in a best runs list. We also covered the more moderate case where the nondeterminism only gives rise to a polynomial number of duplicates.

In the experiments, we first used the exponentially nondeterministic data to show that BEST TREES is better at the best trees task than its predecessor BEST TREES V.1 that uses a less efficient pruning scheme. Then, we compared BEST TREES with the state-of-the-art best-runs algorithm of Huang and Chiang (2005), implemented in TIBURON by May and Knight (2006) for both tasks on all data sets. The results made it clear that BEST TREES is preferable when extracting N best lists of both runs and trees: Betty outperforms Tiburon for the best runs task on all 2 269 input wtas except one. The single exception revealed a corner case where the Huang and Chiang algorithm is faster, namely when there are millions of rules and only a small percentage of them are used to produce N very small (height 0 or 1) runs.

Prior to the experiments, we compared the two algorithms at a conceptual level and discussed the expected effects on their running time. The allocation of queues to hyperedges instead of

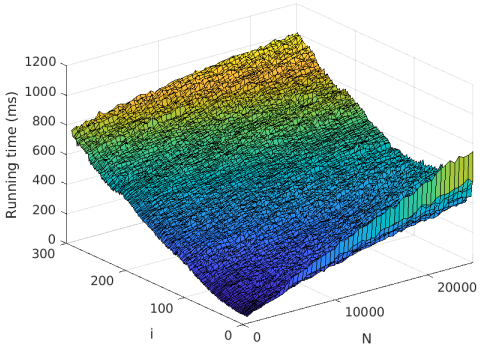


Figure 15
TIBURON solves the best runs problem for the `ExpNonDet` corpus.

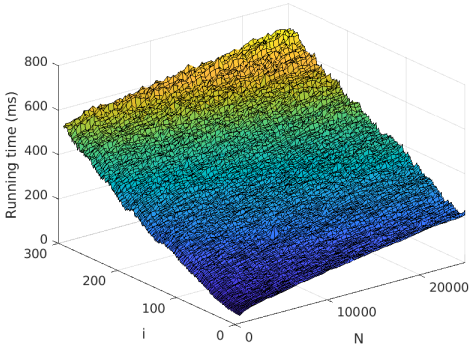


Figure 16
BETTY solves the best runs problem for the `ExpNonDet` corpus.

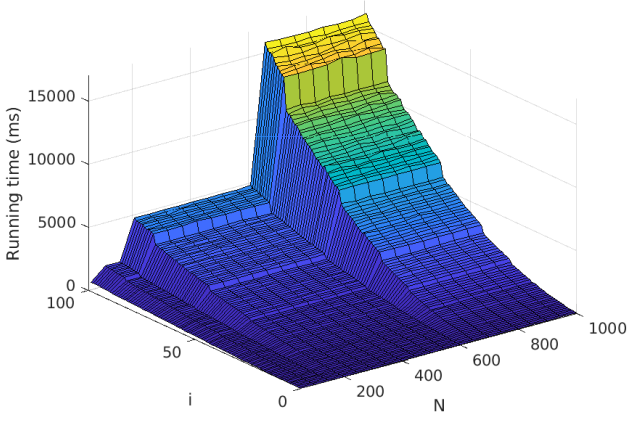


Figure 17
BETTY solves the best trees problem for the `ExpNonDet` corpus.

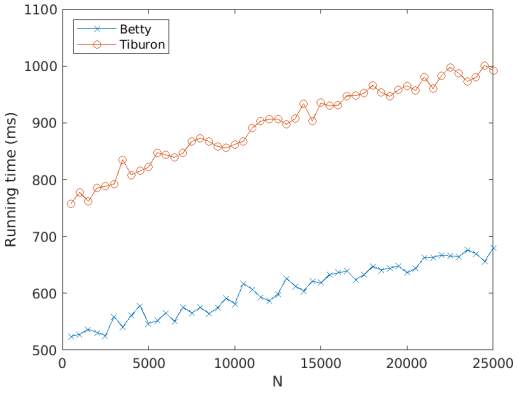


Figure 18
Comparison between TIBURON and BETTY for the best runs problem on the `ExpNonDet` corpus for $i = 299$.

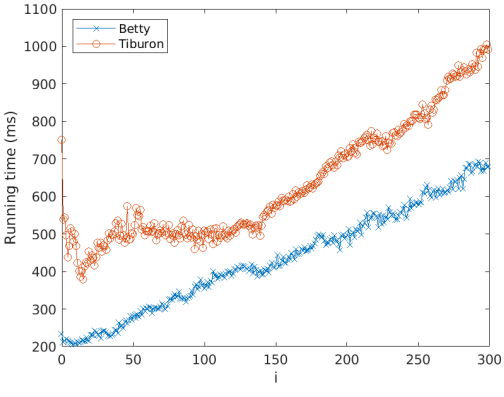


Figure 19
Comparison between TIBURON and BETTY for the best runs problem on the `ExpNonDet` corpus for $N = 25000$.

nodes mainly serves to structure the implementation. As we saw, it does not have a disadvantage with respect to running time. The use of best contexts, generalizing the idea of Mohri and Riley (2002) to trees, has a positive effect: it ensures that the maximum distance between a node and a final state is an upper bound on the maximum number of main loop iterations before the next run is outputted. This is because the best contexts guide the algorithm to take the shortest route in constructing the next best run that reaches a final state.[9]

The disadvantage of using best contexts is that it limits what semirings can be used. The technique is compatible with the tropical semiring and, as discussed in Section 2, with the equivalent Viterbi semiring, but seemingly not with semirings that are not extremal. It is currently unclear to us whether an appropriate extension is possible, and we leave this question for future work. However, such extensions seem only relevant for the N-best runs problem, because it appears highly unlikely that the N-best trees problem could ever be solved with reasonable efficiency in cases where the weight semiring is not extremal. The reason is that, in that case, the best run on a tree does not determine the weight of that tree. However, since the tropical semiring and the Viterbi semiring are the most prominent ones used to rank hypotheses in NLP, the computational advantages seen in this paper seem to justify the restriction to these semirings, even when looking only at the N-best runs problem.

Acknowledgements

We are thankful to Andreas Maletti for providing the machine translation data set used in our experiments; to Jonathan May for his support in the application of TIBURON to the N-best problem; to André Berg for sharing his expertise in mathematical statistics; and to Aarne Ranta, Peter Ljunglöf, and Krasimir Angelov for introducing us to Grammatical Framework.

References

Björklund, Henrik, Frank Drewes, and Petter Ericson. 2016. Between a rock and a hard place — parsing for hyperedge replacement DAG grammars. In *10th International Conference on Language and Automata Theory and Applications*.

Björklund, Johanna, Frank Drewes, and Anna Jonsson. 2018. A comparison of two n-best extraction methods for weighted tree automata. In *23rd International Conference on the Implementation and Application of Automata (CIAA 2018)*, Lecture Notes in Computer Science, pages 197–108.

Björklund, Johanna, Frank Drewes, and Anna Jonsson. 2019. Faster computation of N-best lists for weighted tree automata. Under review.

Björklund, Johanna, Frank Drewes, and Niklas Zechner. 2019. Efficient enumeration of weighted tree languages over the tropical semiring. *Journal of Computer and System Sciences*, 104:119–130.

Büchse, Matthias, Daniel Geisler, Torsten Stüber, and Heiko Vogler. 2010. n-best parsing revisited. In *Proceedings of the 2010 Workshop on Applications of Tree Automata in Natural Language Processing*, pages 46–54, Association for Computational Linguistics, Uppsala, Sweden.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

Finkel, Jenny Rose, Christopher D. Manning, and Andrew Y. Ng. 2006. Solving the problem of cascading errors: Approximate Bayesian inference for linguistic annotation pipelines. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 618–626, Association for Computational Linguistics, Sydney, Australia.

Huang, Liang and David Chiang. 2005. Better k-best parsing. In *Proc. of the Conference on Parsing Technology 2005*, pages 53–64, Association for Computational Linguistics.

Jurafsky, Dan and James H. Martin. 2009. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*. Pearson Prentice Hall.

9 We have conducted a small experiment not mentioned in the previous section, by switching off that feature. It showed that the use of best contexts (in that particular, randomly chosen case) reduced the size of rule queues by a factor of 10.

Knight, Kevin and Jonathan Graehl. 2005. An overview of probabilistic tree transducers for natural language processing. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 1–24, Springer.
Knuth, Donald E. 1974. Computer Programming as an Art. *Commun. ACM*, 17(12):667–673.
May, Jonathan and Kevin Knight. 2006. Tiburon: A weighted tree automata toolkit. In *International Conference on Impl. and Application of Automata*, pages 102–113, Springer.
Mohri, Mehryar and Michael Riley. 2002. An efficient algorithm for the n-best-strings problem. In *Proceedings of the Conference on Spoken Language Processing (ICSLP 02)*, pages 1313–1316.
Quernheim, Daniel. 2017. *Bimorphism Machine Translation*. Ph.D. thesis.
Ranta, Aarne. 2011. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford.
Steedman, Mark. 1987. Combinatory grammars and parasitic gaps. *Natural Language and Linguistic Theory*, 5:403–439.

V

Contextual Hyperedge Replacement Grammars for Abstract Meaning Representations

Frank Drewes
Umeå University
drewes@cs.umu.se

Anna Jonsson
Umeå University
aj@cs.umu.se

Abstract

We show how contextual hyperedge replacement grammars can be used to generate abstract meaning representations (AMRs), and argue that they are more suitable for this purpose than hyperedge replacement grammars. Contextual hyperedge replacement turns out to have two advantages over plain hyperedge replacement: it can completely cover the language of all AMRs over a given domain of concepts, and at the same time its grammars become both smaller and simpler.

1 Introduction

Natural language processing applications that receive sentences as input mainly make use of lexical and syntactic properties of the input sentences. Even though these properties are an important basis for the analysis of a sentence, one is usually more interested in the meaning of a sentence, i.e., its semantics. This is particularly true in the case of machine translation where a semantic error can cause far more bewilderment than a syntactic one.

Thus, a general-purpose formalism for modelling the semantics of sentences in a way that allows for efficient analysis would be widely useful in natural language processing. This study focuses on the generation of a semantic representation that was proposed some years ago, the abstract meaning representation (AMR) (Langkilde and Knight, 1998; Banarescu et al., 2013). An AMR[1] is a directed, rooted, acyclic, node- and edge-labelled graph that represents the semantics of an English sentence[2]; the nodes and edges represent concepts and their relations, respectively. A corpus of AMRs over a limited domain can be found in (Braune et al., 2014). As in the case of syntax trees, where tree grammars and tree automata (Knight and Graehl, 2005) provide a model for distinguishing structurally correct trees from incorrect ones, the algorithmic processing of AMRs would benefit from the existence of appropriate formal models for their generation or recognition. Here, we focus on the generation of AMRs by graph grammars, which have previously been proposed as formal models for this very task (Chiang et al., 2013).

The usefulness of two types of hyperedge replacement grammar (HRG, see Habel (1992); Drewes et al. (1997)) for AMR generation was investigated by Jonsson (2016a) (see also (Jonsson, 2016b)), namely the predictive top-down (PTD) parsable grammar (Drewes et al., 2015) and the restricted directed acyclic graph (rDAG) grammar (Björklund et al., 2016). Both are of particular interest because their study was, among other possible application areas, motivated by AMR generation. A specific advantage of these special cases of HRGs is that their membership problem is solvable in polynomial time. However, Jonsson (2016a) concludes that neither of them is able to generate the complete set of AMRs over a given concept domain.

Unrestricted HRGs allow for better coverage at the expense of greater computational complexity. However, a general disadvantage of hyperedge replacement remains. The nonterminal items in an HRG are hyperedges – edges that may be attached to more (or fewer) than two nodes. Replacement of a hyperedge inserts a new subgraph in its place,

[1] We use the term AMR to refer not only to the concept of Abstract Meaning Representation as such (Langkilde and Knight, 1998; Banarescu et al., 2013), but also to its individual graphs.

[2] Although AMR is to some extent language independent, it is biased towards English (Banarescu et al., 2013), and therefore not truly an interlingua (Xue et al., 2014).

Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+13), pages 102–111,
Umeå, Sweden, September 4–6, 2017. © 2017 Association for Computational Linguistics

connecting it to the host graph via the nodes the replaced hyperedge was incident on. Intuitively, nonterminal hyperedges keep track of a number of potentially relevant nodes for the purpose of being able to attach new edges to them later on in the derivation. This process is well known (and easily seen) to generate graph languages of bounded treewidth. As shall be illustrated in Section 6 the ability of hyperedges to keep track of a bounded number of previously generated nodes can be used to ensure structural properties such as those caused by control verbs. However, it appears that other types of reentrancies, like those arising from the use of pronouns, are of a different nature. If, for example, several instances of the concept `boy` have been generated, any of them can in principle be referred to from anywhere else in the AMR. As a consequence, there is no reasonable a priori bound on the treewidth of the graph. Nonterminal hyperedges generating other parts of the AMR would have to keep track of all `boy` instances to accomplish full coverage. On the one hand, this is not possible in an HRG. On the other hand, it does not seem to be desirable either, because keeping track of every `boy` instance individually would enable a level of control far beyond what is needed.

Here we consider contextual hyperedge replacement grammars (CHRGs) (Drewes et al., 2012; Drewes and Hoffmann, 2015) to learn whether they can be used to overcome these disadvantages. CHRGs are also based on hyperedge replacement, but the left-hand side of a rule can contain so-called contextual nodes. This provides access to nodes other than those immediately controlled by the nonterminal hyperedge, thus enabling rules to establish connections of the type discussed in the previous paragraph. The additional ability is severely limited, far below true context-sensitivity in power, because nodes are terminal items and derivation steps cannot distinguish between contextual nodes with the same label. For instance, in the situation sketched above a rule application would just pick *any* occurrence of `boy` elsewhere in the host graph. As a consequence, however, the treewidth of generated graphs is not necessarily bounded anymore.

In the present paper we study and illustrate the advantages of CHRGs over HRGs for AMR generation by looking at an example concept domain in a theoretical case study. To this end, we build a CHRG that generates AMRs over a restricted domain and argue that it exhibits perfect coverage. The baseline domain is the one introduced by Braune et al. (2014), consisting of the concepts `boy`, `girl`, `want` and `believe` along with two basic relations (called `arg0` and `arg1`) that are used to bind the concepts together and correspond to the agent and patient of a `want` or `believe` event. We also consider the construction of CHRGs for more general AMRs to explore the advantages of the more generous rule format. Therefore we add a small set of possible modifiers, allow an arbitrary number of boys and girls to appear in an AMR,[3] and discuss how to handle control verbs.

The conclusion of our study is that contextual hyperedge replacement is indeed a promising formalism for describing sets of AMRs. On the one hand, AMRs contain the mentioned local structures that must satisfy certain well-formedness constraints, such as in the case of control verbs. This can be implemented like it would in an HRG, using a nonterminal hyperedge to keep track of the involved nodes. On the other hand, contextual nodes can be used to implement the kinds of coreferences which may occur anywhere without following strict local rules, such as those relating to the use of pronouns. As discussed above, the latter creates problems in HRGs because nonterminal hyperedges would have to keep track of potential antecedents, which seems inappropriate for various reasons: it is restricted by the rank of hyperedges, provides an unnecessarily detailed level of control (thus creating the risk of overfitting), and leads to a huge number of rules to account explicitly for all the possible nondeterministic choices arising from the (exponentially) many ways in which coreferences can be inserted.

The obvious downside of using CHRGs is that computational problems may potentially become more difficult. However, recent results on shift-reduce parsing for both HRGs and CHRGs (Drewes et al., 2017)[4] indicate that this may not be the case. In fact, as the rank of hyperedges and the number of rules are central parameters in the complexity of membership algorithms for both unrestricted HRGs and CHRGs, it may even pay off to turn to CHRGs since this leads to smaller ranks and much fewer rules, the latter

[3] Braune et al. (2014) only consider at most one boy and at most one girl.

[4] See `https://www.unibw.de/inf2/grappa/` for the extension to CHRGs.

because the use of contextual nodes removes the necessity to implement nondeterministic choices explicitly by creating a separate rule for each.

In Section 2, we lay the ground for the rest of the paper with some basic definitions. The CHRG is defined in Section 3, and the subset of AMR to be considered here is discussed in Section 4. The construction of a CHRG for this domain is described in Section 5. In Section 6, we indicate how to generalise it to larger domains, and in particular how control verbs can be added. Finally, the results are discussed in Section 7 followed by the conclusions and future work in Section 8.

Acknowledgement We thank the reviewers for useful comments that helped us clarify the line of argumentation (as we hope).

2 Preliminaries

For a set A, we write A^* to denote the set of finite sequences or strings over A, and $A^{®}$ for the set of strings over A in which no element is repeated; ε denotes the empty sequence. Elements of A are identified with strings of length 1 over A, and thus subsets of A are string languages at the same time.

Furthermore, we let 2^A denote the power set of A, i.e., the set of all subsets of A. The extension of a function $f\colon A \to A'$ to sequences $a_1, \ldots, a_n$ where $a_i \in A$ for $0 \leq i \leq n$ is denoted $f^*\colon A^* \to A'^*$ and defined by $f^*(a_0, \ldots, a_n) = f(a_1) \cdots f(a_n)$. Concatenation of strings is denoted by simple juxtaposition, and elementwise concatenation of two string languages L, L' is denoted by $L \cdot L'$, i.e., $L \cdot L' = \{uv \mid u \in L, v \in L'\}$.

A *labelling alphabet* is a set Σ partitioned into three mutually disjoint sets Σ_V, Σ_E and Σ_N on which an *arity* function $arity\colon \Sigma_E \uplus \Sigma_N \to 2^{\Sigma_V^*}$ is defined. (See Section 5 for an example of an alphabet and its arity function.) The sets Σ_V, Σ_E and Σ_N are referred to as *node labels*, *(hyper)edge labels* and *nonterminal labels*, respectively.

A *hypergraph* is a generalisation of directed graphs by the usage of edges that can connect an arbitrary number of nodes. Here, we consider node- and edge-labelled hypergraphs.

Definition 1 (Hypergraph (Drewes et al., 2012))**.** *A* labelled hypergraph (hypergraph, *for short) over a labelling alphabet* Σ *is a tuple* $G = (V, E, att, label_V, label_E)$ *such that*

- V *is a finite set of* nodes.
- E *is a finite set of* hyperedges.
- $att : E \to V^{®}$ *is the* attachment of hyperedges.
- $label_V : V \to \Sigma_V$ *is the* labelling of nodes.
- $label_E : E \to \Sigma_E \cup \Sigma_N$ *with* $label_V^*(att(e)) \in arity(label_E(e))$ *for all* $e \in E$ *is the* labelling of hyperedges.[5]

The *rank* of a hyperedge e is $|arity(label_E(e))|$. Hyperedges with labels in Σ_N are called *nonterminals*; $\mathcal{G}_\Sigma$ denotes the set of all hypergraphs over Σ. For a hypergraph G and a hyperedge $e \in E$, the hypergraph resulting from removing e from G is denoted by $G - e$. The empty hypergraph is denoted by $()$.

In illustrations, nodes and hyperedges are drawn as ellipses and squares, respectively, with inscribed labels. The attachment of a hyperedge is shown by lines, and the attachment order is depicted using numbers (these can be left out if the attachment order is clear from the context or irrelevant). If a hyperedge connects exactly two nodes (i.e., it is binary), it can be drawn as an arrow directed from the first node of the attachment to the second with its label next to it. See Section 5 for various examples of hypergraphs. Note that a hypergraph containing only binary hyperedges is equivalent to an ordinary directed graph; this is the case in e.g. Figure 1.

3 Contextual Hyperedge Replacement

Given a hypergraph containing nonterminals, rules can be applied to it in order to generate a new hypergraph. A set of such rules along with a fixed hypergraph to which they are to be applied forms a grammar. The grammar type considered here was proposed in (Drewes et al., 2012; Drewes and Hoffmann, 2015) and uses the following rule type.

Definition 2 (Contextual Rule)**.** *A* contextual hyperedge replacement rule *(or* contextual rule*) is a pair* (L, R) *where* L *and* R *are hypergraphs over the labelling alphabet* Σ *such that*

- L *(the* left-hand side*) contains exactly one hyperedge* e *that must be a nonterminal, and*
- R *(the* right-hand side*) is an arbitrary supergraph of* $L - e$*.*

A contextual rule for which all nodes in the left-hand side are connected to e is called *context-free*. The nodes that are not connected to e are referred to as *contextual nodes*.

[5] The arity function used differs from the one in (Drewes et al., 2012), but the resulting hypergraph definition remains the same.

We denote a contextual rule by letting ::= separate the left- and right-hand sides. Moreover, we allow rules that share the same left-hand side to be drawn more compactly; in this case, the left-hand side is only drawn once, and a vertical line is used to separate the right-hand sides from each other. To save further space, we use rule schemata in which labels may be variables ranging over a specified subset of the set of all labels. As an example of a set of contextual rules, consider the rules in (iii) in Figure 4 of Section 5. Every choice of z, u, v and a_1, a_2 in the range specified beneath the rules yields three rules. Each has the nonterminal N_1 in its left-hand side, and the node labelled u is a contextual node. In addition, the third right-hand side contains a newly generated node labelled v.

A contextual rule (L, R) can be applied to a hypergraph G containing an isomorphic copy of L, i.e., a subgraph that is equal to L up to renaming of nodes and hyperedges. Suppose for simplicity that L is a subgraph of G. Then the application of the rule works in the following manner:

1. Remove e from G, yielding $G - e$.
2. Add R to $G - e$, disjointly.
3. Identify the nodes in $L - e$ with the corresponding nodes in R.

The resulting hypergraph is denoted by $G[R/e]$. Now, we can formally define the grammar type that makes use of contextual rules.

Definition 3 (Contextual Hyperedge Replacement)**.** *A* contextual hyperedge replacement grammar (CHRG) *is a triple* $\Gamma = (\Sigma, \mathcal{R}, Z)$ *where*

- Σ *is a finite labelling alphabet,*
- $\mathcal{R}$ *is a finite set of contextual rules, and*
- $Z \in \mathcal{G}_\Sigma$ *is a start hypergraph.*

If $G' = G[R/e]$ for some contextual rule $(L, R) \in \mathcal{R}$, we say that G' is *derived* from G in Γ, and we write $G \Rightarrow_\mathcal{R} G'$. The language generated by Γ is $\mathcal{L}(\Gamma) = \{G \in \mathcal{G}_{\Sigma \setminus N} \mid Z \Rightarrow^*_\mathcal{R} G\}$ where $\Rightarrow^*$ is the reflective and transitive closure of $\Rightarrow$. Two CHRGs Γ_1 and Γ_2 are *equivalent* if $\mathcal{L}(\Gamma_1) = \mathcal{L}(\Gamma_2)$, i.e., if they generate the same language. If all of the rules of Γ are context-free, then Γ is a *hyperedge replacement grammar* (HRG). Thus, CHRG is a generalisation of HRG through the extension of context-free rules to contextual rules. Intuitively, the difference between the two is that CHRGs can nondeterministically pick a previously generated node with a specified label without that node being connected to the replaced nonterminal. HRGs do not have this ability.

The graph languages generated by CHRGs are in NP (Drewes and Hoffmann, 2015), and can thus be NP-complete, as this already holds for HRGs. Hence, unless P = NP there are CHRGs which do not admit a polynomial membership test. For HRGs, there exist polynomial membership algorithms for nontrivial special cases such as PTD parsable, shift-reduce parsable, and rDAG HRGs. The fact that membership testing is not harder for CHRG than for HRG (at least in theory) strengthens the hope that there are subclasses of CHRG with efficient membership tests. Indeed, this has partially been confirmed: the membership algorithms for PTD and shift-reduce parsable HRGs can be extended to CHRGs.[4]

4 Abstract Meaning Representation

Abstract meaning representation (AMR) (Langkilde and Knight, 1998; Banarescu et al., 2013) denotes sentence meaning as directed, rooted, acyclic graphs with node and edge labels. To the extent possible, AMR aims to provide a unique representation of semantics, i.e., while numerous sentences can express the same meaning, they should all map to the same AMR. The idea is that the nodes of the graph represent the concepts identifiable in the sentence, and the edges represent the relations between the concepts. Intuitively, the subgraph rooted at any one given node represents an event, a fact, or an entity. See Figure 1 for example AMRs that can be realised into the English sentences "The boy wants the girl to believe him" or "[...] to believe the other boy."

The previous example highlights that every event or entity represented in an AMR should occur once and only once. In fact, this is the major difference between AMRs and syntax trees, in which several subtrees may refer to the same semantic thing. In the second AMR in Figure 1, the fact that the wantee is not represented by the same node as the believee implies that these two are distinct. Representing the first sentence by the second AMR (or the second one by the first) is an error.

Thus, to achieve complete coverage, a grammar for generating AMRs over the given domain must generate both graphs in Figure 1. Figures 2 and 3 show another pair of AMRs, of which the former correctly represents the semantics of the sentence "The boy wants the girl to believe in herself and this is what the girl wants, too." The interpretation of the latter is less obvious. We do not endeav-

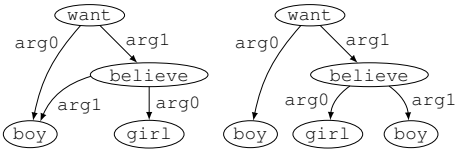


Figure 1: AMRs representing an event `want`, where the wanter is a `boy` and the wantee is the event `believe` for which the believer is a `girl` and the believee is either the formerly mentioned `boy` (left) or a different one (right).

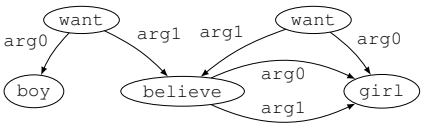


Figure 2: Another AMR.

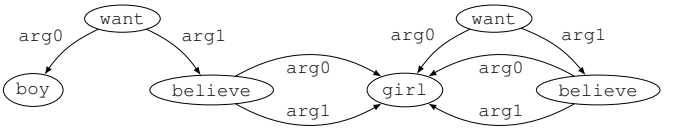


Figure 3: An AMR similar to the one in Figure 2, but with two distinct `believe` events.

our to discuss whether this AMR is meaningful at all, but it certainly seems to be less probable. Unfortunately, it turns out that structures such as the one in Figure 3 are easy to generate by a HRG and even by the aforementioned PTD parsable HRGs, whereas trying to include the more desirable one in Figure 2 meets severe difficulties. This is an instance of the problems mentioned in the introduction: a HRG generating structures like the one in Figure 2 (even one that is not PTD parsable) would have to generate the `believe` node early on and then keep track of it in its nonterminal hyperedges to establish the desired relations later on, when the two `want` nodes are generated.[6] The nondeterministic choices this creates seem to destroy PTD parsability. Further, even if PTD parsability is abandoned in favour of generative power, the desired effect can only be approximated: as the number of `believe` nodes grows, it eventually exceeds the number of nodes that the nonterminal hyperedges have been designed to keep track of.

4.1 The Boy-Girl AMR Corpus

The *boy-girl AMR corpus* is a set of 10 000 AMRs over a restricted domain that was presented in (Braune et al., 2014). Each AMR of this corpus fulfils the following conditions:

- The node label alphabet consists of the concept names `boy`, `girl`, `want`, and `believe`.
- The edge label alphabet consists of the relation names `arg0` and `arg1`.
- The node labels `boy` and `girl` occur at most once each, and label the leaves of the graph.
- For each `want` and `believe` node, the outgoing edges carry distinct labels and all incoming edges are labelled `arg1`.

The relation `arg0` is used for marking the agent of an action expressed by a concept in the form of a verb, and the patient of the same action is given by the concept pointed to by `arg1`. The above restrictions simply give us the domain and tell us that a person cannot be used as a verb, and that verbs cannot be agents, but that an event (a subgraph with a verb concept as root) can act as a patient. The left AMR in Figure 1 is a boy-girl AMR, whereas the left one is not, as it contains two copies of `boy`.

To make things more interesting, we remove the restriction that there can only be one girl and one boy, and extend the concept domain by months, weekdays and the words `happy` and `angry`. Let Σ_V denote this extended domain. Finally, we add the relations `manner`, `month` and `day`, which together with `arg0` and `arg1` form Σ_E.

5 Construction of a Boy-Girl CHRG

Let us now discuss how to construct a CHRG that generates the complete language of boy-girl AMRs. The alphabet used is that of Section 4.1, enlarged by $\Sigma_N = \{S, N, N_1, M\}$ and with the

[6] This assumes for simplicity that a bottom-up generation strategy is employed. However, the difficulties arising depend only marginally on the choice of strategy.

arity function given as follows: for $A \in \Sigma_E$, $arity(A) = \{\texttt{want}, \texttt{believe}\} \cdot TAR_A$ where

$$\begin{aligned} TAR_{\texttt{arg0}} &= \{\texttt{boy}, \texttt{girl}\} \\ TAR_{\texttt{arg1}} &= \Sigma_V \setminus \{\texttt{happy}, \texttt{angry}\} \\ TAR_{\texttt{manner}} &= \{\texttt{happy}, \texttt{angry}\} \\ TAR_{\texttt{month}} &= \{\texttt{Jan}, \ldots, \texttt{Dec}\} \\ TAR_{\texttt{day}} &= \{\texttt{Mon}, \ldots, \texttt{Sun}\}. \end{aligned}$$

Furthermore, $arity(S) = arity(N) = \varepsilon$ and $arity(N_1) = arity(M) = \{\texttt{want}, \texttt{believe}\}$. The start hypergraph Z consists of a single nonterminal labelled S. The rules of the boy-girl CHRG can be seen in Figure 4.

The initial rules of the grammar, the ones of schema (i), simply generate the first leaf of the graph. The rules of schema (ii) choose between terminating the derivation by generating the empty graph or continuing it by generating a non-leaf node. Schemata (iii) and (iv) connect the newly generated node with label z to at least one previously generated node. Moreover, these rules connect a nonterminal labelled M to the node, which makes it possible to add zero or more outgoing `manner` edges from the node currently being handled to suitable (new) leaves. In addition, at most one `month` and one `day` edge can be generated, and the latter only in connection with the former. We note here that these restrictions are not intended to be semantically particularly meaningful. They only serve to illustrate that this type of "regular control" can be used to put together the combination of outgoing relations a node shall have.

To restart the cycle of either generating another `want` or `believe` node or terminating the derivation, (iii) and (iv) also create a new nonterminal labelled N.

We can see that each node must be given all of its outgoing `arg0` and `arg1` edges before another one is generated, making sure that the resulting AMR is acyclic (because `manner`, `month`, and `day` edges only point to leaves). Every node generated by (iii), (iv), or (v) is immediately connected to an already existing node. Moreover, the new node generated by the second rule of (ii) is connected to a nonterminal labelled N_1 until that node, by (iii) or (iv), is connected to an older node. Using this, it follows by induction that only connected graphs are generated.

An example of a derivation using the boy-girl CHRG can be seen in Figure 5. The rule(s) used in every step are indicated above the derivation symbol ($\Rightarrow$) combined with the right-hand side index of the used rule (starting at 1). What variables are mapped to which labels throughout the derivation is shown implicitly. The resulting AMR is the previously discussed one in Figure 2.

It should be clear that this grammar generates the complete language of AMRs over our small domain: as we are only interested in generating acyclic graphs this is always possible by generat-

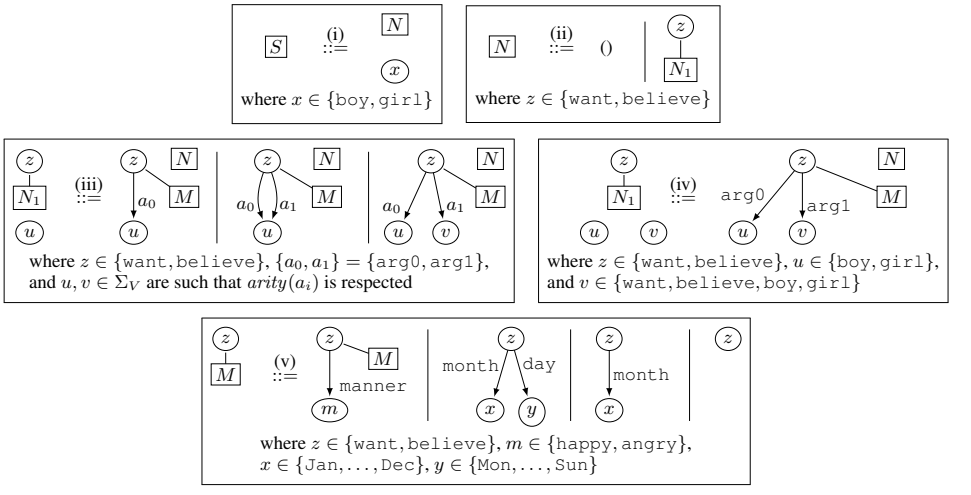


Figure 4: A boy-girl CHRG exemplifying general rule structures. Rules are named for later reference by a superscript on the operator '::='.

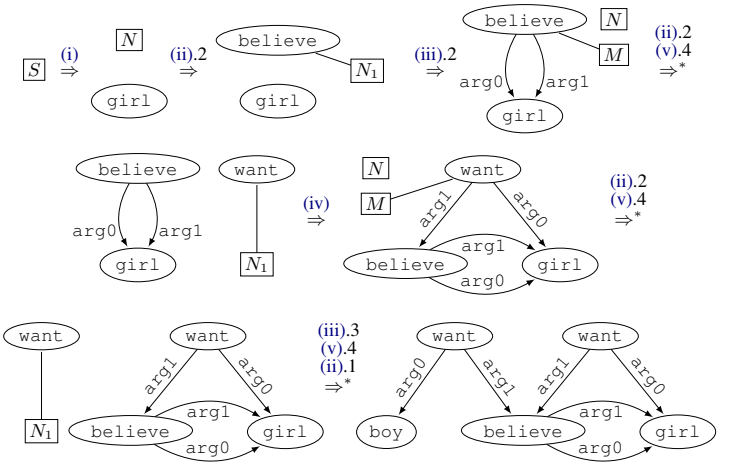


Figure 5: A derivation of an AMR using the boy-girl CHRG.

ing the nodes in reverse topological order. In other words, the CHRG constructed indeed generates the complete AMR language described above.

6 Generalisations

Let us now formulate some general rules about how to create a CHRG that generates AMRs over a given, finite domain of concepts and relations.

Let Σ_V contain the concept names of the domain and Σ_E its relations – these can be any sets as long as they are finite. Define the arity function of Σ as $\mathit{arity}(r) = \{c_i c_j \mid r \text{ is a valid relation from } c_i \text{ to } c_j \text{ for } c_i, c_j \in \Sigma_V\}$. The arity function is used to restrict which concepts can be connected using a particular relation. (For example, in the boy-girl case, we know that verbs cannot be agents and that persons cannot have agents. Thus, `want boy` and `want girl` are allowed in $\mathit{arity}(\texttt{arg0})$, but not `want believe` or `girl want`.)

As in the boy-girl CHRG, generation starts with the base case – a single leaf nondeterministically chosen from all the concept names that may appear as leaves. A nonterminal similar to N in the boy-girl case generates one new non-leaf node at a time. All of the outgoing edges to other non-leaf nodes are generated before returning to N. This guarantees acyclicity as it prevents nodes from being given outgoing edges to nodes generated posterior to them. As in the previous section, contextual rules are thus used to (1) enable the generation to refer back to previously generated nodes by adding incoming relations and to (2) make sure that the AMRs are connected. Further leaves can only be generated along with the generation of an outgoing relation from another node.

We may also want for a CHRG to generate various combinations of outgoing relations from the latest non-leaf node (in the boy-girl grammar represented as z). This can be done similarly to the generation of `manner`, `month`, and `day` edges by M in Figure 4.

In view of the previous discussion the reader may wonder whether one will ever have the need to use nonterminal labels A with $|\mathit{arity}(A)| > 1$. It might seem that arguments can always be picked using contextual nodes. However, this selects targets exclusively based on their labels and is thus inappropriate if finer structural control is required. To illustrate this, let us add the object control verb `persuade` and the subject control verb `try` to our concept set (i.e., to Σ_V). We also need a new relation `arg2` to connect an occurrence of `persuade` to its indirect object, i.e., $\mathit{arity}(\texttt{arg2}) = \{\texttt{persuade}\} \cdot \Sigma_{\text{verb}}$ where $\Sigma_{\text{verb}} = \{\texttt{want}, \texttt{believe}, \texttt{persuade}, \texttt{try}\}$.

Recall that, whenever an `arg0` edge is created in one of the rules in Figure 4, the subsequent creation of further `want` and `believe` nodes is taken care of by a nonterminal N generated at the same time. To implement control, we use variants of these rules which, instead of N, use a nonterminal C with $\mathit{arity}(C) = \Sigma_{\text{verb}} \cdot \{\texttt{boy}, \texttt{girl}\}$. This nonterminal is attached to the two nodes of

the arg0 edge, thus remembering where the control should take place instead of floating freely.

Some of the new rules are illustrated in Figure 6. The rules in (vi) work like those in (iii), but create nonterminals labelled by C instead of N, in the way just described. A similar rule obtained from (iv) is left out to save space.

The remaining rules insert the control verbs: those in (vii) implement subject control by try whereas those in (viii) implement object control by persuade. Each of the rules corresponds to a succession of two rules in Figure 4, namely (iii).1 followed by rule (ii).2. The first of each pair of rules initiates another level of control whereas the second returns to the "uncontrolled" case. Note that we, for simplicity, drop the nonterminals M that should be attached to the control verbs to follow (iii).1 strictly. Also, there should be further rules corresponding to (iii).2, (iii).3, and (iv), which are omitted because they are constructed along the same lines as those shown in the figure.

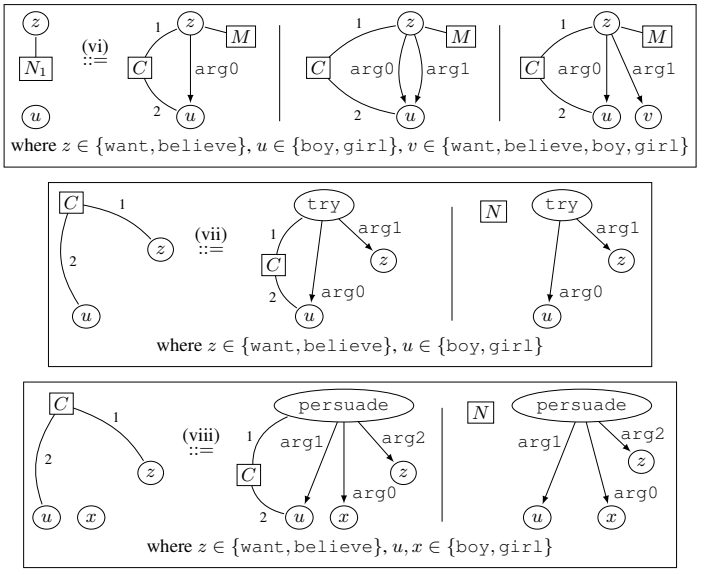


Figure 6: Rules implementing subject control (vii) and object control (viii).

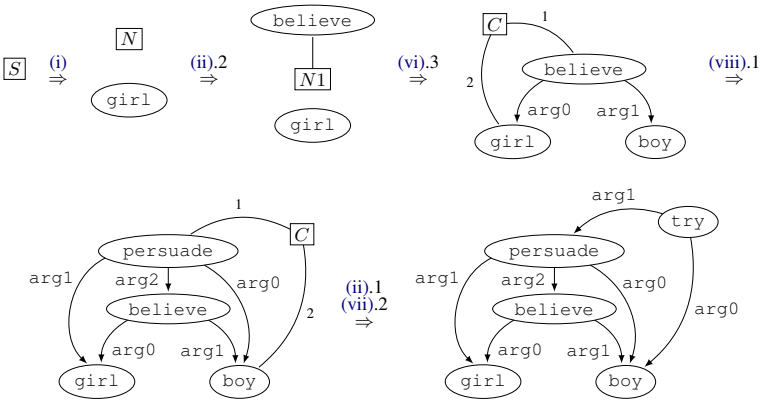


Figure 7: An example derivation using the rules in Figures 4 and 6.

Figure 7 shows an example derivation involving the new rules, the sentence being "The boy tries to persuade the girl to believe him." The reader may wish to add the remaining rules not shown in Figure 6, so that AMRs for sentences such as "The boy wants to persuade the girl to try to persuade the other girl to believe him" can be generated.

7 Discussion

Being able to generate complete AMR languages is a clear advantage of CHRGs compared to PTD parsable and rDAG grammars, and even over unrestricted HRGs. The latter can only generate graph languages of bounded treewidth, and despite the fact that real-world AMRs usually seem to be of small treewidth (Chiang et al., 2016) it does not seem to be justified to impose an a priori upper bound on their treewidth.

However, the advantage of CHRGs for modelling AMR languages exceeds the formal aspect of unlimited treewidth. In an HRG, nonterminal hyperedges have to keep track of all nodes to which edges shall (potentially) be attached later on in the process. This includes the implementation of non-local phenomena like anaphora, for which little if any structural control is required, thus resulting in an artificial increase not only in the rank of hyperedges but also in the number of rules. The latter may be significant, even exponential in the number of additional nodes to be kept track of, because in a right-hand side every nonterminal hyperedge would nondeterministically have to choose a subset of additional nodes to attach to. Even so, the number of nodes that can be kept track of is restricted by a constant depending on the rank of hyperedges. In contrast, CHRGs do not need to carry around such additional information at all as they can simply view antecedents as contextual nodes when it is time to insert a reference. The finer control provided by nonterminal hyperedges can be reserved for situations in which structural requirements must be met, such as implementing control verbs, quantifiers, and the like.

It remains to be seen whether the algorithmic properties of AMR-generating CHRGs are sufficiently good, especially when compared to HRGs. Since CHRGs generalise HRGs one may expect them to be algorithmically more demanding. However, the preceding discussion indicates that the converse may be true in practice. The efficiency of algorithms for analysing graphs with respect to a given (C)HRG depends most significantly on two things: the ranks of hyperedges and the number of rules (see in particular Chiang et al. (2013)). Thus, the greater algorithmic complexity of CHRGs may very well turn out to be outweighed by them requiring much smaller ranks and fewer rules, because the difference in size, as indicated above, will most likely be exponential in the desired number of potential antecedents.

The fact that CHRGs allow for structurally simpler rules may also make it possible to cast CHRGs like the one discussed here into a special form suitable for efficient analysis like shift-reduce parsing (Drewes et al., 2017) whereas the same may happen to be impossible for an HRG, even though the former has better coverage than the latter. Whether these possibilities can be realised is a question to be addressed by future work.

8 Conclusion

We have shown how CHRGs can generate complete AMR languages in cases where HRGs fail to do so because they do not provide appropriate means for the implementation of arbitrary coreferences. Whether CHRGs can generate complete AMR languages over arbitrary concept domains, including phenomena such as quantification while excluding structurally incorrect graphs, remains to be studied. In any case, the simplicity of the grammar discussed here seems to be promising. Future work, should investigate how efficiently problems such as the membership problem can be solved in practise for AMR-generating CHRGs. In this context, a better understanding of how quickly such CHRGs grow with the size of the input domain would also be valuable. If CHRGs indeed turn out to be a suitable device for AMR generation, a long-term goal should be to define a weighted version of CHRGs and to devise machine learning methods that make it possible to extract rule weights or even entire grammars from AMRbank.

Finally, it should be mentioned that there are other formalisms for defining languages of directed acyclic graphs that seem promising and should therefore be investigated for AMR modelling, e.g. DAG automata (Blum and Drewes, 2016, 2017; Chiang et al., 2016). In particular, it would be interesting to study the relative advantages and disadvantages of these options, and whether they can be combined in a fruitful way.

References

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*. pages 178–186.

Henrik Björklund, Frank Drewes, and Petter Ericson. 2016. Between a rock and a hard place — parsing for hyperedge replacement DAG grammars. In *10th International Conference on Language and Automata Theory and Applications*.

Johannes Blum and Frank Drewes. 2016. Properties of regular dag languages. In *10th International Conference on Language and Automata Theory and Applications*.

Johannes Blum and Frank Drewes. 2017. Language theoretic properties of regular DAG languages. To appear.

Fabienne Braune, Daniel Bauer, and Kevin Knight. 2014. Mapping between English strings and reentrant semantic graphs. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation*. pages 4493–4498.

David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. 2013. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. pages 924–932.

David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. 2016. Weighted DAG automata for semantic graphs. Submitted.

Frank Drewes and Berthold Hoffmann. 2015. Contextual hyperedge replacement. *Acta Informatica* 52(6):497–524.

Frank Drewes, Berthold Hoffmann, and Mark Minas. 2012. *Applications of Graph Transformations with Industrial Relevance: 4th International Symposium, Revised Selected and Invited Papers*, chapter Contextual Hyperedge Replacement, pages 182–197.

Frank Drewes, Berthold Hoffmann, and Mark Minas. 2015. Predictive top-down parsing for hyperedge replacement grammars. In *Proceedings of the 8th International Conference on Graph Transformation*. pages 19–34.

Frank Drewes, Berthold Hoffmann, and Mark Minas. 2017. Predictive shift-reduce parsing for hyperedge replacement grammars. In *Proc. 10th Intl. Conf. on Graph Transformation (ICGT'17)*. Lecture Notes in Computer Science.

Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. 1997. Hyperedge replacement graph grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 95–162.

Annegret Habel. 1992. *Hyperedge replacement: grammars and languages*, volume 643. Springer Science & Business Media.

Anna Jonsson. 2016a. *Generation of Abstract Meaning Representations by Hyperedge Replacement Grammars – A Case Study*. Master's thesis.

Anna Jonsson. 2016b. On the generation of abstract meaning representations using polynomial-time parsable hyperedge replacement grammars. The Sixth Swedish Language Technology Conference.

Kevin Knight and Jonathan Graehl. 2005. An overview of probabilistic tree transducers for natural language processing. In *International Conference on Intelligent Text Processing and Computational Linguistics*. pages 1–24.

Irene Langkilde and Kevin Knight. 1998. Generation that exploits corpus-based statistical knowledge. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics - Volume 1*. pages 704–710.

Nianwen Xue, Ondrej Bojar, Jan Hajic, Martha Palmer, Zdenka Uresova, and Xiuhong Zhang. 2014. Not an interlingua, but close: Comparison of English AMRs to Chinese and Czech. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation*. pages 1765–1772.

VI

Polynomial Graph Parsing with Non-Structural Reentrancies

Johanna Björklund
Dept. Computing Science
Umeå University
Umeå, Sweden
johanna@cs.umu.se

Frank Drewes
Dept. Computing Science
Umeå University
Umeå, Sweden
drewes@cs.umu.se

Anna Jonsson
Dept. Computing Science
Umeå University
Umeå, Sweden
aj@cs.umu.se

Abstract

Graph-based semantic representations are valuable in natural language processing, where it is often simple and effective to represent linguistic concepts as nodes, and relations as edges between them. Several attempts has been made to find a generative device that is sufficiently powerful to represent languages of semantic graphs, while at the same allowing efficient parsing. We add to this line of work by introducing graph extension grammar, which consists of an algebra over graphs together with a regular tree grammar that generates expressions over the operations of the algebra. Due to the design of the operations, these grammars can generate graphs with *non-structural reentrancies*, a type of node-sharing that is excessively common in formalisms such as abstract meaning representation, but for which existing devices offer little support. We provide a parsing algorithm for graph extension grammars, which is proved to be correct and run in polynomial time.

Index Terms

algorithmic graph theory, semantic representations, hyperedge replacement grammars, natural language processing

I. Introduction

Semantic representations are a key component in algorithmic solutions for high-level language processing. Word embeddings such as BERT [14] and GPT-3 [9] have become a popular choice of representation, enabling rapid progress on many tasks. Due to the descriptive power of the neural networks that generate them, they rarely submit to analysis by deductive means. However, for some applications the transparency gained by formal methods is a must. In such cases, we believe that graph-based semantic representations offer a useful complement. These generally represent objects as nodes, and relations as directed edges. Probabilities and other weights can be added to reflect quantitative aspects such as likelihoods and uncertainties.

In this work, we model languages of graph-based semantic representations as the combination of a graph algebra $\mathcal{A}$ together with a tree grammar g whose generated trees are well-formed

This work has been supported in part by the Swedish Research Council under Grant No 2020-03852

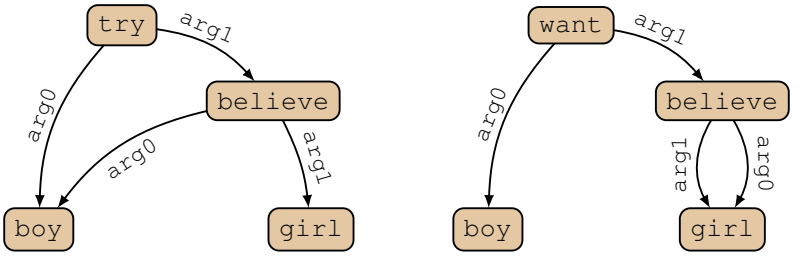


Fig. 1. AMR graphs for the pair of sentences "The boy tries to believe the girl" and "The boy wants the girl to believe herself". The nodes represent the concepts 'boy', 'girl', 'try', 'want', and 'believe'. The edges labelled 'arg0' give the agent for each verb, and the edges labelled 'arg0' the patient.

expressions over the operations of $\mathcal{A}$. Each tree translates into a graph when evaluated by $\mathcal{A}$, meaning that the tree language generated by g evaluates to a graph language. If we are careful about how we construct and combine g and $\mathcal{A}$, we can make parsing efficient. In other words, given a graph G, we can in polynomial time find a tree in the language of g that evaluates to G under $\mathcal{A}$ (or decide that no such tree exists). In the literature, this approach is known as tree-based generation [15]. The novelty in the present work lies in the design of the algebra.

For inspiration, we look at a particular instance of graph-based representations, namely abstract meaning representation (AMR) [30], [2]. It is characterised by its graphs being directed, acyclic, and having unbounded node degree. AMR was first introduced by [30] based on a semantic abstraction language by [28]. The notion was refined and popularized by [2] and instantiated for a limited domain by [8]. To ground AMR in formal language theory, Chiang et al. [11] analyse the AMR corpus of Banarescu et al [2]. They note that even though the node degree is generally low in practice, this is not always the case, which speaks in favour of models that allow an unbounded node degree. Regarding the treewidth of the graphs in the corpus, they find that it never exceeds 4 and conclude that an algorithm can depend exponentially on this parameter and still be feasible in practice.

In the context of semantic graphs, it is common to talk about *reentrancies*. Figure 1 illustrates this concept with a pair of AMR graphs, both of which require node sharing, or *reentrant edges*, to express the correct semantics. We propose to distinguish between two types of reentrancies: In the first graph, the reentrancy is *structural*, meaning that the boy must be the agent of whatever he is trying, so the 'arg0' edge can only point to him. In the second graph, the reentrancy is *non-structural* in the sense that although in this particular case the girl believes in herself, there is nothing to prevent her believing in someone else, or that someone else believes in her. In general, we speak of (i) structural reentrancies when they are syntactically governed, e.g., by control or coordination, and of (ii) non-structural reentrancies when they represent coreferences which can in principle refer to any antecedent, essentially disregarding the structure of the graph. Another example of (i) is subject control as in "They persuaded him to talk to her", where the person who talks and the person who was persuaded must be one and the same. In contrast, "[. . .], but she liked them" is an example of (ii) since antecedent of "them" may be picked from

anywhere in "[...]" for the semantic representation to be valid.

Another important characteristic of AMR and other notions of semantic graphs in natural language processing is that certain types of edges, like those in Figure 1, point to *arguments* of which there should be at most one. For example, the concept 'try' in Figure 1 must only have one outgoing 'arg0' edge. Other edges, of types not present in Figure 1, can for example represent modifiers, such as believing *strongly* and *with a passion*. Outgoing edges of these types are usually not limited in number.

We have previously shown that contextual hyperedge-replacement grammars (CHRGs), an extension of the well-known context-free hyperedge-replacement grammars (HRGs), can handle both structural and non-structural reentrancies [22]. Moreover, a parser generator has been provided by Drewes, Hoffmann, and Minas for CHRGs that, when it succeeds in producing a parser, guarantees that the parser will run in quadratic (and in the common case, linear) time in the size of its input graph [19], [21]. However, similarly to common LL- and LR-parsers for context-free languages the parser generator may discover a parsing conflict, thus failing to output a parser. Compared to the string case, the possible reasons for conflicts are much subtler, which makes grammar construction a complex and error-prone endeavour.

For this reason, we are now proposing a CHRG variant that by construction allows for polynomial parsing while retaining the ability to specify graph languages with reentrancies of both type (i) and (ii). One key property of these grammars that enables polynomial parsing is that, intuitively, nodes are provided with all of their outgoing edges the moment they are created. Using ordinary contextual hyperedge replacement, this would thus result in graph languages of bounded out-degree, the bound being given by the maximal number of outgoing edges of the nodes in the right-hand sides of production rules. To be able to generate semantic graphs in which a concept can have an arbitrary number of edges representing modifiers, we use a technique from [17] known as *cloning*. To incorporate both contextuality and cloning in a natural way without sacrificing efficient parsability, we formalise our grammars using the concept of *tree-based graph generation*. While tree-based hyperedge-replacment grammars are well-known to be equivalent to ordinary ones (see below), the tree-based formulation does make a difference in the contextual case as it avoids the problem of cyclic dependencies that has not yet been fully characterised and can make parsing intractable [20].

A. Related work

Tree-based generation dates back to the seminal paper by Mezei and Wright [34] which generalises context-free languages to languages over arbitrary domains, by evaluating the trees of a regular tree language with respect to an algebra.[1] Operations on graphs were used in this way for the first time by Bauderon and Courcelle [3]. See the textbook [13] by Courcelle and Engelfriet for the eventual culmination of this line of work. Graph operations similar to those used here (though without the contextual extension) appeared first in Courcelle's work on the monadic-second order logic of graphs [12, Definition 1.7]. Essentially, if the right-hand side of a production contains k nonterminal hyperedges, it is viewed as an operation that takes k

[1]Mezei and Wright formulate this in terms of systems of equations, but the essential ideas are the same.

hypergraphs as arguments (corresponding to the hypergraphs generated by the k subderivations) and returns the hypergraph obtained by replacing the nonterminals in the right-hand side by those k argument hypergraphs. By context-freeness, evaluating the tree language that corresponds to the set of derivation trees of the grammar (which is a regular tree language) yields the same set of hypergraphs as is generated by the original HRG.

Several formalisms have been put forth in the literature to describe graph-based semantic representations in general and AMR in particular. Most of these can be seen as variations of HRGs; see [27], [3], [23]. It was established early that unrestricted HRGs can generate NP-complete graph languages [1], [29], so restrictions are needed to ensure efficient parsing. To this end, Lautemann [31] proposed a CYK-like membership algorithm and proved that it runs in polynomial time provided that the language satisfies the following condition: for every graph G in the language, the number of connected components obtained by removing s nodes from G is in $O(\log n)$, where n is the number of nodes of G and the constant s depends on the grammar. Lautemann's algorithm was refined by Chiang et al. [10] to make it more suitable for NLP tasks, but the algorithm is exponential in the node degree of the input graph.

In a parallel line of work, Quernheim and Knight [35] define automata for directed acyclic graphs (DAGs) to process feature structures in machine translation. Chiang et al. [11] propose an extended model of DAG automata, focusing on semantic representations such as AMR. Towards this end, they extend DAG automata by allowing the left- and right-hand sides to have the form of a restricted regular expression. Blum and Drewes [7] complement this work by studying language-theoretic properties of the DAG automata. They establish, among other things, that equivalence and emptiness are decidable in polynomial time.

Various types of hyperedge replacement graph algebras for AMR parsing are described in the work by Groschwitz et al., see, e.g. [25], [24], [26], [32], [33]. A central objective is to find linguistically motivated restrictions that can efficiently be trained from data. An algorithm based on such an algebra for translating strings into semantic graphs was presented in [26]: operations of arity zero denote graph fragments, and operations of arity two denote binary combinations of graph fragments into larger graphs. The trees over the operations of this algebra mirror the compositional structure of semantic graphs. The approach differs from ours in that the graph operations are entirely deterministic, and that neither context nodes nor cloning are used. Moreover, as is common in computational linguistics, evaluation is primarily empirical.

Through another set of syntactic restrictions on HRGs, Björklund et al. [6], [5] arrive at order-preserving dag grammars (OPDGs) – with or without weights – which can be parsed in linear time. Intuitively, the restrictions combine to ensure that each generated graph can be uniquely represented by a tree in a particular graph algebra. Despite their restrictions, OPDGs can describe central structural properties of AMR, but their limitation lies in the modelling of reentrancies. Out of the previously discussed types of reentrancies, type (i) can to a large extent be modelled using OPDGs. Modelling type (ii) cannot be done (except in very limited cases) since it requires attaching edges to the non-local context in an stochastic way, which cannot be achieved using hyperedge replacement alone.

CHRGs [18] extend the ordinary HRGs with so-called *contextual* rules, which allow for

isolated nodes in their left-hand-sides. Contextual rules can reference previously generated nodes, that are not attached to the replaced nonterminal hyperedge, and add structure to them. Even though this formalism is strictly more powerful than HRG, it inherits several of the nice properties of HRG. In particular, there are useful normal forms and the membership problem is in NP [16]. Paralleling previous work on HRG, we set out to find a syntactically restricted subclass of CHRG, enriched by a benign form of cloning, that is sufficiently descriptive to model AMR, while having a membership problem in P.

II. Graphs and Graph Extension Grammars

We first recall some standard definitions and notations from discrete mathematics, automata theory, and algorithmic graph theory. The set of natural numbers (including 0) is denoted by $\mathbb{N}$, and for $n \in \mathbb{N}$, $[n]$ denotes $\{1, \dots, n\}$. In particular, $[0] = \emptyset$. For a set S, the set of all finite sequences over S is denoted by S^*; the subset of S^* containing only those sequences in which no element of S occurs twice is denoted by $S^{\circledast}$. Both contain the empty sequence ε. For $w \in S^*$, $[w]$ denotes the set of all elements of S occurring in w.[2] The canonical extensions of a function $f\colon S \to T$ to S^* and to $\wp(S)$ are denoted by f as well, i.e., $f(s_1 \cdots s_n) = f(s_1) \cdots f(s_n)$ for $s_1, \dots, s_n \in S$ and $f(S') = \{f(s) \mid s \in S'\}$ for all $S' \subseteq S$. The restriction of $f\colon S \to T$ to $S' \subseteq S$ is denoted by $f|_{S'}$. The power set of S is denoted by $\wp(S)$. For functions $f_1\colon S_1 \to T_1$ and $f_2\colon S_2 \to T_2$ by $f_1 \cup f_2$, we let $f_1 \sqcup f_2\colon S_1 \cup S_2 \to T_1 \cup T_2$ be given by

$$(f_1 \sqcup f_2)(s) = \begin{cases} f_1(s) & \text{if } s \in S_1 \\ f_2(s) & \text{if } s \in S_2 \setminus S_1 \end{cases} .$$

Trees and Algebras

As sketched in the introduction, we generate trees over an algebra $\mathcal{A}$ of graph operations and then evaluate each tree t in a non-deterministic fashion to a set of graphs $val_{\mathcal{A}}(t)$. To be able to ensure that the graph operations are always well defined on their arguments, we work with *typed* trees. Let $\mathcal{T}$ be a set of *types*. A *$\mathcal{T}$-typed signature* is a pair $A = (\Sigma, \sigma)$ such that Σ is a finite set of symbols and σ is a function that assigns to every $f \in \Sigma$ a pair $\sigma(f) \in \mathcal{T}^* \times \mathcal{T}$. We usually keep σ implicit, thus identifying A with Σ, and write $f\colon \tau_1 \cdots \tau_k \to \tau$ to indicate that $\sigma(f) = (\tau_1 \cdots \tau_k, \tau)$. The number k is said to be the *rank* of f in Σ. Furthermore, we may drop the epithet $\mathcal{T}$-typed when speaking of signatures in case $\mathcal{T}$ is of little relevance or understood from context.

The family $\mathrm{T}_\Sigma = (\mathrm{T}_\Sigma^\tau)_{\tau \in \mathcal{T}}$ of all well-formed trees over a signature Σ is defined inductively, as usual. It is the smallest family of sets T_Σ^τ of formal expressions such that, for every $f\colon \tau_1 \cdots \tau_k \to \tau$ and all $t_1 \in \mathrm{T}_\Sigma^{\tau_1}, \dots, t_k \in \mathrm{T}_\Sigma^{\tau_k}$, we have $f[t_1, \dots, t_k] \in \mathrm{T}_\Sigma^\tau$. Note that, for every tree $t \in \bigcup_{\tau \in \mathcal{T}} \mathrm{T}_\Sigma^\tau$ there is a unique type τ such that $t \in \mathrm{T}_\Sigma^\tau$. Therefore, we may simply write $t \in \mathrm{T}_\Sigma$ and call τ the type of t.

We refer to the subtrees of a tree by their Gorn addresses. Thus, we define the set $addr(t) \subseteq \mathbb{N}^*$ of *addresses* in a tree t, and the subtrees they designate, in the usual way: for $t = f[t_1, \dots, t_k]$,

[2]The similarity to the notation $[n]$, $n \in \mathbb{N}$, is intentional, since both operations turn single elements into sets.

$addr(t) = \{\varepsilon\} \cup \bigcup_{i\in[k]}\{i\alpha \mid \alpha \in addr(t_i)\}$ (note that the basis of the recursion is the case $k = 0$). Moreover, for all $\alpha \in addr(t)$

$$t/\alpha = \begin{cases} t & \text{if } \alpha = \varepsilon \\ t_i/\alpha' & \text{if } \alpha = i\alpha' \text{ for some } i \in [k] \text{ and } \alpha' \in addr(t_i) \end{cases} .$$

Let Σ be a $\mathcal{T}$-typed signature. A Σ-*algebra* is a pair $\mathcal{A} = ((\mathbb{A}_\tau)_{\tau\in\mathcal{T}}, (f_\mathcal{A})_{f\in\Sigma})$ where $\mathbb{A}_\tau$ is a set for each $\tau \in \mathcal{T}$ and $f_\mathcal{A}$ is a function $f_\mathcal{A} \colon \mathbb{A}_{\tau_1} \times \cdots \times \mathbb{A}_{\tau_k} \to \mathbb{A}_\tau$ for every $f \colon \tau_1 \cdots \tau_k \to \tau$ in Σ. Given a tree $t \in \mathrm{T}_\Sigma$, the result of evaluating t with respect to $\mathcal{A}$ is denoted by $val_\mathcal{A}(t)$. It is defined recursively: if $t = f[t_1, \ldots, t_k]$ then

$$val_\mathcal{A}(t) = f_\mathcal{A}(val_\mathcal{A}(t_1), \ldots, val_\mathcal{A}(t_k)) .$$

Regular Tree Languages

To generate trees (i.e., expressions) over the operations of an algegra, we use *regular tree grammars*. Our interest is in algebras over graph domains, but to illustrate the general idea, we can look at an algebra over the rational numbers $\mathbb{Q}$. In this setting, the operations could be the standard arithmetic ones, such as $+, -, /, \times$, all binary, together with constant-valued operations in $\mathbb{Q}$, all of arity zero. It is hopefully easy to see how the trees $+[\times[2, \frac{1}{2}], 2]$ and $/[10, -[10, 4]]$ evaluate to 2 and $\frac{5}{3}$, respectively. Moreover, by introducing types, we may work with an algebra over a heterogeneous domain, e.g., natural numbers and rationals, and use the types to ensure that when a tree is evaluated, operations are always applied to suitable arguments.

Definition 1 (regular tree grammar)**.** *Let $\mathcal{T}$ be a set of types. A $\mathcal{T}$-typed* regular tree grammar *(or, for brevity, regular tree grammar) is a system $g = (N, \Sigma, P, S)$ consisting of the following components:*

- *N is a $\mathcal{T}$-typed signature of symbols of rank 0 called* nonterminals*; we write N_τ to denote the set of all nonterminals $A \in N$ such that $A \colon \varepsilon \to \tau$, where $\tau \in \mathcal{T}$.*
- *Σ is a $\mathcal{T}$-typed signature of* terminals *which is disjoint with N.*
- *P is a finite set of* productions *of the form $A \to f[A_1, \ldots, A_k]$ where, for some types $\tau, \tau_1, \ldots, \tau_k \in \mathcal{T}$, we have $f \colon \tau_1 \cdots \tau_k \to \tau$, $A_i \in N_{\tau_i}$ for all $i \in [k]$, and $A \in N_\tau$.*
- *$S \in N_\tau$ is the* initial nonterminal *for some type $\tau \in \mathcal{T}$.*

We also say that g is a regular tree grammar over *Σ.*

Definition 2 (regular tree language)**.** *Let $g = (N, \Sigma, P, S)$ be a regular tree grammar. The family $(L_A(g))_{A\in N}$ is the smallest family of sets of trees such that, for all $A \in N$, $L_A(g)$ is the set of all trees $f[t_1, \ldots, t_k]$ such that there is a production $(A \to f[A_1, \ldots, A_k]) \in P$ for some $k \in \mathbb{N}$ and $A, A_1, \ldots, A_k \in N$, and $t_i \in L_{A_i}(g)$ for all $i \in [k]$. The* regular tree language *generated by g is $L(g) = L_S(g)$.*

Note that, as can be shown by a straightforward induction, $L_A(g) \subseteq \mathrm{T}^\tau_\Sigma$ for all $A \in N_\tau$. In particular $L(g) \subseteq \mathrm{T}^\tau_\Sigma$ where τ is the type of S, i.e., $S \in N_\tau$.

Graphs and Graph Operations

We now define the type of graphs used in this paper, and the operations used to construct them. In short, we work with node- and edge-labelled directed graphs, each equipped with a finite set of ports. From a graph operation point of view, the sequence of ports is the "interface" of the graph; its nodes are the only ones that can individually be accessed. To ensure that the resulting graph is well-formed, we need to pay attention to the number of ports, which determines the *type* of the graph.

Definition 3 (graph)**.** *A* labelling alphabet *(or simply* alphabet*) is a pair* $\mathbb{L} = (\dot{\mathbb{L}}, \bar{\mathbb{L}})$ *of finite sets* $\dot{\mathbb{L}}$ *and* $\bar{\mathbb{L}}$ *of labels. A* graph *(over* $\mathbb{L}$*) is a system* $G = (V, E, lab, port)$ *where*

- V *is a finite set of* nodes*,*
- $E \subseteq V \times \bar{\mathbb{L}} \times V$ *is a (necessarily finite) set of* edges*,*
- $lab \colon V \to \dot{\mathbb{L}}$ *assigns a node label to every node, and*
- $port \in V^{\circledast}$ *is a sequence of nodes called* ports*.*

The type *of* G *is* $|port|$*. The set of all graphs (over an implicitly understood labelling alphabet) of type* τ *is denoted by* $\mathbb{G}_\tau$*. For an edge* $e = (u, \ell, v) \in E$*, we let* $src(e) = u$ *and* $tar(e) = v$*.*

An *isomorphism* (on G) is a bijective function $\mu \colon V \to V'$ for some set V' of nodes. It maps G to the graph $\mu(G) = (V', E', \mu \circ lab, \mu(port))$ where $E' = \{(\mu(u), \ell, \mu(v)) \mid (u, \ell, v) \in E\}$. If such an isomorphism exists, then the graphs G and $\mu(G)$ are said to be *isomorphic*.

The graph operations are of two kinds. The first is essentially disjoint union where the port sequences of the argument graphs are concatenated. Let $\tau, \tau' \in \mathbb{N}$. The operation $\uplus_{\tau\tau'} \colon \mathbb{G}_\tau \times \mathbb{G}_{\tau'} \to \mathbb{G}_{\tau+\tau'}$ is defined as follows: for $G \in \mathbb{G}_\tau$ and $G' \in \mathbb{G}_{\tau'}$, $\uplus_{\tau\tau'}(G, G')$ yields the graph in $\mathbb{G}_{\tau+\tau'}$ obtained by making the two graphs disjoint through a suitable renaming of nodes and taking their union. This is defined in the obvious way for the first three components and concatenates the port sequences of both graphs. Thus, the union operation $\uplus_{\tau\tau'}$ is not commutative and only defined up to isomorphism. We usually write $G \uplus_{\tau\tau'} G'$ instead of $\uplus_{\tau\tau'}(G, G')$. To avoid unnecessary technicalities, we shall generally assume that G and G' are disjoint from the start, and that no renaming of nodes takes place. We extend $\uplus_{\tau\tau'}$ to an operation $\uplus_{\tau\tau'} \colon \wp(\mathbb{G}_\tau) \times \wp(\mathbb{G}_{\tau'}) \to \wp(\mathbb{G}_{\tau+\tau'})$ in the usual way: for $\mathcal{G} \subseteq \mathbb{G}_\tau$ and $\mathcal{G}' \in \mathbb{G}_{\tau'}$, $\mathcal{G} \uplus_{\tau\tau'} \mathcal{G}' = \{G \uplus_{\tau\tau'} G' \mid G \in \mathcal{G}_\tau,\ G' \in \mathcal{G}'\}$.

To introduce the second type of graph operation, we first define an auxiliary nondeterministic *cloning operation*. The purpose of this operation is to make the expansion operations (to be defined afterwards) more flexible by allowing parts of it to be replicated an arbitrary number of times. This increases the flexibility with which it can attach to an existing structure. The operation was originally introduced to formalise the structure of object-oriented programs [17], and later adopted in computational linguistics [5].

To define the cloning operation, consider a graph $G = (V, E, lab, port)$ and let $C \subseteq V \setminus [port]$. Then $clone_C(G)$ is the set of all graphs obtained from G by replacing each of the vertices in C and their incident edges by an arbitrary number of copies (Figure 2 illustrates the construction). Formally, $G' = (V', E', lab', port') \in clone_C(G)$ if there is a family $(C_v)_{v \in V}$ of pairwise disjoint sets C_v of vertices, such that the following hold:

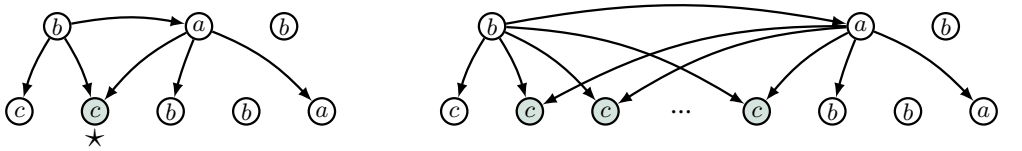


Fig. 2. The cloning operation takes a graph G and a subset C of its nodes, and replaces each of the nodes in C and their incidents edges by an arbitrary number of copies. The above figure shows G on the left-hand side, with C containing the single vertex marked with a star. On the right-hand side, the vertex and attaching structure has been replaced by a number of clones.

- $C_v = \{v\}$ for all $v \in V \setminus C$,
- $V' = \bigcup_{v \in V} C_v$,
- $E' = \bigcup_{(u,\ell,v) \in E} C_u \times \{\ell\} \times C_v$, and
- for all $v \in V$ and $v' \in C_v$, $lab'(v') = lab(v)$.

Note that cloning does *not* rename nodes in $V \setminus C$. In the following, we shall continue to denote by C_v ($v \in V$) the set of clones of v in a graph in $clone_C(G)$.

We can now define the second type of operation. It is a unary operation called *graph expansion operation* or simply *expansion operation*. We will actually use a restricted form, called *extension operation*, but define the more general expansion operation first. An expansion operation is described by a graph enriched by two additional components: let

$$\Phi = (V_\Phi, E_\Phi, lab_\Phi, port_\Phi, dock_\Phi, C_\Phi)$$

where both $\underline{\Phi} = (V_\Phi, E_\Phi, lab_\Phi, port_\Phi)$, henceforth called the *underlying graph* of Φ, and $(V_\Phi, E_\Phi, lab_\Phi, dock_\Phi)$ are graphs, and $C_\Phi \subseteq V_\Phi \setminus ([port_\Phi] \cup [dock_\Phi])$. We call $CONT_\Phi = V_\Phi \setminus ([port_\Phi] \cup [dock_\Phi])$ the set of *context nodes* and $NEW_\Phi = [port_\Phi] \setminus [dock_\Phi]$ the *new nodes*. The nodes in C_Φ are the *clonable nodes*. This terminology reflects that, according to the following definition, an application of Φ to a graph adds new (copies of the) nodes in NEW_Φ while those in $CONT_\Phi$ refer to nondeterministically chosen non-ports in the input graph. While every ordinary context node refers to exactly one node in the input graph, clonable context nodes refer to any number of nodes.

Throughout the rest of this paper, we shall continue to denote the components of an expansion operation Φ by V_Φ, E_Φ, lab_Φ, $port_\Phi$, $dock_\Phi$, and C_Φ.

Applying Φ to an argument graph $G = (V, E, lab, port) \in \mathbb{G}_\tau$ is possible if $\tau = |dock|$. It then yields a graph of type $\tau' = |port_\Phi|$ by fusing the nodes in $dock_\Phi$ with those in $port$. Moreover, all context nodes, i.e., all nodes in the set $CONT_\Phi$, are fused with arbitrary (pairwise distinct) nodes in $V \setminus [port]$ carrying the same label. Prior to this nodes in C_Φ can be cloned an arbitrary number of times. Thus, the application of the operation to G clones the nodes in C, fuses those in $dock_\Phi$ with those in $port$, and fuses every node in C_v ($v \in CONT_\Phi$) injectively with any node in $V \setminus [port]$ carrying the same label. The port sequence of the resulting graph is $port_\Phi$.

Formally, let $\tau = |dock_\Phi|$ and $\tau' = |port_\Phi|$. Then Φ is interpreted as the nondeterministic operation $\Phi\colon \mathbb{G}_\tau \to \wp(\mathbb{G}_{\tau'})$ defined as follows. Given a graph $G = (V, E, lab, port) \in \mathbb{G}_\tau$, a graph $H \in \mathbb{G}_{\tau'}$ is in $\Phi(G)$ if it can be obtained by the following steps:

1) Choose a graph $G' \in clone_{C_\Phi}(\underline{\Phi})$ and an isomorphism μ on G' such that $\mu(G') = (V', E', lab', port')$ satisfies
   a) $\mu(NEW_\Phi) \cap V = \emptyset$,
   b) $\mu(dock_\Phi) = port$, and
   c) for all $v \in CONT_\Phi$ and $v' \in \mu(C_v)$, it holds that $\mu(C_v) \subseteq V \setminus [port]$ and $lab_\Phi(v) = lab(v')$.
2) Define $H = (V \cup V', E \cup E', lab \sqcup lab', port')$.

Note that, by the definition of $\sqcup$, the labels of nodes in $dock_\Phi$ are disregarded, i.e., the nodes which are ports in G keep their labels as determined by lab.[3] Thus, in specifying an expansion operation Φ as above, the labels of nodes in $dock_\Phi$ can be dropped, essentially regarding these nodes as unlabelled ones.

In the following, we mainly view Φ as an operation $\Phi\colon \wp(\mathbb{G}_\tau) \to \wp(\mathbb{G}_{\tau'})$ defined as $\Phi(\mathcal{G}) = \bigcup_{G\in\mathcal{G}} \Phi(G)$ for all $\mathcal{G} \subseteq \mathbb{G}_\tau$. We call $\underline{\Phi} = (V_\Phi, E_\Phi, lab_\Phi, port_\Phi)$ the *underlying graph* of Φ, and may also denote Φ as $(G, dock, C)$ where $G = \underline{\Phi}$.

We now specialise expansion operations to extension operations by placing conditions on their structure. The intuition is to make sure that graphs are built bottom-up, i.e., Φ always extends the input graph by placing nodes and edges "on top", and in such a way that all nodes of the argument graph are reachable from the ports. For this, we require that only new ports – ports that do not correspond to ports of the input graph – have outgoing edges, and that all ports of the input graph that are not ports of the resulting graph carry incoming edges. Noting that the former set is NEW_Φ and the latter is $[dock_\Phi] \setminus [port_\Phi]$, this results in the following pair of formal requirements:

(R1) $\{src(e) \mid e \in E_\Phi\} \subseteq NEW_\Phi$, and
(R2) $[dock_\Phi] \setminus [port_\Phi] \subseteq \{tar(e) \mid e \in E_\Phi\}$.

Example 4. *Figure 3 depicts an extension operation together with a graph to which it can be applied. In Figure 4, we see three different graphs, all resulting from the application of the extension operation in Figure 3 to the graph in Figure 4. Note that the resulting graphs differ only with respect to the number of cloned nodes.*

A *graph extension algebra* is a Σ-algebra $\mathcal{A} = ((\wp(\mathbb{G}_\tau))_{\tau\in\mathcal{T}}, (f_\mathcal{A})_{f\in\Sigma})$ where $\mathcal{T}$ is a finite subset of $\mathbb{N}$ and every symbol in Σ is interpreted as an extension or union operation or the set $\{\phi\}$, where ϕ is the empty graph $(\emptyset, \emptyset, \emptyset, \varepsilon)$. Note that the operations of the algebra act on sets of graphs rather than on single graphs. This is necessary because of the nondeterministic nature of extension operations.

[3]This idea has its origins in personal discussions between the second author and Berthold Hoffmann during a visit in 2019.

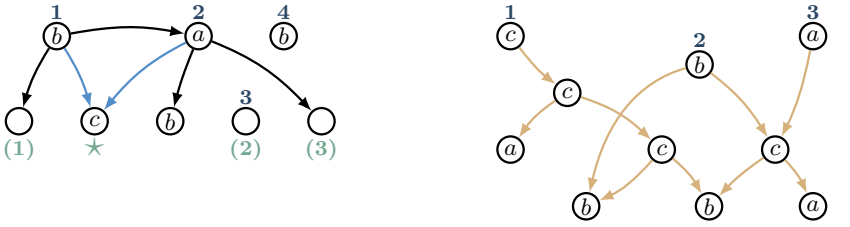


Fig. 3. The figure on the left shows an extension operation Φ with three ports (indicated with numbers above the vertices), three docks (indicated with numbers below the vertices), and one clonable vertex (indicated with a star). The figure on the right shows a graph G with three ports (again, indicated with numbers above the vertices). The operation Φ is applicable to G because (i) the number of ports of G coincides with the number of docks of Φ, so the types are compatible, and (ii) the pair of vertices in $CONT_\Phi$ can be identified with vertices in G outside of the ports of G (in several ways).

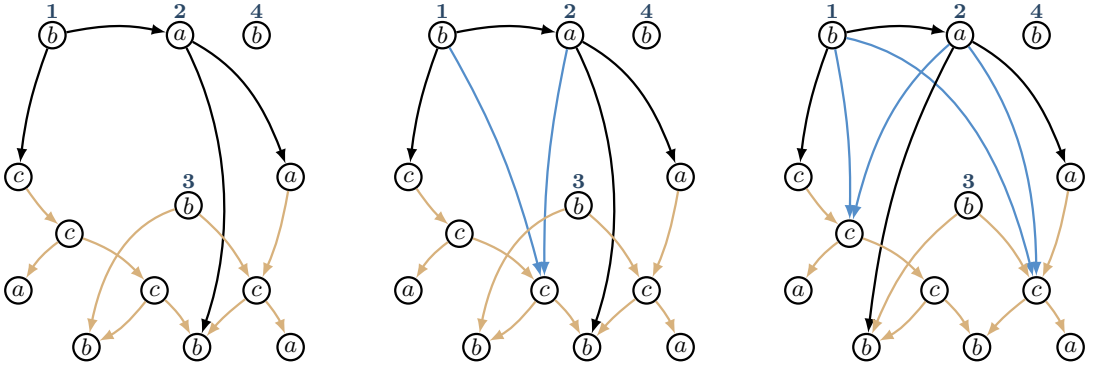


Fig. 4. Three graphs in $\Phi(G)$ where the extension operation Φ and the graph G are as in Figure 3. The differences between the graphs reflect the number of times the vertices in C_Φ have been replicated, and the choice of mapping from the vertices in $CONT_\Phi$ to vertices in G. The single vertex in C_Φ has been replicated 0 times to obtain the first graph, once to obtain the second, and twice to obtain the third. In the first two graphs, the non-dock node labelled b has been identified with one b-labelled node in G, and in the right graph with another.

Definition 5 (Graph Extension Grammar)**.** *A (tree-based)* graph extension grammar *is a pair* $\Gamma = (g, \mathcal{A})$ *where* $\mathcal{A}$ *is a graph extension* Σ*-algebra for some signature* Σ *and* g *is a regular tree grammar over* Σ*. The graph language generated by* Γ*, denoted by* $L(\Gamma)$*, is defined as*

$$L(\Gamma) = \bigcup_{t \in L(g)} val_{\mathcal{A}}(t).$$

To simplify notation, we shall in the following assume that, in a graph extension grammar as above, $f = f_{\mathcal{A}}$ for all $f \in \Sigma$, i.e., we use the operations themselves as symbols in Σ.

Before continuing on to parsing, let us pause to consider at a concrete example of graph extension grammar in the setting of natural language processing.

Example 6. *The graph extension grammar in Figure 5 serves to illustrate how the new formalism can model semantic representations. In this example, the concepts* `girl` *and* `boy` *represent persons that can act as agents (which verbs cannot),* `try` *and* `persuade` *are examples of verbs that require some level of structural control, and* `want` *and* `believe` *represent verbs where this is not needed.*

First, we take a top-down perspective to understand the tree generation. The initial nonterminal is S. The base case for S is the generation of an extension operation that creates a single node representing a `girl` *or a* `boy` *concept. In all other cases, S generates an extension operation that adds a verb and its outgoing edges, and in which the verb is the single port. The nonterminal C has the same function as S with the following two differences: it has no corresponding base case, and the ports of the extension operations mark both the agent of the verb (designated by an* `arg0` *edge) and the verb itself. The nonterminal S' can only generate extensions that create* `girl` *or* `boy` *nodes. As we shall see, this allows us to restrict the addition of* `arg0` *to persons. Finally, there is a pair of nonterminals U and U' that both create union operations, the former with two resulting ports, and the latter with three.*

Now, we take a bottom-up perspective to see how the extension operations of a tree are evaluated. Unless the tree consists of a single node, we will have several extensions generated by S and S' that create `girl` *and* `boy` *nodes as the leaves of the tree. In this case, we can apply one or more union operations to concatenate their port sequences and make them visible to further operations. The construction ensures that an* `arg0` *edge only can have a person, i.e., a valid agent, as its target. After the application of a union operation, it is possible to apply any extension that is applicable to U (or U'), meaning that none of the non-port nodes are contextual nodes. The resulting graph can have one or two ports – if the graph has one port, the applied extension operation was generated by S, and if the graph has two ports, the applied operation was generated by C. In other words: C signals that the graph is ready for the addition of a control verb, and S that the graph is a valid semantic graph with one port. When sufficiently many nodes have been generated, it is no longer necessary (but still possible) to use the union operations. Instead, we can add incoming edges to already generated nodes contextually (unless control is explicitly needed, as for control verbs). The generated graphs can thus contain both structural and non-structural dependencies. See Figure 6 for an example of a tree generated by the grammar in Figure 5, and its evaluation into a semantic graph.* ◇

Readers familiar with HRGs or Courcelle's hyperedge-replacement algebras [12] will have noticed that, when disregarding context nodes, productions using union and extension operations correspond to hyperedge replacement productions of two types:

(a) $A \to B \uplus_{\tau\tau'} C$ corresponds to a production with two hyperedges in the right-hand side, labelled B and C, where the first one is attached to τ of the nodes to which the nonterminal in the left-hand side (which is labelled A) is attached, and the second is attached to the remaining τ' nodes of the left-hand side. In particular, there are no further nodes in the right-hand side.

(b) $A \to \Phi[B]$ corresponds to a hyperedge-replacement production with a single nonterminal hyperedge labelled B in its right-hand side. Such productions are the ones responsible for

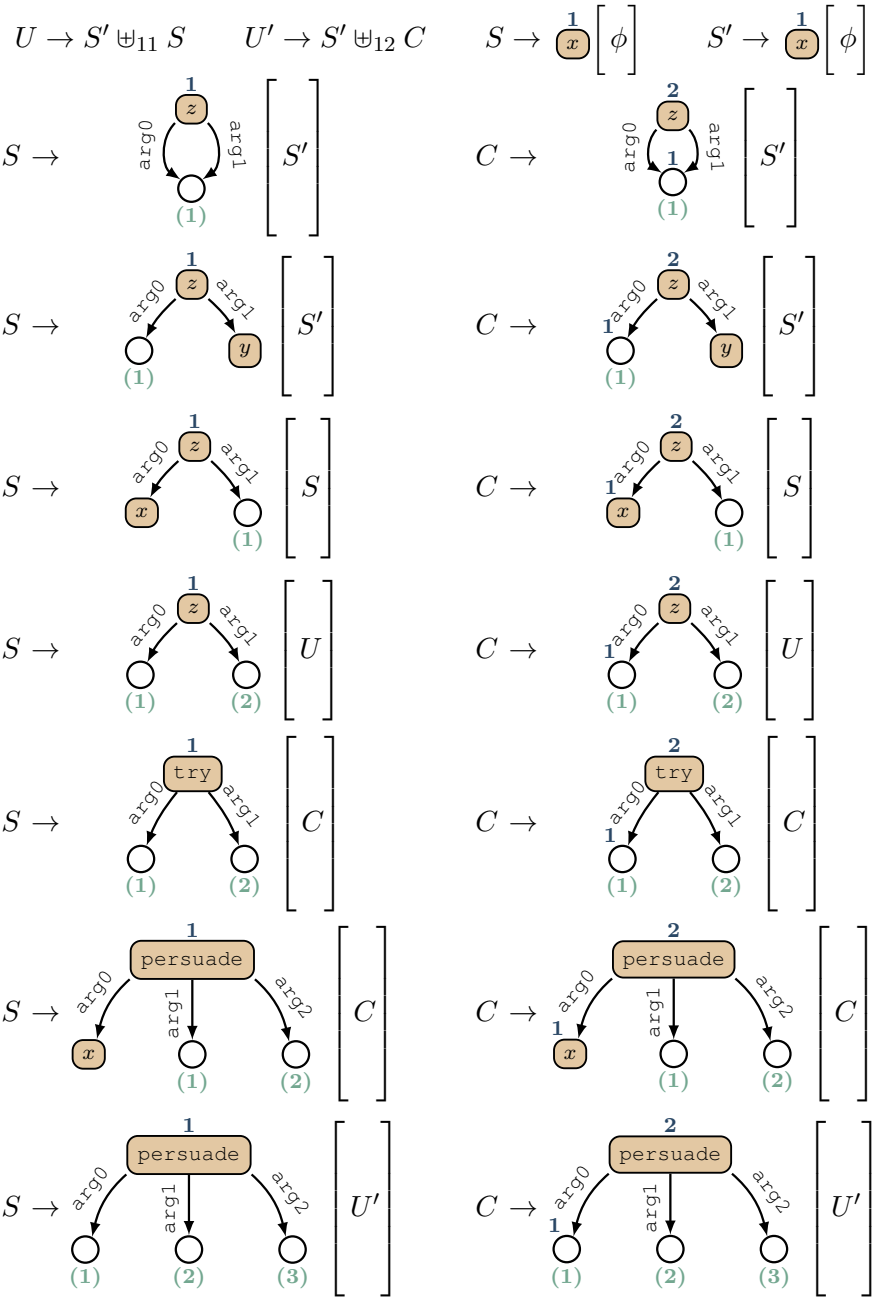


Fig. 5. A (graphical representation of a) graph extension grammar to demonstrate how our formalism can be used to generate semantic graphs akin to AMR. Instead of repeating productions that only differ with respect to node labels, we use variables representing labels to achieve a more compact representation, where $x \in \{\texttt{girl}, \texttt{boy}\}$, $y \in \{\texttt{girl}, \texttt{boy}, \texttt{want}, \texttt{believe}, \texttt{try}, \texttt{persuade}\}$, and $z \in \{\texttt{want}, \texttt{believe}\}$. The nonterminal C generates graphs of type 2 whose ports are to be embedded in a control verb construction.

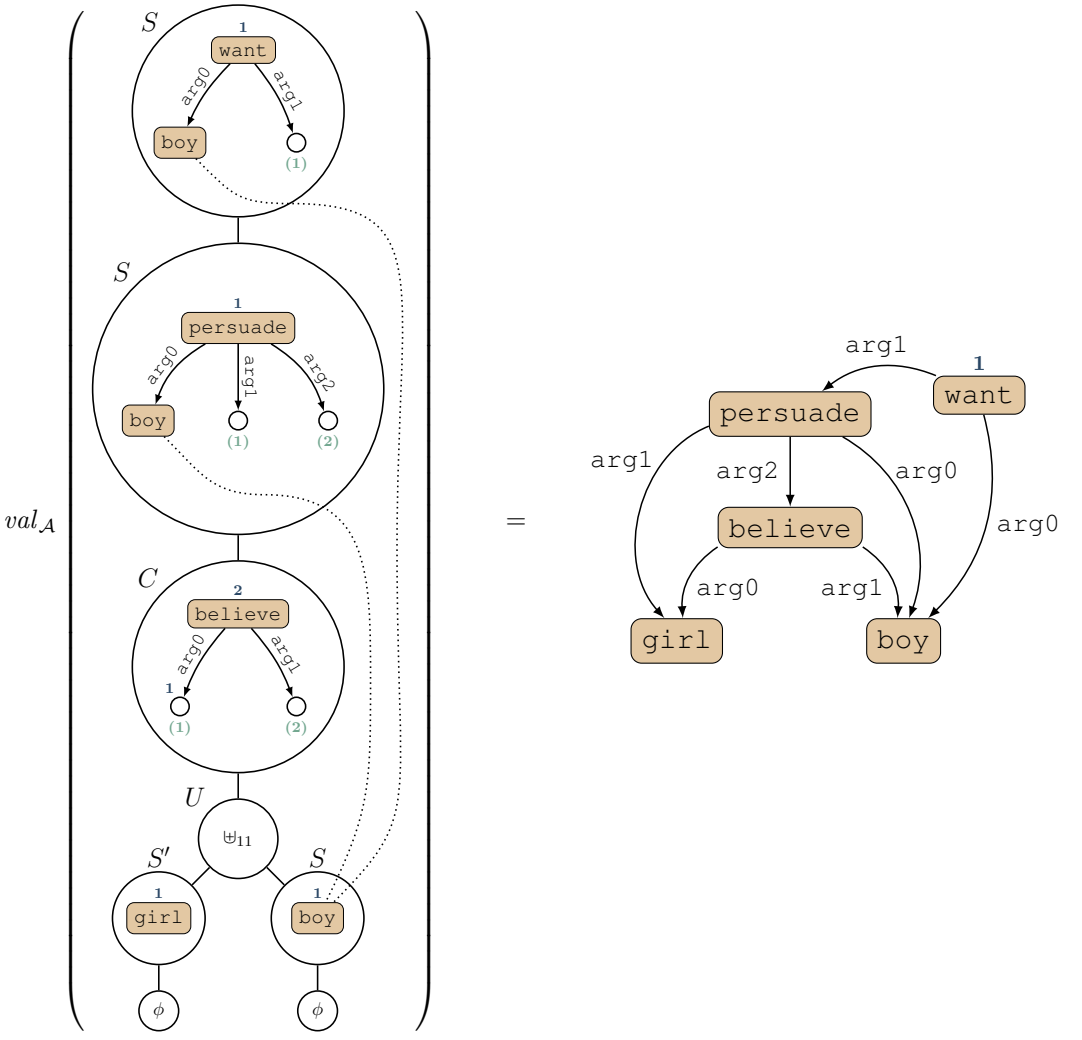


Fig. 6. A (graphical representation of a) tree generated by the regular tree grammar in Figure 5 and the graph resulting from its evaluation. The nonterminal generating each subtree is shown next to the respective circle. The identification of context nodes with previously created nodes is indicated by dotted lines. In this example, there is only one possible candidate for a context node, hence the resulting graph is unique up to isomorphism.

actually generating new terminal items (nodes and edges).

Generalized extension operations of arity greater than 1 can be constructed by defining so-called derived operations through composition of suitable extension and union operations. Given a set of such derived operations, the tree-to-tree mapping that replaces each occurrence of a derived operation by its definition is a tree homomorphism. As regular tree languages are closed under tree homomorphisms, this shows that the restriction to binary unions and unary extensions is no limitation (in contrast to requirements (R1) and (R2), which ensure polynomial parsability).

III. PARSING

We now provide a basic polynomial parsing algorithm for graph extension grammars. For this, and throughout this section, let $\Gamma = (g, \mathcal{A})$ be a graph extension grammar, where $g = (N, \Sigma, P, S)$ and $\mathcal{A}$ is a graph extension Σ-algebra. The goal is parsing, i.e., to decide the membership problem for Γ, and in the positive case, produce a witness derivation in Γ. The membership problem is stated as follows:

Input: A graph G
Question: Does it hold that $G \in L(\Gamma)$?

By definition, extension operations keep the identities of vertices in the input graph unchanged, whereas new vertices added to the graph may be given arbitrary identities (as long as they are not in the input graph). For the union operation, it holds that renaming of vertices is necessary only if the vertex sets of the two argument graphs intersect. As a consequence, when evaluating a tree t, we can without loss of generality assume that operations never change vertex identities of argument graphs. In other words, if $G \in val_{\mathcal{A}}(t)$, then there is a (concrete) graph $G_\alpha \in val_{\mathcal{A}}(t/\alpha)$ such that $G_\varepsilon = G$ and, for every $\alpha \in addr(t)$,

- if $t/\alpha = \Phi[t/\alpha 1]$, then $G_\alpha \in \Phi(G_{\alpha 1})$ and
- if $t/\alpha = t/\alpha 1 \uplus_{\tau\tau'} t/\alpha 2$, then $G_\alpha = (V \cup V', E \cup E', lab \sqcup lab', port\, port')$ where $G_{\alpha i} = (V_i, E_i, lab_i, port_i)$ for $i \in [2]$.

As a consequence, for every $\alpha \in addr(t)$, the graph G_α is a subgraph of G. More precisely, if $G = (V, E, lab, port)$ and $G_\alpha = (V', E', lab', port')$ then $V' \subseteq V$, $E' \subseteq E$, and $lab' = lab|_{V'}$. Note that, while the graphs G_α are usually not uniquely determined by G and t, the important fact is that they do exist if and only if $G \in val_{\mathcal{A}}(t)$. We say in the following that the family $(G_\alpha)_{\alpha \in addr(t)}$ is a *concrete evaluation of t into G*. Hence, the membership problem amounts to decide, for the input graph G, whether there is a tree $t \in L(g)$ that permits a concrete evaluation into G.

The following lemma, which forms the basis of our parsing algorithm, shows that G_α is determined by $port'$ alone, as it is the subgraph of G induced by the nodes that are reachable in G from $port'$ in G. More precisely, given a sequence $p \in V^{\circledast}$ of nodes of G, let $G \triangledown p = (V', E|_{V' \times \bar{\mathbb{L}} \times V'}, lab|_{V'}, p)$, where V' is the set of nodes reachable in G by directed paths from (any of the nodes in) $port$.

Lemma 7. *Let $(G_\alpha)_{\alpha \in addr(t)}$ be a concrete evaluation of a tree $t \in \mathrm{T}_\Sigma$ into a graph G, and let $G_\alpha = (V_\alpha, E_\alpha, lab_\alpha, port_\alpha)$ for every $\alpha \in addr(t)$. Then $G_\alpha = G \triangledown port_\alpha$ for all $\alpha \in addr(t)$.*

Proof. Using the notations from the statement of the lemma, and furthermore denoting the set of all nodes that can be reached from $p \in V^{\circledast}$ in G by $p^{\triangledown}$, we first show the following claim:

Claim 1. For all $e \in E$ and $v \in V$, $src(e) \in V_\alpha$ implies $e \in E_\alpha$, and $v \in port_\alpha^{\triangledown}$ implies $v \in V_\alpha$.

We prove Claim 1 by induction on the length of α. For $G_\alpha = \varepsilon$, the statement holds trivially. Now, assume that it holds for some $\alpha \in addr(t)$. Since the empty graph ϕ has neither ports nor edges, two relevant cases remain.

Case 1. G_α is of the form $G_{\alpha 1} \uplus_{\tau\tau'} G_{\alpha 2}$.

Let $\{i, j\} = [2]$. We have to show that the statement holds for $G_{\alpha i}$. Thus, assume that $src(e) \in V_{\alpha i}$ and $v \in port^{\triangledown}_{\alpha i}$. Then $src(e) \in V_\alpha$ and thus, by the induction hypothesis, $e \in E_\alpha$. By the definition of $\uplus_{\tau\tau'}$ this means that $e \in E_{\alpha i}$. Furthermore, $v \in port^{\triangledown}_{\alpha i}$ implies $v \in port^{\triangledown}_{\alpha}$ and thus, again by the induction hypothesis, $v \in V_\alpha$. Hence, it remains to argue that no node in $G_{\alpha j}$ can be reached from $port_{\alpha i}$. This follows readily from the just established fact that there is no edge $e' \in V \setminus V_{\alpha i}$ for which $src(e') \in E_{\alpha i}$ (together with the fact that $V_{\alpha i} \cap V_{\alpha j} = \emptyset$).

Case 2. G_α is of the form $\Phi(G_{\alpha 1})$ for an extension operation Φ.

To show that the statement holds for $G_{\alpha 1}$, let $G_\alpha = (V_{\alpha 1} \cup V', E_{\alpha 1} \cup E', lab_{\alpha 1} \sqcup lab', port')$, where $(V', E', lab', port')$ is obtained from $\underline{\Phi}$ as in the definition of $\Phi(G_{\alpha 1})$. By the induction hypothesis, $src(e) \in V_\alpha$ implies $e \in E_\alpha$. Thus, $src(e) \in V_{\alpha 1}$ implies $e \in E_{\alpha 1}$ unless $e \in E'$. However, by requirement (R1), all $e \in E'$ satisfy $src(e) \in [port'] \setminus [port]$, which is equivalent to $src(e) \notin V_{\alpha 1}$ because $V_\alpha \setminus V_{\alpha 1} = [port'] \setminus [port]$. This shows that $e \in E_{\alpha 1}$.

Now, let $v \in port^{\triangledown}_{\alpha 1}$. If $v \notin V_{\alpha 1}$, consider the first edge e on a path from a node in $[port_{\alpha 1}]$ to v such that $tar(e) \notin V_{\alpha 1}$. Then $src(e) \in V_{\alpha 1}$ but $e \notin E_{\alpha 1}$, contradicting the previously established fact that $src(e) \in V_{\alpha 1}$ implies $e \notin E_{\alpha 1}$. This finishes the proof of Claim 1.

The other direction of the second part of Claim 1 remains to be shown:

Claim 2. $V_\alpha \subseteq port^{\triangledown}_{\alpha}$.

This time, we proceed by induction on the size of t/α. The statement is trivially true for $G_\alpha = \phi$. Thus, as before, there are two cases to distinguish.

Case 1. G_α is of the form $G_{\alpha 1} \uplus_{\tau\tau'} G_{\alpha 2}$.

By the induction hypothesis, Claim 2 holds for $G_{\alpha 1}$ and $G_{\alpha 2}$. Consequently, $V_\alpha = V_{\alpha 1} \cup V_{\alpha 2} \subseteq port^{\triangledown}_{\alpha 1} \cup port^{\triangledown}_{\alpha 2} = port^{\triangledown}_{\alpha}$.

Case 2. G_α is of the form $\Phi(G_{\alpha 1})$ for an extension operation Φ.

Again, let $G_\alpha = (V_{\alpha 1} \cup V', E_{\alpha 1} \cup E', lab_{\alpha 1} \sqcup lab', port')$, where $(V', E', lab', port')$ is obtained from $\underline{\Phi}$ as in the definition of $\Phi(G_{\alpha 1})$. (In particular, $port' = port_\alpha$.) By the induction hypothesis, $V_{\alpha 1} \subseteq port^{\triangledown}_{\alpha 1}$. Moreover, by requirement (R2), for every node $v \in [port_{\alpha 1}]$ it either holds that $v \in [port_\alpha]$, or there are $u \in [port_\alpha]$ and $e \in E'$ with $src(e) = u$ and $tar(e) = v$. Hence, $[port_{\alpha 1}] \subseteq port^{\triangledown}_{\alpha}$ and thus $V_{\alpha 1} \subseteq port^{\triangledown}_{\alpha 1} \subseteq port^{\triangledown}_{\alpha}$. Since, furthermore, $V' \setminus V_{\alpha 1} \subseteq [port_\alpha]$, this shows that $V_\alpha \subseteq [port_\alpha]$, and thus $V_\alpha \subseteq port^{\triangledown}_{\alpha}$, as claimed. □

We note the following immediate consequence of Lemma 7:

Corollary 8. *Let $(G_\alpha)_{\alpha \in addr(t)}$ be a concrete evaluation of a tree $t \in \mathrm{T}_\Sigma$ into a graph G. If $G_\alpha = G \triangledown \varepsilon$ for some $\alpha \in addr(t)$, then $G_\alpha = \phi$.*

It follows from Corollary 8 that a nonterminal A of type ε can only derive trees that evaluate to the empty graph ϕ. We can therefore replace all rules where A appears on the left-hand side

by a single rule that takes A to the empty graph, without changing the generated graph language. The construction suggests the following normal form for graph extension grammars.

Lemma 9 (normal form)**.** Γ *can be transformed into a graph extension grammar* $\Gamma' = (g', \mathcal{A}')$ *with* $L(\Gamma') = L(\Gamma)$ *such that the co-domain* s *of every extension and union operation in* $\mathcal{A}'$ *satisfies* $|s| \geq 1$.

Proof. We may assume that $L_A(g) \neq \emptyset$ for all $A \in N$. By Corollary 8, $\sigma(A) = \varepsilon$ then implies that $val_{\mathcal{A}}(L_A(g)) = \{\phi\}$. Hence, all rules with the left-hand side A can be replaced by a single rule $A \to \phi$. □

From here on, let us assume that Γ is in the normal form given by Lemma 9. Turning our attention back on Lemma 7, we find the beginnings of a recursive parsing algorithm: In the following, consider an input graph $G = (V, E, lab, port)$, a nonterminal $A \in N$, and a sequence $p \in V^{\circledast}$ with $\dot{\mathbb{L}}(p) = \sigma(A)$. To decide whether $G \triangledown p \in val_{\mathcal{A}}(L_A(g))$ (and thus whether $G \in L(\Gamma)$ if $A = S$ and $p = port_G$), there are two cases. If $p = \varepsilon$, then $G \triangledown p = \phi \in val_{\mathcal{A}}(L_A(g))$ if and only if the production $A \to \phi$ is in P. Otherwise, we need to check each production with the left-hand side A, as follows:

Case 1. If the production is of the form $A \to B_1 \uplus_{\tau_1 \tau_2} B_2$, then we recursively need to determine whether $G \triangledown p_i \in val_{\mathcal{A}}(L_{B_i}(g))$ for $i = 1, 2$, where $p = p_1 p_2$ and $|p_1| = \tau_1$.

Case 2. If the production is of the form $A \to \Phi(B)$, the basic intuition is that we need to check all possible mappings of the edges in E_Φ to edges in E that have their sources in p. However, since nodes in C_Φ can be cloned, and since some context nodes of Φ may be isolated, the technicalities are slightly more involved.

To simplify the reasoning, let us first assume that $\underline{\Phi}$ does not contain isolated context nodes, i.e., that it satisfies the following stronger variant of (R2):

(R3) $V_\Phi \setminus [port_\Phi] \subseteq \{tar(e) \mid e \in E_\Phi\}$.

Let $port_\Phi = u_1 \cdots u_k$, $p = v_1 \cdots v_k$, and define $NEW = \{v_i \mid i \in [k] \text{ and } u_i \notin [dock_\Phi]\}$. In other words, NEW contains the nodes in p which correspond to those in NEW_Φ, which are the only nodes that may have edges to other nodes in Φ. Now, let $E_P = \{e \in E \mid src(e) \in NEW\}$ and $V_p = [p] \cup \{tar(e) \mid e \in E_p\}$. Note that, by (R3), V_p contains all the nodes of V that must be put into correspondence with nodes in V_Φ, while the nodes in in $V \setminus V_p$ are to be left for future, recursive, invocations of the parsing algorithm. We say that a surjective mapping $m \colon V_p \to V_\Phi$ is a *matching of* Φ *to* $G \triangledown p$ if it:

(M1) preserves node labels, that is, $lab_\Phi(m(v)) = lab(v)$ for all $v \in V$ with $m(v) \notin [dock_\Phi]$,

(M2) clones only clonable nodes, meaning that $|m^{-1}(u)| = 1$ for all $u \in V_\Phi \setminus C_\Phi$, and

(M3) bidirectionally preserves edges, that is, $(v, \ell, v') \in E_p \iff (m(v), \ell, m(v')) \in E_\Phi$ for all $v, v' \in V_p$ and all $\ell \in \dot{\mathbb{L}}$.

Now, by Lemma 7, the parsing algorithm needs to check recursively whether there is a matching of Φ to $G \triangledown p$ such that $G \triangledown m^{-1}(dock_\Phi) \in val_{\mathcal{A}}(L_B(g))$.[4]

[4]Recall that $[dock_\Phi] \cap C_\Phi = \emptyset$, and thus $m^{-1}(dock_\Phi) \in V^{|dock_\Phi|}$.

Algorithm 1 Parse a graph G with respect to a graph extension grammar $\Gamma = (g, \mathcal{A})$

procedure PARSE($G = (V, E, lab, port)$)
  return PARSE_REC($S, port$) ▷ Invoke recursive procedure with initial nonterminal S
end procedure
procedure PARSE_REC($A \in N$, $p \in V^{\circledast}$)
  if $\sigma(A) = \varepsilon$ **then** ▷ Leverage normal form
    return $(A \to \phi) \in P$
  else if $result[A, p]$ defined **then** ▷ Use memoïzed result
    return $result[A, p]$
  end if
  memoïze $result[A, p] \leftarrow false$ ▷ No success so far
  for all $(A \to B_1 \uplus_{\tau_1 \tau_2} B_2) \in P$ **do**
    let $p = p_1 p_2$ where $|p_1| = \tau_1$
    if PARSE_REC(B_1, p_1) and PARSE_REC(B_2, p_2) **then**
      memoïze $result[A, p] \leftarrow true$ ▷ Memoïze success
      return $true$
    end if
  end for
  for all $(A \to \Phi[B]) \in P$ **do**
    for all matchings m of Φ to $G \triangledown p$ **do**
      if PARSE_REC($B, m^{-1}(dock_\Phi)$) **then**
        memoïze $result[A, p] \leftarrow true$ ▷ Memoïze success
        return $true$
      end if
    end for
  end for
  return $false$
end procedure

Finally, to avoid the stronger assumption (R3), we have to handle any isolated context nodes $u \in V_\Phi$. Since, by injectivity, such a node u must be mapped to any node v in $G \triangledown p$ with the same label, except those in V_p. Thus, we only need to check the existence of such nodes outside V_p. Note that clonable isolated context nodes u can (and must) be disregarded because they can be cloned zero times, thus having no effect. In summary, let

$$\#_\Phi(\ell) = |\{u \in CONT_\Phi \setminus C_\Phi \mid lab_\Phi(u) = \ell\}|$$

for all $\ell \in \dot{\mathbb{L}}$. Then a matching m must, in addition, satisfy the following condition:

(M4) For all $\ell \in \dot{\mathbb{L}}$, $G \triangledown p$ contains at least $\#_\Phi(\ell)$ nodes v with $lab(v) = \ell$ that are not in V_p.

Note that this condition is independent of m.

The pseudocode of the algorithm is shown in Algorithm 1. As we shall see, the algorithm runs in polynomial time in the size of the input graph. However, the maximum length of the type sequences used appears in the exponent, so the degree of the polynomial depends on a property of the grammar. In the following theorem, we let $|G| = |V| + |E|$ for a graph $G = (V, E, lab, port)$.

Theorem 10. *Let Σ be $\mathcal{T}$-typed and $c = \max\{\tau \mid \tau \in \mathcal{T}\}$. If applied to an input graph G, Algorithm 1 terminates in $O(|G|^{2c+1})$ steps and returns true if and only $G \in L(\Gamma)$.*

Proof. Let $n = |G|$. Correctness should be clear from Lemma 7, the assumption that Γ is in normal form (cf. Lemma 9), and the preceding discussion. Further, by the choice of c, there are at most $O(n^c)$ distinct parameters PARSE_REC can be called with. Since we consider a fixed graph extension grammar Γ, this is a polynomial. For the same reason, the two outer **for** loops of the algorithm are executed a constant number of times during each call. It remains to consider the loop starting on line 19. Undoubtedly, the naïve implementation that enumerates all matchings is exponential. However, it is clear from the loop body that we only really need to determine all sequences $d_1 \cdots d_k = m^{-1}(\delta_1 \cdots \delta_k)$, where $\delta_1 \cdots \delta_k = dock_\Phi$ and m is a matching of Φ to $G \triangledown p$. (Recall that m^{-1}, restricted to $[dock_\Phi]$, is a function by (M2).) Since there are no more than $O(n^c)$ such sequences it remains to be shown that, for each of them, it is possible to check in linear time whether it can be extended to a matching. It bears repeating that we, in the paragraphs that follow, are only looking to establish the existence of such an extension, not compute a full set of complete matchings, as that would be too costly.

Clearly, condition (M4), which is independent of the choice of the sequence $d_1 \dots d_k$, can be checked in linear time. Thus, it remains to be shown how to verify that the mapping $d_i \mapsto \delta_i$ can be extended to a matching under the assumption that (M4) is satisfied.

For this, let us use the notation of Case 2 in the discussion preceding the theorem (i.e., the definition of matchings following (R3)). In particular, $E_P = \{e \in E \mid src(e) \in NEW\}$ and $V_p = [p] \cup \{tar(e) \mid e \in E_p\}$. Consider a candidate sequence $d_1 \cdots d_k \in V_p^k$. We have to check whether there is a matching m such that $m(d_1 \cdots d_k) = \delta_1 \cdots \delta_k$. For this, let

$$lab'(u) = \begin{cases} lab_\Phi(u) & \text{if } u \in V_\Phi \setminus [dock_\Phi] \\ lab(d_i) & \text{if } u = \delta_i \text{ for some } i \in [k]. \end{cases}$$

The idea is now to compute a characterisation, or *profile*, of the nodes in V_Φ and V_p that allows us to quickly decide whether such a matching m exists. In short, we record for each node u a set of triples, where each triple describes an incoming edge to u that must be covered by the matching. For every node $u \in V_\Phi$, let $profile_\Phi(u) = \{(i, \ell, lab'(u)) \mid (u_i, \ell, u) \in E_\Phi\}$. Similarly, for $v \in V_p$, let $profile_G(v) = \{(i, \ell, lab(v)) \mid (v_i, \ell, v) \in E\}$. Figure 7 illustrates the construction with an example.

We now combine the profiles for the nodes in V_Φ into a single, composite, profile for all of Φ, giving us a set of sets of edge descriptions. For each such edge description, we count how many of its type we need to find. This can be an exact number, or a lower bound, depending on whether some of the target nodes are clonable and can be left out or replicated as desired. Thus, define $\Pi_\Phi = \{profile_\Phi(u) \mid u \in V_\Phi \setminus [port_\Phi]\}$ and let, for all $\pi \in \Pi_\Phi$,

$$mult(\pi) = \begin{cases} eq_m & \nexists\, u \in C_\Phi \colon profile_\Phi(u) = \pi \\ gt_m & \text{otherwise} \end{cases}$$

where $m = |\{u \in V_\Phi \setminus ([port_\Phi] \cup C_\Phi) \mid profile_\Phi(u) = \pi\}|$. Then, by the definition of matchings, there exists a matching $m \colon V_p \to V_\Phi$ such that $m(d_1 \cdots d_k) = \delta_1 \cdots \delta_k$ if and only if

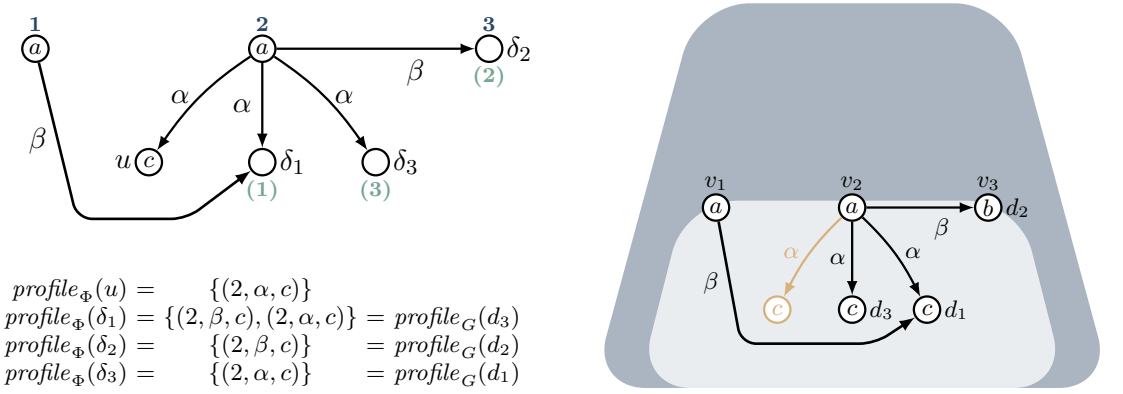


Fig. 7. On the top-left is a schematic view of an extension operation Φ with three ports and equally many docks. Deciding whether there is a matching m such that $m(d_i) = \delta_i$ for $i \in [3]$ boils down to computing the functions $profile_\Phi$ and $profile_G$ for all nodes in V_Φ and V_p, respectively. (Note that $profile_\Phi(\delta_i)$ depends on d_i.) Based on the result, we find that $G \triangledown p$ must contain the part drawn in brown for $d_1 d_2 d_3$ to extend to a matching, but only the existence of such an edge and node is relevant.

- the subgraph of G induced by $[p]$ is isomorphic to the subgraph of $\underline{\Phi}$ induced by $[port_\Phi]$,
- for all $i \in [k]$, $profile_\Phi(\delta_i) = profile_G(d_i)$,[5] and
- for all $\pi \in \Pi_\Phi$,

$$|\{v \in V_p \mid profile(v) = \pi\}| = k \quad \text{if } mult(\pi) = eq_k \text{ for some } k \in \mathbb{N}$$
$$|\{v \in V_p \mid profile(v) = \pi\}| \geq k \quad \text{if } mult(\pi) = gt_k \text{ for some } k \in \mathbb{N} \ .$$

It should be clear that these conditions can be checked in linear time. □

While Algorithm 1 only returns *true* or *false*, i.e., solves the pure membership problem, we can instead solve the actual parsing problem by changing the memoïzation to store for every nonterminal A and every p a tree that gives rise to the generated subgraph $G \triangledown p$ (or all of those trees).

We note here that the theoretical upper bound $O(|G|^{2c+1})$ proved above is unlikely to be reached in the average practical case, because an efficient implementation would only test candidate sequences $d_1 \cdots d_k$ such that $profile_\Phi(\delta_i) = profile_G(d_i)$ for all $i \in [k]$. In fact, there are many useful special cases for which better bounds can be proved. Two examples are mentioned in the next theorem, in which we use the following notation similar to the profiles used in the proof above, but restricted to the first and second components: For every node $u \in V_\Phi$, we let $profile'_\Phi(u) = \{(i, \ell) \mid (u_i, \ell, u) \in E_\Phi\}$ and $\Pi'_\Phi = \{profile'_\Phi(u) \mid u \in V_\Phi \setminus [port_\Phi]\}$.

[5] In fact, because of the previous item this only needs to be checked for $\delta_i \notin [port_\Phi]$.

Theorem 11. *Let Σ be $\mathcal{T}$-typed and $c = \max\{\tau \mid \tau \in \mathcal{T}\}$.*

1) Assume that, for every extension operation Φ of $\mathcal{A}$ and all $u \in [dock_\Phi] \setminus [port_\Phi]$ and $u' \in C_\Phi$, it holds that $profile'_\Phi(u) \neq profile'_\Phi(u')$. Then Algorithm 1 runs in time $O(|G|^{c+1})$.

2) Assume that the following hold:

a) For every extension operation Φ of $\mathcal{A}$ and all distinct $u \in [dock_\Phi] \setminus [port_\Phi]$ and $u' \in V_\Phi \setminus [port_\Phi]$, we have $profile'_\Phi(u) \neq profile'_\Phi(u')$.

b) For all distinct productions of the form $A \to \Phi[B]$ and $A \to \Phi'[C]$, where Φ and $\Phi' = (V'_\Phi, E'_\Phi, lab'_\Phi, dock'_\Phi, C'_\Phi)$, it holds that

$$\{profile'_\Phi(u) \mid u \in V_\Phi \setminus C_\Phi\} \not\subseteq \Pi'_{\Phi'} \quad or \quad \{profile'_{\Phi'}(u) \mid u \in V_{\Phi'} \setminus C_{\Phi'}\} \not\subseteq \Pi'_\Phi \ .$$

c) For all $A \in N$, if there is a production of the form $A \to B \uplus_{\tau\tau'} C$ in P, then there is no other production with the left-hand side A.

Then Algorithm 1 runs uniformly in linear time.

Proof. The assumption of the first statement implies that only a constant number of secquences $d_1 \cdots d_k$ in the proof of Theorem 10 can fulfil the requirement $profile_\Phi(\delta_i) = profile_G(d_i)$ for all $i \in [k]$. Hence, the **for** loop starting on line 19 can be implemented to run in linear time, which proves the statement.

For the second statement, note that the first assumption, which strengthens the assumption of the first part of the theorem, implies that at most one candidate sequence $d_1 \cdots d_k$ can fulfil the requirement that $profile_\Phi(\delta_i) = profile_G(d_i)$ for all $i \in [k]$. Moreover, by the second assumption, if there is such a sequence for a production $A \to \Phi[B]$ considered in the loop starting on line 18, then all other productions to be considered in that loop lack such a sequence. Finally, the third assumption makes sure that at most one production is to be considered on line 11, and if there is one, then there is no production of the form required on line 18.

It follows that the algorithm will decompose G in a deterministic top-down process, each step taking constant time. This proves the assertion. □

IV. Conclusion

We have introduced a graph grammar formalism that is both a restriction and an extension of hyperedge-replacement graph grammars, namely graph extension grammars. These grammars are formalised in a tree-based fashion, consisting of an algebra over graphs and a regular tree grammar that generates expressions over the operations of the algebra. The graphs are directed graphs with ports, constructed using operations of two kinds: The first is disjoint union, and the second is a family of unary operations that add new nodes and edges to an existing graph G. The augmentation is done in such a way that all new edges lead from a new node to either

(i) a port in G,
(ii) an arbitrary node in G, chosen only by its node label, or
(iii) any number of such nodes.

Graph extension grammars are essentially a restriction of contextual hyperedge-replacement grammars [18]. By way of example, we have shown that they can model both the structural and

non-structural reentrencies that are common in semantic graphs of formalisms such as AMR. We have provided a parsing algorithm for our formalism and proved it to be correct and have a polynomial-time running time. Finally, we have formulated conditions on the grammar that, if met, make parsing particularly efficient.

There are several promising directions for future work. On the theoretical side, we would like to apply the new ideas presented here to the formalisms by Groschwitz et al. [24] and Björklund et al. [4] to see if also these can be made to accommodate contextual rules without sacrificing parsing efficiency. It is also natural to generalise graph extension grammars to string-to-graph or tree-to-graph transducers to facilitate translation from natural-language sentences or dependency trees to AMR graphs. On the empirical side, we are interested in algorithms for inferring extension and union operations from AMR corpora, and in training neural networks to translate between sentences and AMR graphs using trees over graph extension algebras as an intermediate representation. Such efforts would make the new formalism available to current data-driven approaches in NLP, with the aim of adding structure and interpretability to machine-learning workflows.

REFERENCES

[1] IJsbrand Jan Aalbersberg, Grzegorz Rozenberg, and Andrzej Ehrenfeucht. On the membership problem for regular DNLC grammars. *Discrete Applied Mathematics*, 13(1):79–85, 1986.

[2] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse, Sofia, Bulgaria*, pages 178–186. Association for Computational Linguistics, 2013.

[3] Michel Bauderon and Bruno Courcelle. Graph expressions and graph rewritings. *Mathematical Systems Theory*, 20(1):83–127, 1987.

[4] Henrik Björklund, Frank Drewes, and Petter Ericson. Between a rock and a hard place - uniform parsing for hyperedge replacement DAG grammars. In Adrian-Horia Dediu, Jan Janousek, Carlos Martín-Vide, and Bianca Truthe, editors, *the 10th International Conference on Language and Automata Theory and Applications, LATA 2016, Prague, Czech Republic*, volume 9618 of *Lecture Notes in Computer Science*, pages 521–532. Springer, 2016.

[5] Henrik Björklund, Frank Drewes, and Petter Ericson. Parsing weighted order-preserving hyperedge replacement grammars. In *Proceedings of the 16th Meeting on the Mathematics of Language, Toronto, Canada*, pages 1–11. Association for Computational Linguistics, July 2019.

[6] Henrik Björklund, Frank Drewes, Petter Ericson, and Florian Starke. Uniform parsing for hyperedge replacement grammars. *Journal of Computer and System Sciences*, 118:1–27, 2021.

[7] Johannes Blum and Frank Drewes. Language theoretic properties of regular dag languages. *Information and Computation*, 265:57–76, 2019.

[8] Fabienne Braune, Daniel Bauer, and Kevin Knight. Mapping between English strings and reentrant semantic graphs. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation, LREC 2014, Reykjavik, Iceland*, pages 4493–4498. European Language Resources Association, 2014.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[10] David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Sofia, Bulgaria*, pages 924–932, 2013.

[11] David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. Weighted DAG automata for semantic graphs. *Computational Linguistics*, 44(1):119–186, 2018.

[12] Bruno Courcelle. The monadic second-order logic of graphs V: on closing the gap between definability and recognizability. *Theoretical Computer Science*, 80:153–202, 1991.

[13] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic – A Language-Theoretic Approach*. Cambridge University Press, 2012.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Minneapolis, Minnesota*, pages 4171–4186. Association for Computational Linguistics, 2019.

[15] Frank Drewes. *Grammatical Picture Generation – A Tree-Based Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[16] Frank Drewes and Berthold Hoffmann. Contextual hyperedge replacement. *Acta Informatica*, 52(6):497–524, 2015.

[17] Frank Drewes, Berthold Hoffmann, Dirk Janssens, and Mark Minas. Adaptive star grammars and their languages. *Theoretical Computer Science*, 411:3090–3109, 2010.

[18] Frank Drewes, Berthold Hoffmann, and Mark Minas. Contextual hyperedge replacement. In A. Schürr, D. Varro, and G. Varro, editors, *Applications of Graph Transformations with Industrial Relevance: 4th International Symposium, Revised Selected and Invited Papers*, number 7233 in Lecture Notes in Computer Science, pages 182–197. Springer, 2012.

[19] Frank Drewes, Berthold Hoffmann, and Mark Minas. Extending predictive shift-reduce parsing to contextual hyperedge replacement grammars. In Esther Guerra and Fernando Orejas, editors, *Proceedings of the 12th International Conference on Graph Transformation, ICGT 2017, Eindhoven, The Netherlands*, volume 11629 of *Lecture Notes in Computer Science*, pages 55–72. Springer International Publishing, 2017.

[20] Frank Drewes, Berthold Hoffmann, and Mark Minas. Predictive shift-reduce parsing for hyperedge replacement grammars. In Juan de Lara and Detlef Plump, editors, *Proceedings of the 10th International Conference on Graph Transformation, ICGT 2017, Marburg, Germany*, volume 10373 of *Lecture Notes in Computer Science*, pages 106–122. Springer, 2017.

[21] Frank Drewes, Berthold Hoffmann, and Mark Minas. Rule-based top-down parsing for acyclic contextual hyperedge replacement grammars. In *Proceedings of the 14th International Conference on Graph Transformation, ICGT 2021*, Lecture Notes in Computer Science, 2021.

[22] Frank Drewes and Anna Jonsson. Contextual hyperedge replacement grammars for abstract meaning representations. In *Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms, Umeå, Sweden*, pages 102–111. Association for Computational Linguistics, 2017.

[23] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. Hyperedge replacement graph grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 95–162. World Scientific Publishing Co., Inc., 1997.

[24] Jonas Groschwitz, Meaghan Fowlie, Mark Johnson, and Alexander Koller. A constrained graph algebra for semantic parsing with AMRs. In *Proceedings of the 12th International Conference on Computational Semantics (Long papers), IWCS 2017, Montpellier, France*. Association for Computational Linguistics, 2017.

[25] Jonas Groschwitz, Alexander Koller, and Christoph Teichmann. Graph parsing with s-graph grammars. In *Proc. 53rd Ann. Meeting of the Association for Computational Linguistics and the 7th Intl. Joint Conf. on Natural Language Processing (Volume 1: Long Papers)*, pages 1481–1490, 2015.

[26] Jonas Groschwitz, Matthias Lindemann, Meaghan Fowlie, Mark Johnson, and Alexander Koller. AMR dependency parsing with a typed semantic algebra. In I. Gurevych and Y. Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Long Papers), ACL 2018*, pages 1831–1841. Association for Computational Linguistics, 2018.

[27] Annegret Habel and Hans-Jörg Kreowski. Some structural aspects of hypergraph languages generated by hyperedge replacement. In Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 1987*, pages 207–219, Berlin, Heidelberg, 1987. Springer.

[28] Robert T. Kasper. A flexible interface for linking applications to Penman's sentence generator. In *Proceedings of the workshop on Speech and Natural Language, Philadelphia, Pennsylvania, HLT 1989*, pages 153–158. Association for Computational Linguistics, 1989.

[29] Klaus-Jörn Lange and Emo Welzl. String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. *Discrete Applied Mathematics*, 16(1):17 – 30, 1987.

[30] Irene Langkilde and Kevin Knight. Generation that exploits corpus-based statistical knowledge. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics (Volume 1), Montreal, Quebec*, pages 704–710, 1998.

[31] Clemens Lautemann. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27(5):399–421, 1990.

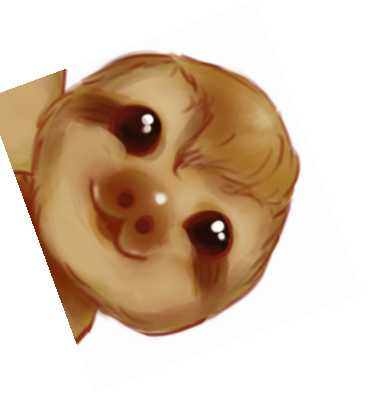
[32] Matthias Lindemann, Jonas Groschwitz, and Alexander Koller. Compositional semantic parsing across graphbanks. In A. Korhonen, D. R. Traum, and L. Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics (Volume 1: Long Papers), Florence, Italy, ACL 2019*, pages 4576–4585. Association for Computational Linguistics, 2019.

[33] Matthias Lindemann, Jonas Groschwitz, and Alexander Koller. Fast semantic parsing with well-typedness guarantees. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 3929–3951. Association for Computational Linguistics, 2020.

[34] Jorge Mezei and Jesse B. Wright. Algebraic automata and context-free sets. *Information and Control*, 11:3–29, 1967.

[35] Daniel Quernheim and Kevin Knight. Towards probabilistic acceptors and transducers for feature structures. In *Proceedings of the Sixth Workshop on Syntax, Semantics and Structure in Statistical Translation, Jeju, Republic of Korea*, pages 76–85. Association for Computational Linguistics, July 2012.

Department of Computing Science
Umeå University, SE-901 87, Umeå, Sweden
www.cs.umu.se



UMEÅ UNIVERSITY

ISBN 978-91-7855-522-2
ISBN 978-91-7855-521-5
ISSN 0348-0542
UMINF 21.04