

UMEÅ UNIVERSITY

Break between words

Efficient Best Trees Extraction and Contextual Grammars for Semantic Language Processing

Anna Jonsson

– Do oDPGs with context allow unique representations?

Thesis, June 2021
Department of Computing Science
Umeå University
Sweden

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 978-91-7601-964-1
ISSN 0348-0542
UMINF 18.12

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

Abstract

The overall goal of the field of *natural language processing* is to facilitate the communication between humans and computers, and to help humans with natural language problems such as translation. In this thesis, we focus on *semantic* language processing. Modelling semantics – the *meaning* of natural language – requires both a *structure* to hold the semantic information and a *device* that can enforce rules on the structure to ensure well-formed semantics while not being too computationally heavy. The devices used in natural language processing are preferably *weighted* to allow for comparison of the alternative semantic interpretations outputted by a device.

The structure employed here is the *abstract meaning representation (AMR)*. We show that AMRs representing well-formed semantics can be generated while leaving out AMRs that are not semantically well-formed. For this purpose, we use a type of graph grammar called *contextual hyperedge replacement grammar (CHRG)*. Moreover, we argue that a more well-known subclass of CHRG – the *hyperedge replacement grammar (HRG)* – is not powerful enough for AMR generation. This is due to the limitation of HRG when it comes to handling co-references, which in its turn depends on the fact that HRGs only generate graphs of *bounded treewidth*.

Furthermore, we also address the N *best problem*, which is as follows: Given a weighted device, return the N best (here: smallest-weighted, or more intuitively, smallest-errored) structures. Our goal is to solve the N best problem for devices capable of expressing sophisticated forms of semantic representations such as CHRGs. Here, however, we merely take a first step consisting in developing methods for solving the N best problem for *weighted tree automata* and some types of *weighted acyclic hypergraphs*.

Acknowledgements

Thanks to:

- ⋆ Johanna Björklund for being the best supervisor there is and for teaching me that failing is okay (as long as there is a party afterwards).
- ⋆ my co-supervisor Frank Drewes for being meticulous at all times and for always having a bad joke at hand.
- ⋆ my reference person Lars Karlsson for making sure everything is in order.
- ⋆ Andreas Maletti for hosting a research visit in Stuttgart, providing machine translation data and answering my questions.
- ⋆ Martin Berglund for never denying me thesis formatting help (and for always bringing tricky problems).
- ⋆ Adam Dahlgren for travelling with me, making sure that we arrived at CIAA 2018 on time, and for being my colleague and friend.
- ⋆ Henrik, Niklas, Petter and Suna the role-model for lunch (and dinner) company, seminars and problem-solving sessions.
- ⋆ Lili for laughing at my little dances in the fika room and for repeatedly answering my curious questions about her food with "it is a long story".
- ⋆ the rest of my colleagues at the department for helping me with all sorts of things, but most importantly for shared laughs.
- ⋆ my family – your support means everything.
- ⋆ Gustav, Lina, Matilda, Martin and William for being my second family.
- ⋆ Micke for all the much needed love.

Finally, I want to dedicate his thesis to my grandparents (and for their sake, I will do so in Swedish). Till farmor Vera som lärde mig att göra grimaser, till farfar Åke som visade mig hur man löser korsord, och till mormor Margit som lärde mig att inte bitas med orden "Stopp stopp, var och en biter sig själv!"

This project was funded by Vetenskapsrådet, DNR 2012-04555.

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 September 2019.

Paper IV Johanna Björklund, Frank Drewes and Anna Jonsson. Experimental Best Trees. Under construction.

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. CHRGs but restricted. Under construction.

List of Contributions

NP

Here the contributions of the thesis author to the papers are stated. Naturally, the ~~contributions include reading~~, commenting on and approving the papers.

author has read, commented on, and accepted all papers.

Paper I ~~Programming of~~ and experiments on software.

Implementation of algorithms, experiments

Paper II Performing, describing and ~~giving~~ the results of the experiments.

reporting on

Paper III Participating in ~~the joint~~ development of the algorithm, and ~~writing~~ the complexity proof.

proving the time-complexity

Paper IV Implementing the algorithm and performing, describing and giving the results of the experiments. Describing the competing algorithm and translating our algorithm to the hypergraph setting.

Paper V Writing of first draft (which we then developed together).

Paper VI TBD

- Separate experiments and experiment design?

Contents

1 Introduction **1**

Part One: The *N*-Best Problem **5**

2 Theoretical Foundation **7**
2.1 Weights and Semirings 7
2.2 Structures and Devices 9
2.3 General Problem Statement 13

3 Related Work **15**
3.1 The Best Runs Problem 16
3.2 The Best Trees Problem 19
3.3 Applications 22

4 Contributions **25**

Part Two: Semantic Graphs **27**

5 Introduction **29**

References **33**

Paper I **37**

Paper II **59**

Paper III **73**

Chapter 1
Introduction

One of the main tasks in the field of natural language processing is to

To understand a computer, it needs to communicate in natural language.

In natural language processing (NLP), an important task is to analyse sentences. Sometimes, the purpose of the analysis is to interpret the sentence, and sometimes it is to learn from the sentence.

The sentences are sequences or *strings* of words.

Natural (human) language is ambiguous.

Computers need to process natural language.

An ambiguous example sentence is "They pet the dog with the stick." Does the petted dog have a stick or do they pet the dog using a stick?

The first interpretation is that the stick was used for petting the dog, and the corresponding analysis can be found in Figure 1.1. The second interpretation, that the dog has the stick, is portrayed in Figure 1.2.

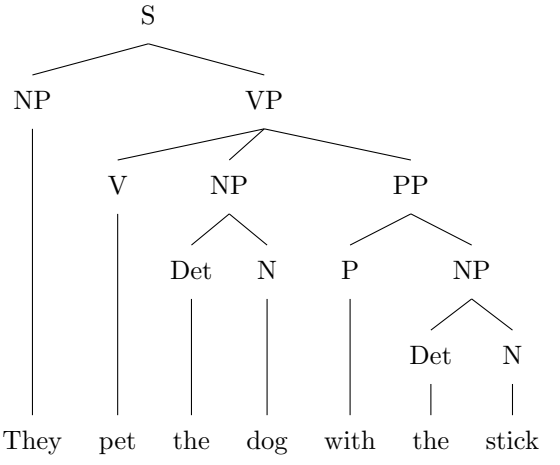


Figure 1.1: Analysis of "They pet the dog with the stick" representing the interpretation that the stick is used for petting the dog.

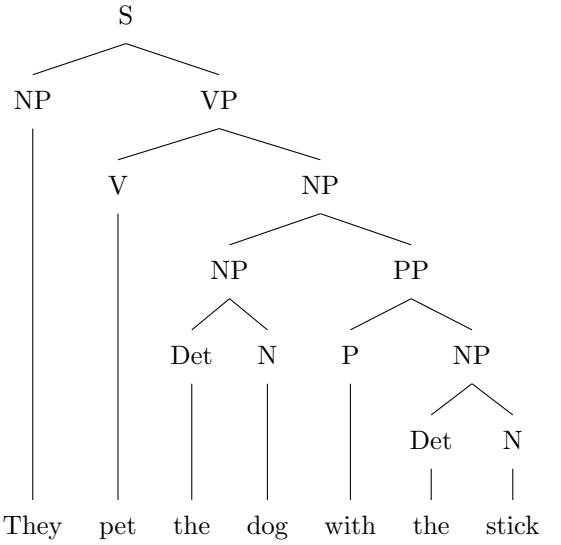


Figure 1.2: Analysis of "They pet the dog with the stick" representing the interpretation that the petted dog has a stick.

These analyses have the form of *syntax trees* in which every *node* has a label; the actual words of the sentence label the *leaves* of the tree, whereas the other node labels are syntactic roles. The structure of the tree given by the *edges* (lines in the figures) marks how these roles work together: First, each word is assigned its syntactic role; N marks a noun, V marks a verb, P marks a preposition, and Det marks a determiner. 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). (The word "They" is an exception: since it is not paired with anything to become part of a noun phrase, we skip the noun label.)

In our case, each of the syntax trees represents one interpretation or *semantics*, but there can be syntax trees that represent the same semantics.

Which interpretation is the most probable one? If the sentence was found in a book about a bunch of mean children, then the first interpretation might be the most likely, but if the sentence came from an interview with an old person living alone with their dog, then the second one is more likely. Given the context of the sentence, we can figure out which interpretation is the most likely one. This is exactly what machine learning is for: teaching a model to value sentence analyses based on their contexts. Doing this is a thesis on its own, which is why in this thesis, we assume that we already have this model, trained on some data, that we ... .

Machine learning can be done using both formal methods and deep learning (e.g. neural networks): it all boils down to assigning weights to various components of the models.

Compact representations of tree languages are given by tree automata and tree grammars.

Both automata and grammars consist of rule sets, but an automaton takes an input, apply rules to it and spits out a weight, whereas the rules of a grammar builds a tree in the language. For automata, we say that a tree is in the tree language represented by the automaton, if and only if there is a *run* of of tree in the automaton. The corresponding term for grammar is *derivation*.

For *deterministic* automata or grammar, each run or derivation corresponds to a single tree. If we instead have *nondeterminism*, there are several runs or derivations for a single tree. In applications, it is common to want to extract the top hypotheses ... . Here, we refer to this as *the N-best trees problem*. However, due to nondeterminism, the N-best trees problem has turned out to be . The solution has been to solve the easier N-best *runs (or derivations)* problem which does not guarantee that the output list of N trees is free from duplicates. The N-best problem makes up Part One of this thesis.

Sometimes we need more than a syntax tree to distinguish between different semantics. For example, the sentence "Anna tells the group to ignore them" says nothing about who "them" refers to; it could be Anna who wishes to be ignored, it could be a person or another group of people who are to be ignored. What is worse: forming a syntax tree does not help us here, all we get from that 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.

A common solution is to add dependencies to syntax trees, turning them into easy-to-handle graphs.

Representing semantics using graphs and graph languages is not as well-researched as using trees for syntactic representations of natural language.

Part One

The *N*-Best Problem

Chapter 2

Theoretical Foundation

The N-best problem is ~~essentially about~~ extracting the N best hypotheses from a hypothesis space. As we have seen, common hypotheses for language processing tasks ~~have~~ 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 some criterion that decides if a structure is better than another, and this is where weights and semirings enter the picture.

2.1 Weights and Semirings

When developing an algorithm that operates on a weighted structure or device, one has to make sure that the algorithm uses weight operations that enable a correctness proof. A *semiring* is, informally, a set of logical entities (weights) and operations defined on them. Proving correctness of an algorithm heavily depends on being able to prove that it ~~works for~~ 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, Mohri's algorithm is less efficient than Dijkstra's.

Below follows the formal semiring definition, starting with ~~a help~~ definition.

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 $a \in \mathbb{K}$, and 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 $a \oplus b = c$ where c is either a or 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 TRUE $\vee$ TRUE = TRUE and FALSE $\vee$ FALSE = 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 $\mathcal{T} = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$.

what is prev. mentioned ?

This semiring is the exact one used in Dijkstra's algorithm (as previously mentioned), 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 that are relevant to the papers on the N-best problem that are brought up in this thesis.

Let S be a set, and let $|S|$ denote the cardinality of S. 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. The length of a string s is in accordance with the set notation given by $|s|$. A *tree* labelled over S is a string t defined recursively as follows:

- $t = s$ for $s \in S$ is a tree consisting of a single *node* labelled by s, and
- $t = s[t_1, \ldots, t_k]$ where $s \in S$ and t_i are trees over S for $i = 1, \ldots, k$ is a tree. (The square brackets and commas in t are simply delimiters that are not part of S.)

We say that t is *rooted* in s and that $t_1, \ldots, t_k$ are *subtrees* of t. Furthermore, s is the *parent* of the roots of $t_1, \ldots, t_k$, and these are in their turn *children* of s. A *leaf* is a node without any children. For convenience, we identify a node by its label when discussing specific trees.

Now that we are done defining the structure types for which we are discussing the N-best problem, namely strings and trees, we move on to defining devices operating 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 set of nonterminals in 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* or *generated by* G. Formally, a derivation is a tree in which the nodes are production rules.

Adding weights to a cfg can be done in various ways. Here, we assign a weight w 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 sum up the weights of all rules in d, and the weight of a string s in $\mathcal{L}(G)$ is the minimal weight over all of the derivations in G that result in s. In other words, we use the tropical semiring to make cfgs weighted. For the next device, the use of the same semiring is part of its definition:

Definition 2.5 A *weighted finite string automaton (wsa)* over the tropical semiring $\mathcal{T} = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$ 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*,
- $\star$ $\delta \colon Q \times \Sigma \times Q \to \mathbb{R}_+ \cup \{\infty\}$ 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*.

For simplicity, we let δ denote the set of transitions, and 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 $w \neq \infty$; i.e., transitions that have non-infinite weight. Thus, the absence of a transition from the stored part of δ should be interpreted as it having weight ∞.)

Given an input string or *word* $s \in \Sigma^*$, we can run a wsa M on s with $|s| = n$ as follows. For $0 \leq i \leq n$, let $w, w_i \in \mathbb{R}_+ \cup \{\infty\}$, $q, q', q_i \in Q$, $\sigma, \sigma_i \in \Sigma$ and $x \in \Sigma^*$ such that σx is a suffix of the input string s and $\sigma_1 \dots \sigma_n = s$. M always starts in its initial state q_0, and we process s from left to right. A *configuration* of M is a pair $(q, \sigma x)$ that ~~marks~~ denotes that we are currently in state q and have yet to process σx. The initial configuration of M is (q_0, s). When M is in the configuration $(q, \sigma x)$, it can apply any transition $(q, \sigma) \xrightarrow{w} q'$ and thereby end up in the configuration (q', x). (Note that we can always apply the transition $(q, \sigma) \xrightarrow{\infty} q'$ in absence of a transition with a smaller weight.) A *run* ρ on s is a sequence of $n+1$ configurations $\rho(s) = (q_0, s)(q_1, \sigma_2 \cdots \sigma_n) \dots (q_{n-1}, \sigma_n)(q_n, \lambda)$ for which $q_n \in Q_f$. Let $(q_{i-1}, \sigma_i) \xrightarrow{w_i} q_i$ be the transition that we applied to go from q_{i-1} to q_i. Then, the *weight of a run* $\rho(s)$ is given by

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

and the *weight of s assigned by M* is

$$M(s) = \min_{\rho} \left(\mathrm{wt}(\rho(s))\right) \ .$$

That is, we sum up the weights of the used transitions to get the weight of a run and take the minimum weight over all runs on s to get the weight of s in M (according to $\mathcal{T}$).

Note that the cfg and the wsa are not equally powerful: wsa weights *regular* string languages whereas cfg weights *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.

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

Figure 2.1: Example wta.

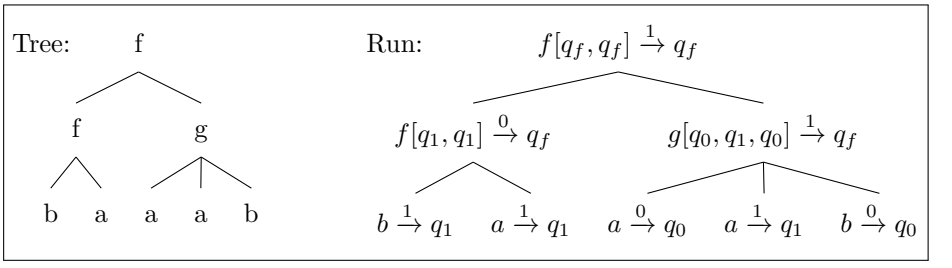


Figure 2.2: Left: A graphic representation of the tree $f[f[b, a], g[a, a, b]]$. Right: An example run for the wta given in Figure 2.1 on the tree to the left.

Definition 2.6 A *weighted tree automaton (wta)* over the tropical semiring $\mathcal{T} = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$ is a quadruple $M = (Q, \Sigma, \delta, Q_f)$ where

- $\Sigma = \biguplus_{k \in \mathbb{N}} \Sigma_k$ is a finite set of symbols with $\text{rank}(\sigma) = k$ for $\sigma \in \Sigma_k$ called a *ranked alphabet*,
- $Q \subset \Sigma_0$ is the set of *states*,
- δ is a finite set of transition rules on the form $\sigma[q_1, \ldots, q_k] \xrightarrow{w} q$ with $\sigma \in \Sigma_k \setminus Q$, $q, q_1, \ldots, q_k \in Q$ and $w \in \mathbb{R}_+ \cup \{\infty\}$, and
- $Q_f \subseteq Q$ is the set of *final states*.

To make the next definition simpler, we turn δ into a ranked alphabet by setting $\text{rank}(\sigma[q_1, \ldots, q_k] \xrightarrow{w} q) = \text{rank}(\sigma)$. A *run* ρ of M on a tree t labelled over Σ is a relabelling of t with symbols from δ, and we denote this relabelling by $\rho(t)$. The relabelling is done such that for any subtree $\sigma[t_1, \ldots, t_k]$ of t, $\sigma \in \Sigma$ can be replaced by any transition rule $\sigma[q_1, \ldots, q_k] \xrightarrow{w} q$ in δ for which that the roots of $t_1, \ldots, t_k$ can be replaced by $q_1, \ldots, q_k$, respectively. If the root of ρ has its right-hand side in Q_f, then ρ is an accepting run of M on t. An example run of the automaton in Figure 2.1 can be seen in Figure 2.2.

The *weight of a run* ρ is denoted by $\text{wt}(\rho)$ and given by the sum of the weights of the transition rules labelling ρ. The weight of the run in Figure 2.2 is 5, and since $q_f \in Q_f$, it is accepting. A tree t can have several runs, and we define *the weight of the tree* t *in* M, $M(t)$, as the smallest weight over all runs of M on t. For the example in Figure 2.2, this weight is 4.

A wta represents a weighted regular tree language, and wta is thus equivalent to weighted regular tree grammar (wrtg, see e.g. [MK06]). Furthermore, both wta and wrtg can be represented as *hypergraphs*.

Definition 2.7 A *weighted hypergraph over a semiring $\mathcal{K}$ 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:

- 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*.
- $\mathrm{wt}_G\colon E \to \mathcal{K}$ is the *hyperedge weights*.

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 2.8 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

$$\mathrm{wt}(d) = \mathrm{wt}_G(e) + \sum_{i=1}^{|e|} \mathrm{wt}(d_i)$$

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 $\mathrm{result}(d) = \mathrm{lab}_E(e)[\mathrm{result}(d_1), \ldots, \mathrm{result}(d_{|e|})]$.

The *weighted tree language defined by a node set $V_0 \subseteq V$ in G* is a mapping from trees t rooted in nodes of V_0 to the minimal weight $\mathrm{wt}(d)$ over all derivations d for which $\mathrm{result}(d) = t$ and $\mathrm{root}(d) \in V_0$ in G.

For the purpose of showing that wta and hypergraphs are equivalent devices, let $M = (Q, \Sigma, \delta, Q_f)$ be a wta. We construct a hypergraph $G = (V, E, \mathrm{att}, \mathrm{lab}_E, \mathrm{wt}_G)$ in the following way: For every state $q \in Q$, add a vertex v_q to V, and for every transition rule $r = (\sigma[q_1, \ldots, q_k] \xrightarrow{w} q)$, add a hyperedge e_r to E with $\mathrm{att}(e_r) = v_q v_{q_1} \ldots v_{q_k}$, $\mathrm{lab}_E(e_r) = \sigma$, $\mathrm{rank}(\sigma) = k$ and $\mathrm{wt}_G(e_r) = w$. (The construction of a wta given a hypergraph is done by reversing the above process.)

Theorem 2.9 *The set of runs of M is equivalent to the derivation set of G.*

Show weight of t and $\varphi(t)$ to be the same.

Proof. A run ρ of M on a tree t can be turned into a derivation d of G by exchanging each transition rule r labelling a node in ρ for its corresponding hyperedge e_r in G. Since r and e_r have both the same weight and the same label, we have $\mathrm{wt}(d) = \mathrm{wt}(\rho(t))$, $\mathrm{result}(d) = t$.

Conversely, let d be any derivation in G. Replacing every label e_r in d with the to e_r corresponding $r = (\sigma[q_1, \ldots, q_k] \xrightarrow{w} q)$ yields a run $\rho(\mathrm{result}(d))$ in M with $\mathrm{wt}(\rho(\mathrm{result}(d))) = \mathrm{wt}(d)$. Also works since $\oplus = \min$ □

Theorem 2.10 *Let the weighted tree language of G be given by the node set V_0 consisting of all nodes v_q for which the corresponding state q in M is in Q_f. Then, M and G are language equivalent.*

Proof. From Theorem 2.9, we know that a run ρ in M has exactly one equivalent derivation d in G.* Since V_0 is given by the set of final states in M, the derivations ending in nodes of V_0 correspond exactly to the set of final runs of M. Thus, the minimal weight over all derivations d for which $\mathrm{result}(d) = t$ is the same as the weight given by $M(t)$. □

* The proof says: Not 1-to-1!



2.3 General Problem Statement

For devices that define weighted formal languages, the *N-best ~~element~~ problem* is formulated as:

> Given a device A and a positive integer N, return 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 $\mathcal{L} \setminus \{x_1, \ldots x_{i-1}\}$
> 3. there is no element x distinct from x_i in ~~the language~~ defined by A with smaller weight than x_i for all $i \in \{1, \ldots, N\}$.

This is the min version, but the optimisation criterion depends on the additive operation of the semiring for which A is defined. If we reformulate the first and third requirements to instead demand that the elements have distinct runs (or derivations), we have the *N-best runs (or derivations) problem.* This is an easier problem since we do not care whether the actual elements are the same, the only thing we do care about is that their processing within A is different. In Chapter 3 we will see specified instances of the above problem statement.

Chapter 3

Related Work

To start our review of research on the N-best problem, we go back to 1977 ~~in which~~ Knuth [Knu77] presents an algorithm that finds the smallest derivation for a 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, ~~uses another way~~: 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 $x_1, \ldots, x_{|\mathrm{nt}(\alpha)|}$ 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 constant-valued functions for all production rules, as we do here, certainly fulfils the superiority criterion. This is naturally less general, but sufficient for the purpose of reviewing the algorithm in this setting.

Algorithm 1 shows the pseudocode for Knuth's algorithm. The algorithm is based on the observation that we can, for every nonterminal A that occurs in productions $r = (A \xrightarrow{w} \alpha)$ with $\mathrm{nt}(\alpha) = \emptyset$, find at least one string s such that $A \Rightarrow s$. Furthermore, at least one of these strings is optimally derived from its corresponding nonterminal, namely the string that minimises the rule weight over all production rules with $\mathrm{nt}(\alpha) = \emptyset$ in R. Say that the minimal weight is w' and that the production rule giving rise to w' has A as its left-hand side nonterminal. Then, w' is saved as a result in $res[A]$ (given that the nonterminals can be internally represented as integers, we allow indexing arrays using nonterminals). To mark that the weight of A's smallest derivation is now defined, A is added to the formerly empty set D. Now knowing the smallest derivation for the nonterminal in D, we are no longer restricted to rules that have only terminals in their right-hand side, but these can now also contain the nonterminal in D since its smallest weight is defined. 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.

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

Algorithm 1 An algorithm that given a cfg $G = (N, \Sigma, S, R)$ with ~~superior~~ weights 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
```

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 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 smallest 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 split up into two groups: those that find the N best runs (or derivations) (Section 3.1) and those that find the N best structures (Section 3.2). Finally, we review some applications of primarily the N-best trees problem (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. As proven in Theorem 2.10, 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 monotone in the sense that increasing one of the tail weights does not decrease the head's weight. 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 output a sequence of N derivations $d_1, d_2, \ldots, d_N$ such that for $i, j \in \{1, \ldots, N\}$ we have

1. $\mathrm{root}(d_i) = v_0$, and
2. $\mathrm{wt}(d_i)$ is minimal in $\{d \mid \mathrm{root}(d) = v_0\} \setminus \{d_1, \ldots, d_{i-1}\}$.

Write in italics if the target node is now defined.

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 $(N-1)$-best derivation. The best derivations for a node v is 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 ~~asks for~~ computes it recursively. The best out of the candidates for v_0 is then picked by extracting the minimal element from the heap $cands[v_0]$, which unlocks new ~~possible~~ 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), given that 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.

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.

We present the reasoning behind each of the generalisations without repeating the Huang and Chiang algorithm itself.

The first generalisation is the labelling of the hypergraphs which is motivated by it allowing modelling of tree-based language models such as weighted

the need to represent

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 $G = (V, E, \mathrm{att}, \mathrm{lab}_{\mathrm{E}}, \mathrm{wt}_G)$.

```
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
```

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 recursion.

Third, there is the generalisation to structured weight domains such as vectors over $\mathbb{R}_+$. This is done by defining a linear pre-order $\precsim$ on the derivations' weights, and by extension, on the derivations themselves. Let $c = e[c_1, \ldots, c_{|e|}]$ and $d = e[d_1, \ldots, d_{|e|}]$ be any two derivations. A *linear pre-order* $\precsim$ is a binary relation that is reflexive, transitive and linear, the latter meaning that if not $c \precsim d$, then $d \precsim c$ is. 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 the Knuth algorithm for computing the 1-best derivations, 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 anti-symmetry) which ensures that structured weight domains can be used.

Finally we have the explicit handling of cycles in the hypergraph. Huang and Chiang conjecture that their second phase (that is, 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.

Since the first three generalisations do not alter the Huang and Chiang algorithm, the main contribution of this paper 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 strings problem for 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.

Definition 3.1 The *N-best-strings problem for a wsa $M = (\Sigma, Q, \delta, q_0, Q_f)$* is the problem of finding N distinct strings $w_1, \ldots, w_N \in \Sigma^*$ such that there is no $w' \in \Sigma^*$ for which $M(w') \leq M(w)$ for $w \in \{w_1, \ldots, w_i\}$ and $i \in \{1, \ldots, N\}$.

The algorithm of Mohri and Riley is based on an on-the-fly determinisation of M, i.e., they create the equivalent wsa $M' = (\Sigma, Q', \delta', q'_0, Q'_f)$ with a single

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

```
1:  L_O ← 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_O, (q, w, c))
8:          if List-Size(L_O) = N then
9:              return L_O
10:         end if
11:     end if
12:     if List-Size(L_O) ≤ 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
```

run per input string such that $M'(w) = M(w)$ for all $w \in \Sigma^*$. Not all weighted automata are determinisable, but since only the portion of M' that is used in the N best computation needs to be created, and since all acyclic automata are determinisable, this does not pose a problem here.

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 $(q'_a, \sigma) \xrightarrow{t} q'_b$ where

- $q'_a = \{(q_i, r_i) \mid 0 \leq i \leq n\}$ is the current state,
- $t = \min\{r_i + t_i \mid (q_i, r_i) \in q'_a,\ (q_i, \sigma) \xrightarrow{t_i} q'_i \in \delta\}$ is the transition weight,
- and $q'_b = \{(q'_i, r'_i) \mid (q_i, r_i) \in q'_a,\ (q_i, \sigma) \xrightarrow{t_i} q'_i \in \delta,\ r'_i = \min(r_i + t_i - t)\}$ 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 $q'_i \in Q_f$.

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 we have seen so far. The format of the runs is (q, w, c) where q is the current state, w is the string that has been processed this far, and c is the cost of the partial run. A list L_O 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 still has elements, 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_O 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 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.

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 [MR02] to the tree domain.

To facilitate the description of the algorithm, the input wta $M = (Q, \Sigma, \delta, Q_f)$ is split into two wtas given a state $q \in Q$: $M^q = (Q, \Sigma, \delta, \{q\})$ that takes us to q, and $M_q = (Q, \Sigma \cup \{\Box\}, \delta \cup \{\Box \xrightarrow{0} q\}, Q_f)$ that takes us from q to a final state. 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 the 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 [Knu77]. 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 for 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 with labels 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 such 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 Eppstein algorithm [Epp98] for finding the N shortest paths in a graph is used. The construction of the input graph to Eppstein's algorithm is included in line 15 (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 ← ∅                                                          ▷ seen trees
2:  K ← QUEUE-EMPTY()                                              ▷ priority queue
3:  counter ← 0                                                    ▷ counter for outputted trees
4:  for σ ∈ Σ_0 do                                                 ▷ initialisation
5:      prune(T, QUEUE-ENQUEUE(K, σ, min_{q∈Q}{w + M^q(c_q) | (σ →^w q) ∈ R}))
6:  end for
7:  while counter < N ∧ ¬QUEUE-IS-EMPTY(K) do
8:      t ← QUEUE-DEQUEUE(K)
9:      T ← T ∪ {t}
10:     if M(t) = min_{q∈Q}{M_q(t) + M^q(c_q)} then
11:         output(t)
12:         counter ← counter + 1
13:     end if
14:     for r = (σ[q_1, ..., q_k] →^w q) do
15:         E ← Eppstein(r, t, N)      ▷ the N best instantiations of r that use t
16:         for t' ∈ E do
17:             M(t') ← 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 finite hypergraphs as 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 ~~are one and the same~~. Otherwise, there can be up to an exponential number of derivations for one tree, which illustrates why the N-best trees problem is a more difficult one.

3.3 Applications

Socher et al. [SBMN13] introduce the compositional vector grammar (cvg) that is built on top of a standard probabilistic context-free grammar (pcfg) model, ~~and in addition, it uses~~ 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 here we see 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 ~~investigation~~ consists ~~of~~ 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 ~~look at a number~~ of possible replacements for fsts in the form of various tree transducers. Their conclusion is that there are replacement tree transducers that could be used 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 mention 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.

Their motivation for developing this new method is that the N-best lists algorithms that are available are ~~not good enough~~ 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

Part Two

Semantic Graphs

Chapter 5
Introduction

Humans communicate in *natural language* such as English or Swedish. When humans talk to each other, it is acceptable for us to make grammatical mistakes since we will likely still understand each other. In other words: the meaning of what we are saying – the *semantics*[1] – will still come across, regardless of the quality of the *syntax* (specific formulation). A computer, on the other hand, would deem the sentence useless unless the syntax is perfectly correct; it can only interpret language that follows pre-specified rules strictly.

Might not be true for word embeddings.

The aim of the field of natural language processing is to make it possible for computers to communicate in and help humans with natural language (here, we limit ourselves to its written form). To do that, the computer must be able to extract the semantics even if the input sentence is slightly syntactically incorrect. Then it has to process the semantics to compute the desired result – also in the form of semantics. Finally, it should transform the result into a syntactically passable sentence and output it to the user. Thus, we need to handle both syntax and semantics when developing natural language processing tools.

We can still argue that humans are more robust.

As a starting point, we know that the strength of computers lies in their ability to store data and carry out formal directions (e.g. performing computations). We use this information to build language models for the computer. Such models normally contain the following two parts:

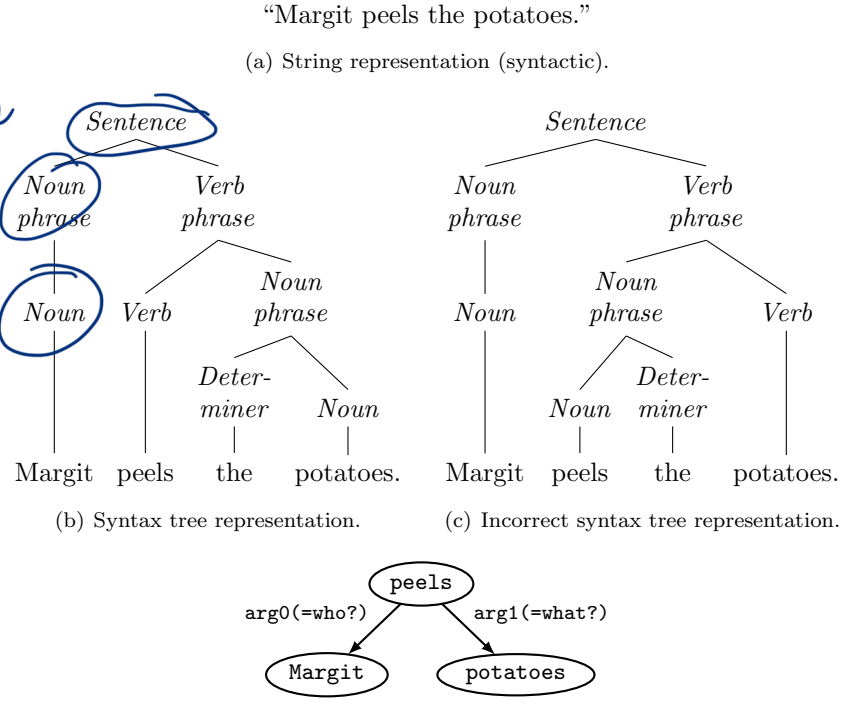
(1) a structure in which we can express the syntax *or* semantics of pieces of language (usually sentences), and

dependency parses are in-between (?)

(2) a device that scores syntactic *or* semantic structures based on how well they follow a given set of rules, and outputs the ones that follow them well enough.

Let us start by looking at (1). The structure can for example simply be a string as in Figure 5.1(a) – in this case, it coincides with a sentence since a sentence is nothing but a string of natural language. It can also be a tree as seen in Figure 5.1(b) which contains more information about the sentence and is therefore more often used in natural language processing. A third option is to use a graph where the nodes hold the main concepts of the sentence and the edges are the relations given by the sentence (see Figure 5.1(d) for an example).

[1] We include pragmatics in this term since it is not helpful to separate them in this case.



(a) String representation (syntactic).

(b) Syntax tree representation.

(c) Incorrect syntax tree representation.

(d) Semantic graph representation.

Figure 5.1: Different representations of the same sentence.

The first two structures – strings and trees – are dependent on the syntax of the sentence and therefore said to be *syntactic representations*. In our example, the semantics might as well be expressed as "The potatoes are peeled by Margit" which would result in changed string and tree representations. The graph representation, however, would remain the same; it is therefore considered a *semantic representation*. When translating a sentence from one natural language to one or several others, it is valuable to have a structure that can represent the semantics regardless of syntax since the syntax changes between languages. When time comes to present the result to the user, however, a syntactic representation is needed. The ultimate goal of this research is to extract and process semantic structures. Important tasks are to use these structures efficiently in computations and to judge their quality. This is where (2) comes in.

Consider the tree in Figure 5.1(c). The sentence represented by the tree uses 'potatoes' as a verb and 'peels' as a noun, making it mean "Margit potatoes the peels". To humans, it is obvious that this is an example of incorrect natural language. How can we teach computers to differentiate between structures that represent correct and structures that represent incorrect natural language? Since computers are rule-followers, we use devices based on rules that

express desired properties of the structures. The devices considered here are *grammars* and *automata*. A grammar is a rule set that *generates* a structure. An automaton is defined by transition rules that *accept* a structure. Apart from using different mechanics, automata and grammars are interchangeable. The set of structures that can be generated by a grammar or accepted by an automaton is called a *formal language*, or *language* for short.

These devices are practical in the sense that they remove the necessity of saving entire languages in memory, allowing for infinitely large languages – and also for efficient processing. We can instead, given a structure, check whether or not the device generates or accepts it. This check is referred to as *parsing*. Unfortunately, depending on the expressiveness of the device, parsing can be a very difficult problem. An expressive device can handle more complex structures and rules, but the more expressive a device is, the harder is its parsing problem.

Both grammars and automata can be *weighted* in which case they provide an error score for the structure. Using the error score, we can compare different structures and thereby obtain a quality measure. For example, if we receive a set of translations in the form of an automaton, then we can pick the translation with the lowest weight. In this thesis, we only work with these two device types.

When we have both (1) and (2), we can model natural language using a formal language. Using the rules of the device, we can try to mimic the actual grammar of the natural language. The problem is that it is hard to find a device that is expressive enough whose parsing problem is solvable using a computer. Countless combinations of structures, devices and device expressibility have been explored, with varying results and usage areas. Yet another possibility is explored in this thesis.

Finding a good structure and device combination is not enough, however. We must also be able to process language in all sorts of ways: automatically changing between natural language representations, translating natural language in a computer (machine translation), extracting language from images and videos and put it into language representations, analysing natural language correctness and much more. Here, we focus on a very specific problem which is motivated by an application in machine translation.

One way to implement machine translation is to use *cascade evaluation* (see Figure 5.2). Cascade evaluation is when a problem can be solved in several steps where the output of one step is the input of the next. When the intermediate

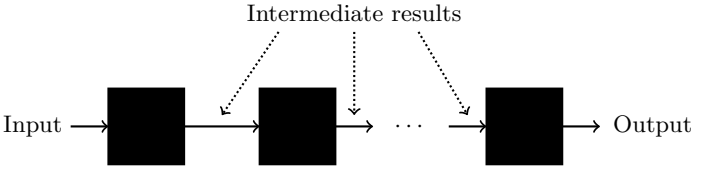


Figure 5.2: Cascade evaluation.

results are devices that describe infinite languages, we aggregate too much data to handle. This is the case for machine translation where we get a weighted tree automaton in each step. One approach to solving this is taking the result that is given the lowest error score by the device and propagate only that to the next step. This might not give us the optimal solution to the problem, but it is a good heuristics. An even better heuristics is achieved by propagating more that one result in each step. Finding the lowest-weighted structure is an easy problem for many devices whereas finding the $N > 1$ best structures is a harder one – the latter is a central topic of this thesis.

The outline of the remainder of Part Two of this thesis follows:

References

[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.

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

[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.

[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.

[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.

[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.

[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.

[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 $pow(S)$. 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 $\pi\colon pow(S) \to pow(S)$. 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

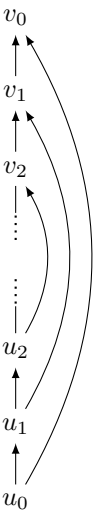


Figure 1: A graph that satisfies condition (a) but not condition (b)

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 $hull(\emptyset)$ 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 $G = (V, E, src, tar, wgt)$ such that (V, E, src, tar) is a layered graph and $wgt\colon E \to \mathcal{A}$ is its *weight assignment*. The induced *weight function* $w\colon V \cup E \to A$ determined by the following conditions:

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

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 $tar(\pi) = v$. 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`

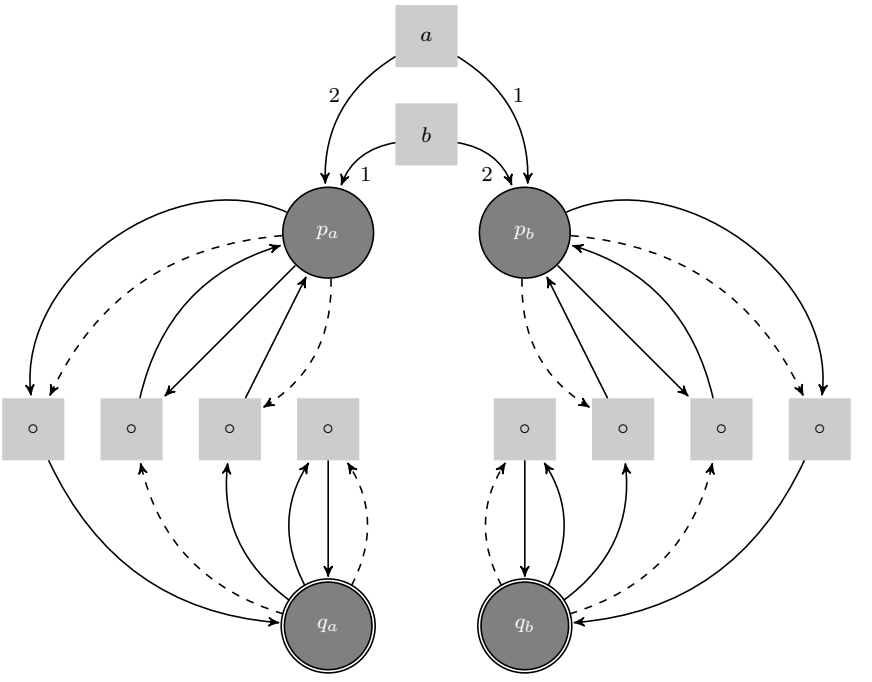


Figure 2: The input wta used in our experiments, taken from [2]. Round nodes (with double circles if final) represent states, and squares represent transitions. The consumed input symbols are shown inside the squares. Solid arcs point to the right-hand side of the transition in question and are labelled 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

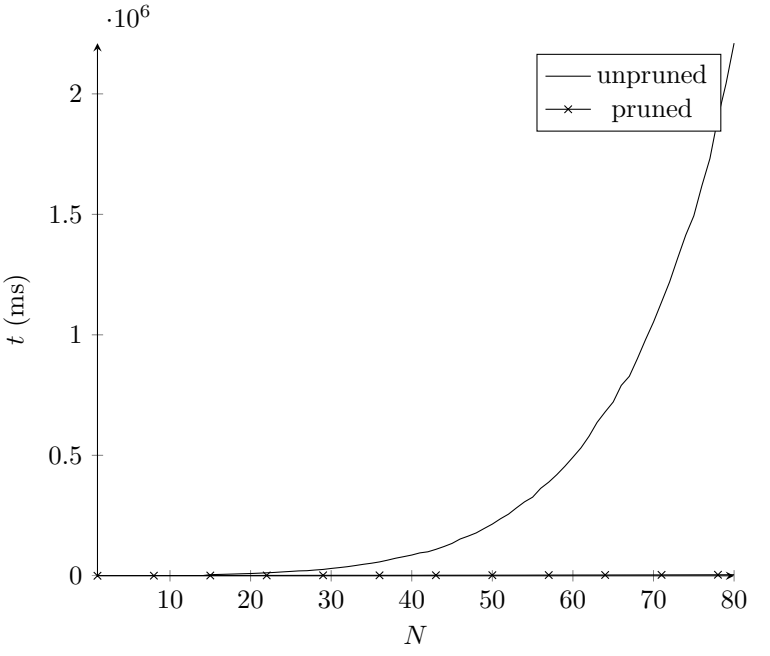


Figure 3: The running time of both versions of the algorithm applied to the wta in Fig. 2 for different values of N.

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.

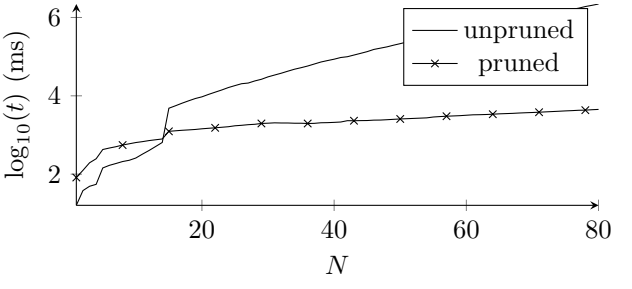


Figure 4: The running time of both versions of the algorithm applied to the wta in Fig. 2 for different values of N on a logarithmic scale.

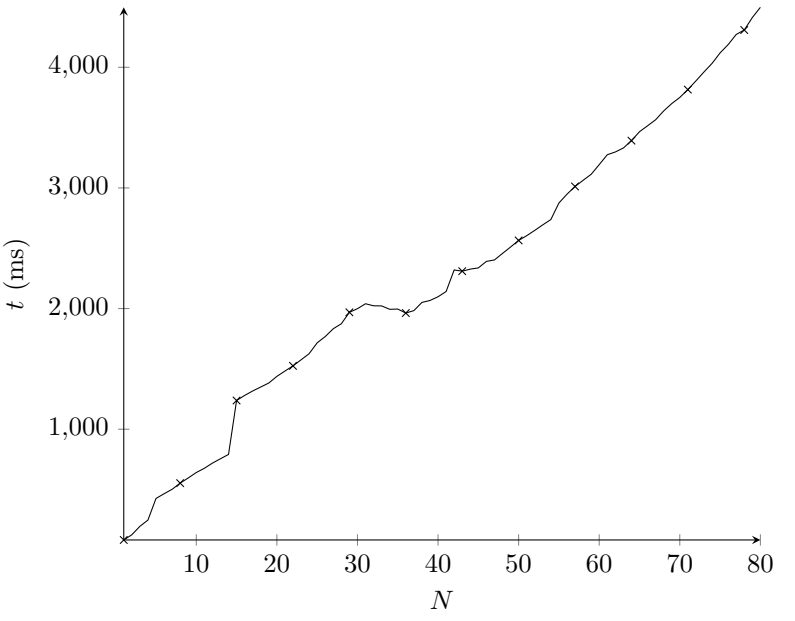


Figure 5: The running time of the pruning version of the algorithm applied to the wta in Fig. 2 for different values of N.

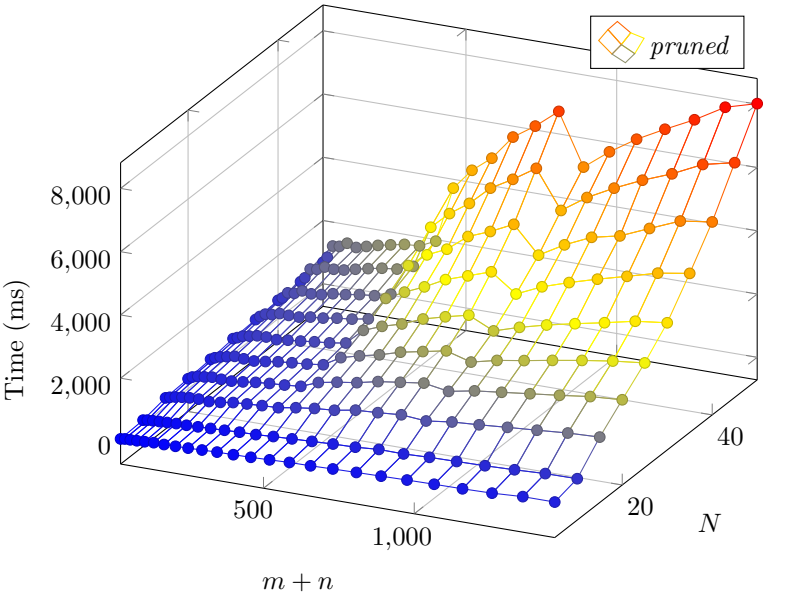


Figure 6: The running time of the pruned version of the algorithm for different values of N on different sized wtas (measured in $m + n$).

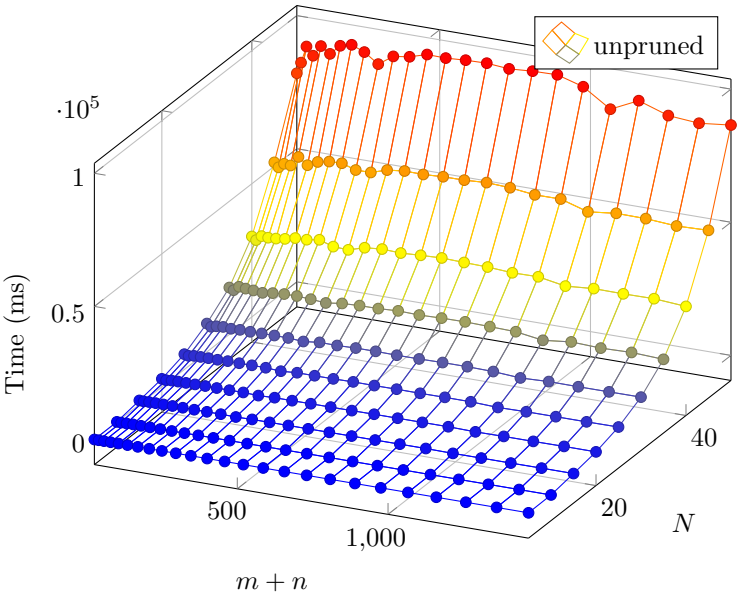


Figure 7: The running time of the unpruned version of the algorithm for different values of N on different sized wtas (measured in $m+n$).

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 $\mathit{pow}(S)$. 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 $\mathit{dom}(t)$ is a finite non-empty set that is prefix-closed and closed to the left: whenever $vi\in\mathit{dom}(t)$ for some $v\in\mathbb{N}_+^*$ and $i\in\mathbb{N}_+$, it holds that $v\in\mathit{dom}(t)$ (prefix-closedness) and $vj\in\mathit{dom}(t)$ for all $1\le j\le i$ (closedness to the left). The size of t is $|t|=|\mathit{dom}(t)|$. An element v of $\mathit{dom}(t)$ is called a *node of* t, and $|\{i\in\mathbb{N}_+\mid vi\in\mathit{dom}(t)\}|$ is the *rank of* v. The *subtree* of $t\in T_\Sigma$ 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 $f[t_1,\dots,t_k]$, 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 $\mathit{rank}(f)$. The set T_Σ of ranked trees over Σ consists of all Σ-labelled trees t in which the rank of every node $v\in\mathit{dom}(t)$ 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 $v\in\mathit{dom}(c)$ with $c(v)=\Box$. The *substitution* of another tree t into c results in the tree $c[\![t]\!]$ given by $\mathit{dom}(c[\![t]\!])=\mathit{dom}(c)\cup\{vu\mid u\in\mathit{dom}(t)\}$ and

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

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

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.

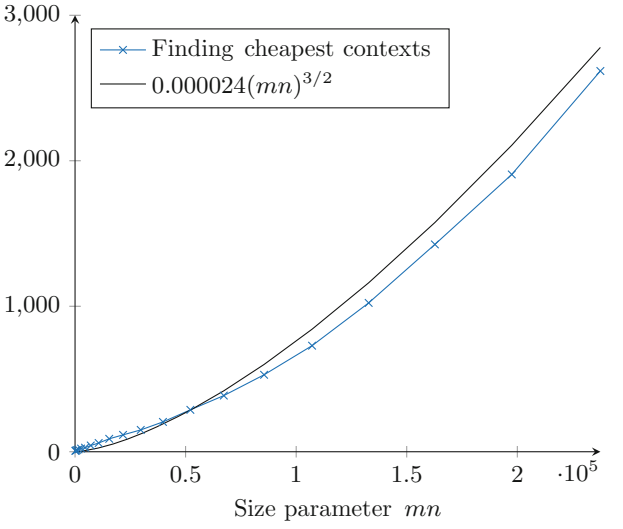
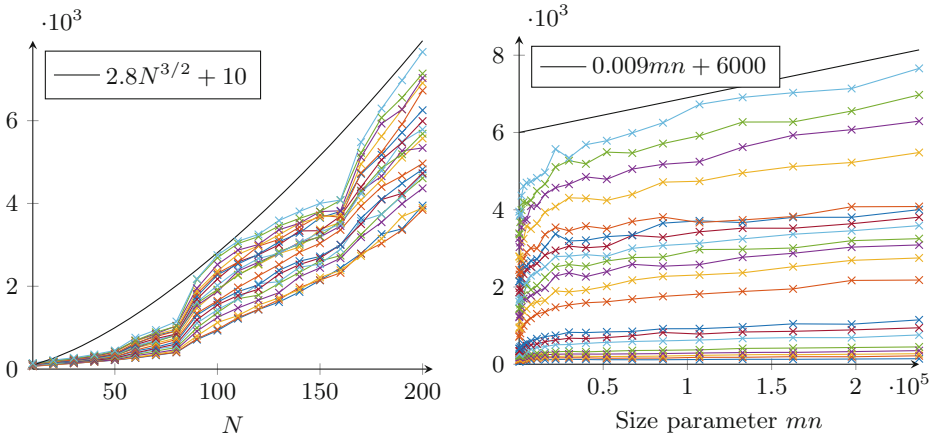


Fig. 1. Running time of finding the cheapest contexts for wtas of increasing size in the set Basic Example.

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.

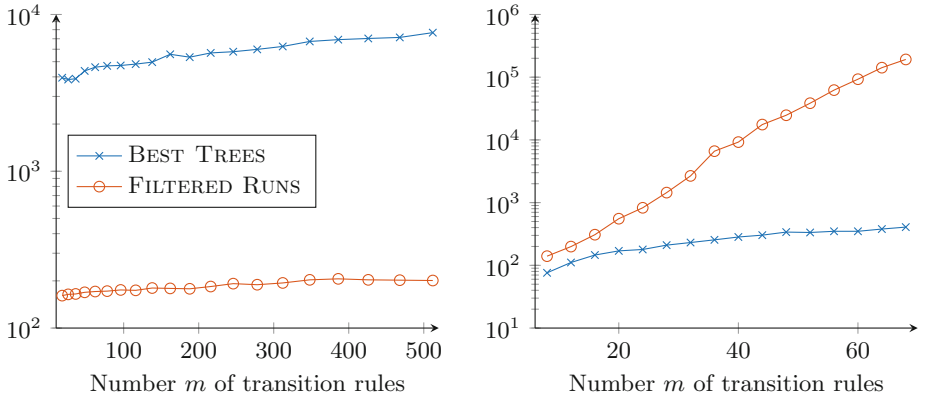
The measured running time as a function of the size of wtas is also lower than the theoretical upper bound $O((mn)^{3/2})$ in this example, namely linear.

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 $N = 200$.

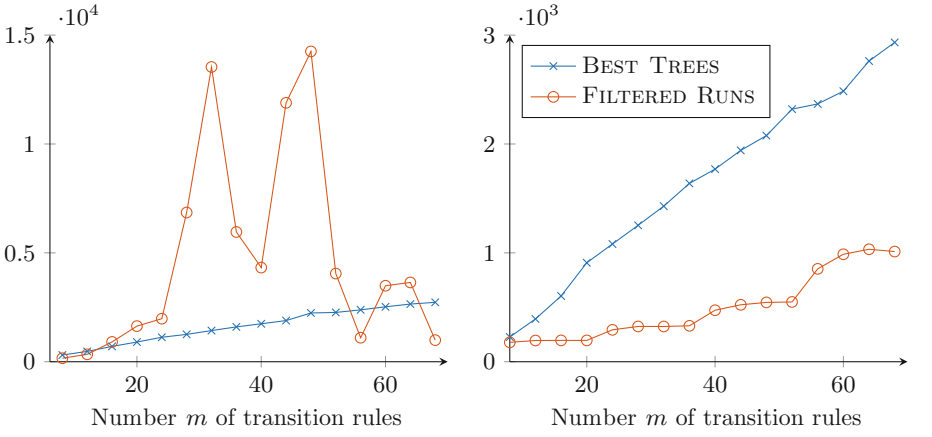
(b) Comparison of Best Trees and Filtered Runs when run on the Equal Weights wta set for $N = 30$.

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

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-

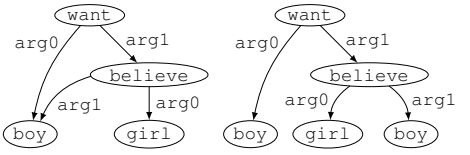


Figure 1: AMRs representing an event `want`, where the wanter is a `boy` and the wantee is the event `believe` for which the believer is a `girl` and the believee is either the formerly mentioned `boy` (left) or a different one (right).

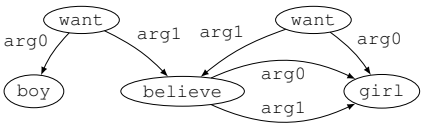


Figure 2: Another AMR.

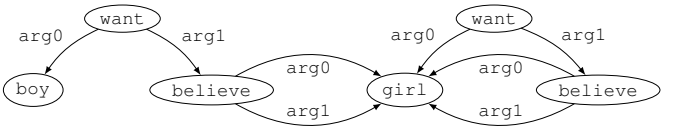


Figure 3: An AMR similar to the one in Figure 2, but with two distinct `believe` events.

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-

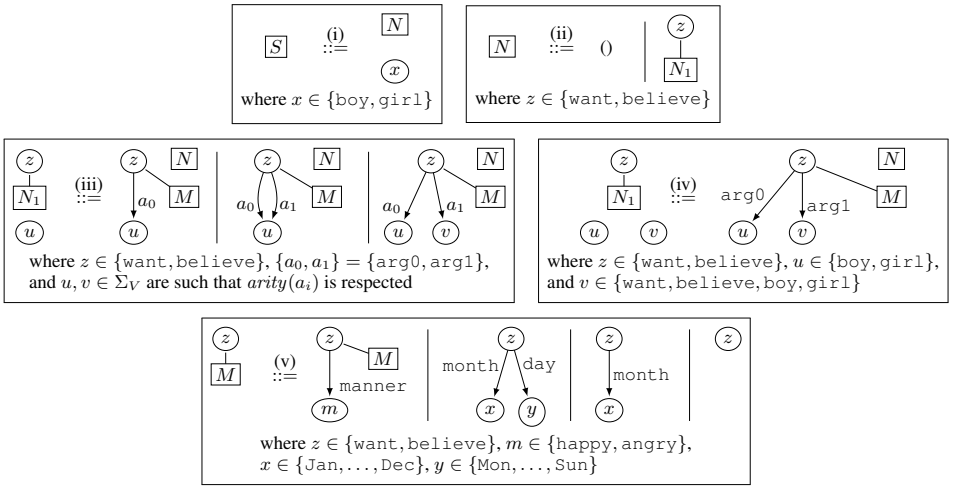


Figure 4: A boy-girl CHRG exemplifying general rule structures. Rules are named for later reference by a superscript on the operator '::='.

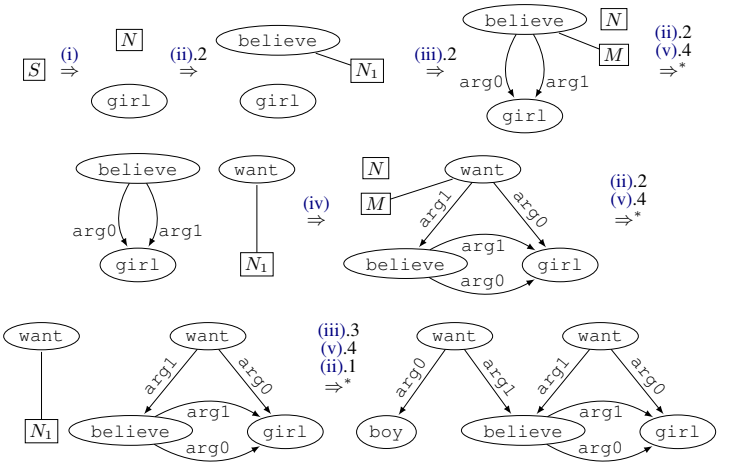


Figure 5: A derivation of an AMR using the boy-girl CHRG.

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.

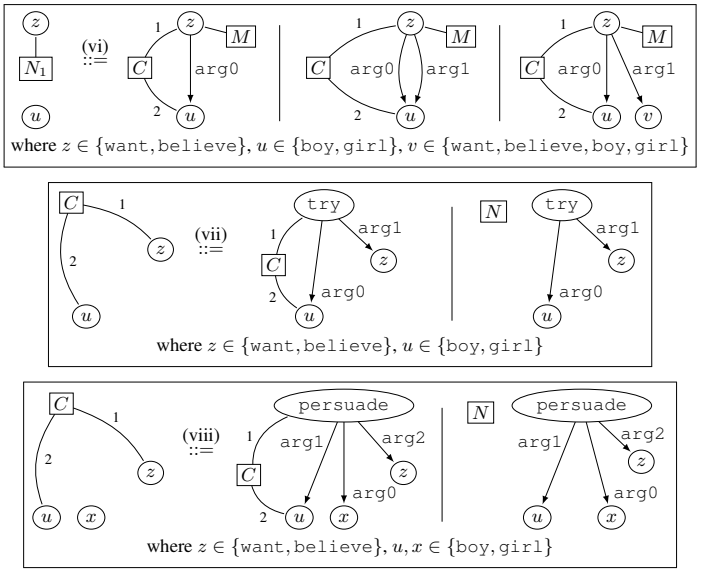


Figure 6: Rules implementing subject control (vii) and object control (viii).

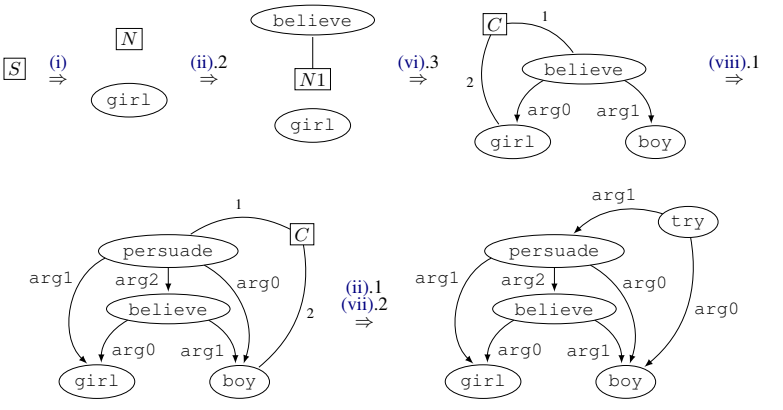


Figure 7: An example derivation using the rules in Figures 4 and 6.

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.