

UMEÅ UNIVERSITY

Efficient Best Trees Extraction and Contextual Grammars for Semantic Language Processing

Anna Jonsson

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

Acknowledgements

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

Here the contributions of the thesis author to the papers are stated. Naturally, the author has read, commented on, and accepted all papers.

Paper I	Implementation of algorithms and designing and performing experiments on software.
Paper II	Designing, performing, describing and reporting on the results of the experiments.
Paper III	Participating in the development of the algorithm, and proving its time complexity.
Paper IV	Implementing the algorithm and designing, performing, describing and giving the results of the experiments. Describing the competing algorithm and translating our algorithm to the hypergraph setting.
Paper V	Writing of first draft (which we then developed together).
Paper VI	TBD

Contents

1 Introduction **1**
1.1 Outline 5

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

2 Theoretical Foundation **9**
2.1 Weights and Semirings 9
2.2 Structures and Devices 11
2.3 General Problem Statement 15

3 Related Work **17**
3.1 The Best Runs Problem 18
3.2 The Best Trees Problem 21
3.3 Applications 24

4 Contributions **27**

Part Two: Semantic Graphs **31**

5 Background **33**

6 Contributions **35**

References **37**

Paper I **41**

Paper II **63**

Paper III **77**

Paper IV **91**

Chapter 1

Introduction

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

A *tree* is structure of *nodes* connected by *edges* (depicted as lines). Each node has at most one connected *parent* node (depicted above the node) and an arbitrary number of connected *child* nodes (depicted below the node); a node without a parent is a *root*, and a node without children is a *leaf*. In a syntax tree, every node has a label; the 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 shows how these roles work together. First, each word is assigned its syntactic role as its parent: 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), which is also the root of a syntax tree representing an entire sentence.

In Figure 1.1 we see the syntax tree for the interpretation that the stick was used for petting the dog, and Figure 1.2 portrays the interpretation that the dog has the stick. (The word "They" is an exception to the above: since it is not paired with anything to become part of a noun phrase, we skip the noun label.) In our case, the syntax trees represent one interpretation or *semantics* each, but there can be different syntax trees that represent the same semantics.

Now, which interpretation is the most probable one? If the example sentence is found in a book about a group 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 dogs, then the second one is perhaps more likely. Given the context of the sentence, we can learn which interpretation has the highest probability. This is exactly what machine learning is for: teaching a model to value sentence analyses based on the contexts of the

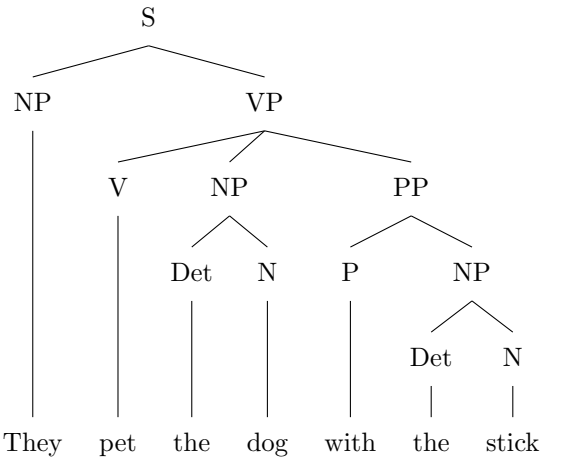


Figure 1.1: Analysis of "They pet the dog with the stick" representing the interpretation that the stick is used for petting the dog.

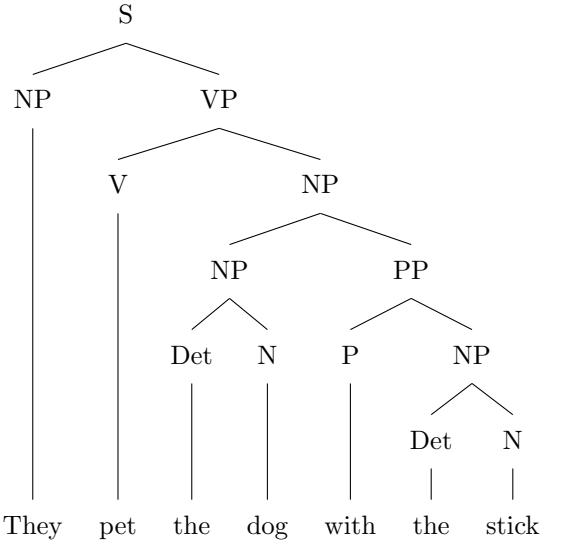


Figure 1.2: Analysis of "They pet the dog with the stick" representing the interpretation that the petted dog has a stick.

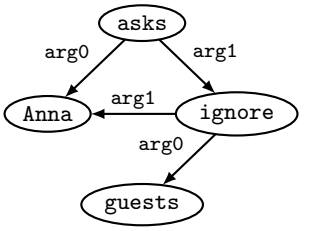
sentences. Machine learning of a model reduces to assigning weights to various components of the model. Common models are deep learning models (e.g. neural networks) and rule-based formal models; the latter are the focus of this thesis. Doing the actual machine learning in the best possible way is a thesis on its own, which is why we assume that we have already trained models that can tell us the ranking of the various interpretations or analyses.

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

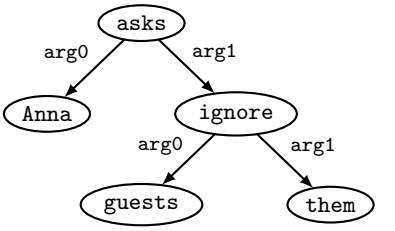
A wta can be seen as a compact representation of a weighted tree language by the use of *transition rules*. Given an input tree, a wta applies transition rules to it until the entire tree is processed, and outputs a weight. The application of transition rules to the tree forms a *run* on the tree. For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

Sometimes we need more than a syntax tree to distinguish between different semantics. For example, the sentence "Anna asks the guests 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 a group of people who are to be ignored. Forming a syntax tree does not help us distinguish between interpretations in this case, all we would get from the syntax tree is that "them" is a (pro)noun and how it is linked to the rest of the syntax of the sentence – not more semantic information. This semantic phenomenon of referencing back to some concept that has already been introduced is called *coreferencing* and demands a more complex representation.

A common solution to this problem is to model the semantics as a *graph*. A graph is a structure consisting of nodes and edges, just as trees but without the constraint that each node can have at most one parent. This implies that we have to indicate the relation between the nodes in some other way than drawing them above or below one another; here, we use directed edges pointing from the parent to the child. Figure 1.3 shows two graphs that each represent one



(a) The interpretation that 'them' refers to 'Anna'.



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

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

interpretation of the example sentence "Anna asks the guests to ignore them". The nodes are depicted as ellipses with their label inscribed, and the directed edges are drawn as arrows. The edges also have labels: `arg0` indicates that the child is the agent of the parent and `arg1` that the child is its patient. To interpret the graph, we start at the root in which we have the word "asks". By following the `arg0` edge, we find the asker, who is Anna. Next, we want to know what Anna is asking for, so we follow the `arg1` edge and arrive at "ignore". The ones who are asked to ignore are the guests, given by the outgoing `arg0` edge. Now we have arrived at the sole difference between these two graphs. In Figure 1.3(a), the `arg1` edge outgoing from "ignore" arrives at "Anna", indicating that Anna is the one to ignore, but in Figure 1.3(b), "them" is a node of its own and not connected to anything we have already seen in this particular sentence, meaning that "they" is a concept different from what we already have. If we had more data, we could probably say more about to whom "them" actually refers.

Note that a piece of semantics can be expressed in several ways using syntax trees or sentences: For example, the semantics of Figure 1.3(a) can additionally be expressed as the sentence "Anna is the one who asks to be ignored by the guests", which gives rise to a syntax tree that differs from the one for "Anna asks the guests to ignore them". Moreover, we should note that for handling coreferencing, so called *dependency trees* are commonly used. These are syntax trees that allows edges amongst leaf nodes, indicating to whom a pronoun refers, for example.

Det här är nog inte korrekt.

Brytpunkt: efter detta finns det inget mer färdigt att se i introduktionskapitlet

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

To represent weighted tree languages, we used weighted tree automata. For graphs, the formalism needs to be more general since we can express more complex dependencies using graphs. In our work, we use contextual hyper-

edge replacement grammars (CHRGs) for describing graph languages. These are an extension of the hyperedge replacement grammar (HRG). The intuitive difference is that CHRG can grab things that are not connected to any nonterminal and start to work on that part as well. Parsing of HRGs is intractable in general, but we attempt to restrict the rule type such that we get polynomial parsing and still can express a rather large variety of semantic graphs. Semantic modelling using graph grammars make out Part Two of this thesis, and the papers included in this part are Papers V and VI.

1.1 Outline

Part One of this thesis handles the N-best problem and its outline follows. Chapter 2 lays the theoretical foundation needed to discuss the N-best problem, Chapter 3 gives the background to the problem in the form of related work, and finally Chapter 4 summarises the papers on the N-best problem included in this thesis.

Part Two comprises semantic modelling by use of contextual hyperedge replacement grammars and has the following outline:

Part One

The *N*-Best Problem

Chapter 2

Theoretical Foundation

The N-best problem consists in extracting the N best hypotheses from a hypothesis space. As we have seen, common hypotheses for language processing tasks take the form of strings or trees – we call the hypothesis type the *structure*. Then, we have the *devices* that define the hypothesis space compactly such as automata and grammars. Finally, there has to be 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, we have to make sure that the algorithm uses weight operations that enable a correctness proof. A *semiring* is, informally, a set of ~~logical entities~~ values (weights) and operations defined on them. Proving correctness of an algorithm ~~heavily~~ often depends on being able to prove that it is compatible with a certain semiring or family of semirings. For example, Dijkstra's greedy algorithm for finding the shortest path in a weighted directed graph [Dij59] assumes weights from $\mathbb{R}_+$ (non-negative reals), and uses the operations + and min: + combines weights along a path and min picks the smallest-weighted path over all paths in the graph. The min operation makes it possible to find the shortest path without considering all possible paths, and is therefore central to the correctness and efficiency of Dijkstra's algorithm. Another example is given by Mohri [Moh02] who generalises the work of Dijkstra by developing a shortest-path algorithm that works for not only one, but a family of semirings. In short, Mohri's algorithm requires that the semiring used is such that the path weights cannot be decreased by traversing cycles in the input graph, a condition that is trivially fulfilled by the preconditions of applying Dijkstra's algorithm but also works for other combinations of weights and operations. As is common when generalising to a richer domain, Mohri's algorithm is less efficient than Dijkstra's.

Below follows the formal definition of a semiring, starting with an auxiliary 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 $\text{TRUE} \vee \text{TRUE} = \text{TRUE}$ and $\text{FALSE} \vee \text{FALSE} = \text{FALSE}$, the Boolean semiring is idempotent; it is also extremal since the $\vee$ operation always returns either TRUE or FALSE. Moreover, the commutative operation $\wedge$ is its semiring multiplication, which implies that the Boolean semiring is also commutative. Another common idempotent, extremal and commutative semiring is the *tropical semiring*.

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

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

2.2 Structures and Devices

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

Let S be a set, and let $|S|$ denote the cardinality of S; the set operator $\uplus$ denotes the disjoint union of sets. A *string* over S is a finite sequence of symbols from S, and S^* denotes the set of all strings over S; λ represents the empty string. The length of a string s is 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 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 children. For convenience, we identify a node by its label when discussing specific trees.

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

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

- N is a finite set of *nonterminals*,
- Σ with $\Sigma \cap N = \emptyset$ is a finite set of symbols or *terminals*,
- $S \in N$ is the *start nonterminal*, and
- R is a finite set of *production rules* on the form $A \to \alpha$ with $A \in N$ and $\alpha \in (N \cup \Sigma)^*$; the 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 in G of a string $s \in \mathcal{L}(G)$ is a tree whose nodes are production rules from G such that the result of applying the production rules (starting at the root) is s.

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. The next device uses the same semiring:

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

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

Which "set of transitions"?

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 ∞.) Formally, this turns δ into a partial function $\delta : Q \times \Sigma \times Q \nrightarrow \mathbb{R}_+$

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 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.

Tycker det här är för invecklat och inte riktigt i linje med definitionen i trädfallet. Varför inte säga att en run är en sträng $q_0 \cdots q_n$ där q_0 är det init. tillståndet och $q_1, \dots, q_n \in Q$? Och att en run är accepterande om $q_n \in Q_f$?

S

define weighted

faktiskt inte

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

Figure 2.1: Example wta.

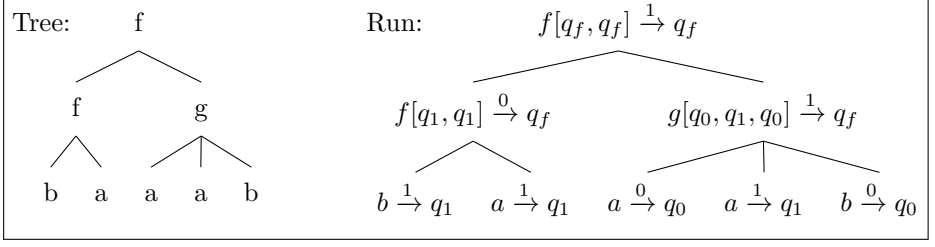


Figure 2.2: Left: A graphic representation of the tree $f[f[b, a], g[a, a, b]]$. Right: An example run for the wta given in Figure 2.1 on the tree to the left.

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*,
- $Q \subseteq \Sigma_0$ 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 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 ~~for $d = e[d_1, \ldots, d_{|e|}]$,~~ 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 $G_M = (V, E, \mathrm{att}, \mathrm{lab_E}, \mathrm{wt}_G)$ in the following way: For every state $q \in Q$, add a node v_q to V, and for every transition rule $r = (\sigma[q_1, \ldots, q_k] \xrightarrow{w} q)$, 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.) We define the homomorphism h from trees labelled over δ to trees labelled over E as $h(r[\rho_1, \ldots, \rho_k]) = e_r[h(\rho_1), \ldots, h(\rho_k)]$.

Här skulle du kunna hänvisa till vanliga ändliga automater och deras grafrepresentation som tillståndsdiagram.

Vore det inte bättre att använda samma typ av hypergraf som i pappret?

V^+

Ogrammatiskt

[redundant]

?

?

Otydligt

Theorem 2.9 *Let M be a wta and G_M its corresponding hypergraph. Then, ρ is a run of M on a tree t if and only if $h(\rho)$ is a derivation of G_M with* $\mathrm{wt}(\rho) = \mathrm{wt}(h(\rho))$ *and* $\mathrm{result}(h(\rho)) = t$.

Proof. In the following, let ρ be a run of M on a tree t. If t consists of a single node labelled σ, then ρ is by the definition of run a single node labelled with some transition rule $r = (\sigma \xrightarrow{w} q)$, and applying h to ρ yields $h(\rho) = e_r$. Moreover, e_r is by construction of G_M and Definition 2.8 a derivation of v_q with $\mathrm{wt}(h(\rho)) = w = \mathrm{wt}(\rho)$ and $\mathrm{result}(h(\rho)) = \sigma = t$. Now assume for the sake of induction that the theorem holds for runs $\rho_1, \ldots, \rho_k$ on trees $t_1, \ldots, t_k$, respectively. Then, for the run $\rho = r[\rho_1, \ldots, \rho_k]$ with $r = (\sigma[q_1, \ldots, q_k] \xrightarrow{w} q)$ on $t = \sigma[t_1, \ldots, t_k]$, we have $h(\rho) = e_r[h(\rho_1), \ldots, h(\rho_k)]$. Since $h(\rho_1), \ldots, h(\rho_k)$ are derivations of $v_{q_1}, \ldots, v_{q_k}$, respectively, then – again by construction of G_M and Definition 2.8 – $e_r[h(\rho_1), \ldots, h(\rho_k)]$ is a derivation of v_q in G_M with $\mathrm{wt}(e_r[h(\rho_1), \ldots, h(\rho_k)]) = w + \mathrm{wt}(\rho_1) + \cdots + \mathrm{wt}(\rho_k) = \mathrm{wt}(\rho)$ and $\mathrm{result}(e_r[h(\rho_1), \ldots, h(\rho_k)]) = \sigma[\mathrm{result}(h(\rho_1)), \ldots, \mathrm{result}(h(\rho_k))] = \sigma[t_1, \ldots, t_k] = t$.

Conversely, let d be any derivation in G.

□

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 $M(t)$. □

2.3 General Problem Statement

For devices that define weighted formal languages, the *N-best 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
> 3. there is no element x distinct from $x_1, \ldots, x_i$ 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.

Tycker att du slänger bort en chans som din fina formalisering öppnar. Den säger ju inte vad ett "element" är. Du skulle kunna diskutera att om "device" är vita och den betraktas som något som definierar språket av alla dess runs så fås N-best runs-problemet, som är förhållandevis enkelt eftersom beteendet är deterministiskt, dvs '+' i semiringen används inte. Går man över till träd så innebär det att man betraktar allt under homomorfismen som mappar runs till motsvarande träd, som kan vara ickeinjektiv, och då kommer + (här min) in i bilden, dvs man behöver förkasta dupletter. Du skulle t.o.m. kunna diskutera att det skulle bli ännu svårare om semiringar med icke extremalt (aj!) + används eftersom det då inte längre skulle räcka till med att förkasta dupletter.

Chapter 3

Related Work

To start our review of research on the N-best problem, we go back to 1977 when 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, takes a different approach: Every production rule $r = (A \to \alpha)$ is assigned a function f_r. The variables of f_r are given by $\text{nt}(\alpha) = \{x_1, \ldots, x_{|\text{nt}(\alpha)|}\}$, and if $\text{nt}(\alpha) = \emptyset$, 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 ~~$x_1, \ldots, x_{|\text{nt}(\alpha)|}$~~ and that $f_r(x_1, \ldots, x_{|\text{nt}(\alpha)|})$ 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 developed by Knuth.

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 $\text{nt}(\alpha) = \emptyset$, 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 $\text{nt}(\alpha) = \emptyset$ 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 weighted cfg $G = (N, \Sigma, S, R)$ with the superiority property produces a string of smallest weight in $\mathcal{L}(G)$.

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

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 divided 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 applications of the N-best trees problem, and related problems (Section 3.3).

3.1 The Best Runs Problem

Huang and Chiang [HC05] present an algorithm that solves the N-best derivations problem for a finite hypergraph (see Definition 2.7). As shown 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. (This

is equivalent to the superiority property of [Knu77], but they call it monotone since they formulate it as a property of the hypergraph.) Using an idempotent and commutative semiring such as the tropical semiring is therefore less general in what weight relations we can have but fulfils the requirements and lets us compare this algorithm to similar ones in a clearer manner.

The input to Huang and Chiang's algorithm is a triple (G, v_0, N) where $G = (V, E, \mathrm{att}, \mathrm{lab_E}, \mathrm{wt}_G)$ is a hypergraph, $v_0 \in V$ is the *target node*, and $N \in \mathbb{N}^\infty$. The goal is to 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 $\{d \mid \mathrm{root}(d) = v_0\} \setminus \{d_1, \ldots, d_{i-1}\}$.

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 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 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.

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
```

The first generalisation is the labelling of the hypergraphs which is motivated by the need to represent tree-based language models such as weighted tree automata. This generalisation simply adds labels to the hyperedges, and does not require any changes to the algorithm itself. Similarly, the use of the programming language Haskell and its lazy evaluation scheme (the second generalisation) does not change the algorithm other than expressing it in a functional programming setting – the Huang and Chiang algorithm is already lazy by 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 $d \precsim c$. 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's 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 $w \in \Sigma^* \setminus \{w_1, \ldots, w_{i-1}\}$ for which $M(w) \leq M(w_i)$ for $i \in \{1, \ldots, N\}$.

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
```

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

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. We from here on refer to this algorithm as Best Trees v.1.

To facilitate the description of the algorithm, the input wta $M = (Q, \Sigma, \delta, Q_f)$ is split into two wtas given a state $q \in Q$: $M^q = (Q, \Sigma, \delta, \{q\})$ that takes us to q, and $M_q = (Q, \Sigma \uplus \{\square\}, \delta \uplus \{\square \xrightarrow{0} q\}, Q_f)$ that takes us from q to a final state. The symbol $\square$ of rank 0 is merely a placeholder for some subtree given by M^q. A tree labelled over $\Sigma \cup \{\square\}$ with exactly one occurrence of $\square$ 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 coincide. 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

To motivate the research on the N-best trees problem, we review applications in which N-best lists are extracted. All of the applications mentioned here use algorithms that extract (the trees corresponding to) the N best runs instead of trees. Theoretically, this could lead to duplicate trees in the N-best list, unless the input device is deterministic. Since duplicates do not contribute with more information, they do not contribute to increased result quality. Instead

extracting the best trees would guarantee that the resulting N best list is free from duplicates and thereby contain more information.

Socher et al. [SBMN13] introduce the compositional vector grammar (cvg) that is built on top of a standard probabilistic context-free grammar (pcfg) model which is then combined with a recursive neural network (rnn). First, the pcfg is trained on the input data, then the top 200 derivations are extracted and input into the rnn which then outputs the final result. This is a type of re-ranking of the trees, with the exception that the output tree of the rnn can be distinct from any of the input trees.

Training a pcfg or a similar model and then extracting the top derivations is a common way of propagating partial results to the next step of the language processing pipeline, and 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 study consists in looking at existing string-based methods and exploring the literature to find replacement tree transducer types and algorithms that work for the tree case.

Pipelines for NLP tasks often consist of a number of fsts and end in a finite-state string acceptor (fsa) that represents and ranks the output strings. Knight and Graehl consider several possible replacements for fsts in the form of various tree transducers. Their conclusion is that there are replacement tree transducers that could be employed for every use case considered. Next, probabilistic regular tree grammars (prtgs) are mentioned as suitable replacements for fsas. In connection with this, Knight and Graehl 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 insufficient since they do not extract N distinct hypotheses but N distinct runs. To support this motivation, they run the Stanford parser, extract the $N = 50$ best runs and observe that about half of the output trees are actually duplicates – enough to affect the outcome of the language model pipeline. Thus, extracting the best trees instead of the best runs is not only theoretically better, but also of practical significance.

Kom ihåg att beskriva det egna bidraget till varje papper, lämpligen i anslutning till texten som beskriver pappret.

Chapter 4

Contributions

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

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

Paper I presents a generalised and abstract algorithm for N-best extraction. The device is a hypergraph (see Definition 2.7) that is acyclic and possibly infinite, and the goal is to compute N vertices (or nodes) that are optimal with respect to a *nice* semiring. A nice semiring is a semiring that is finitely generated, idempotent and that has $\mathbb{1}$ as its smallest element. The generalisation is done with respect to the Best Trees v.1 algorithm (Björklund et al. [BDZ15]) which only covers wta over the tropical semiring: this algorithm covers all devices whose computations can be modelled as a hypergraph, and the algorithm is proven to work for a family of semirings instead of a single one. The abstraction consists in having to implement a number of unimplemented functions that depend on the input of the instantiating algorithm.

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

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

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

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

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

In Paper III, we develop a new and more efficient version of Best Trees v.1, here called Best Trees. The two versions both use a priority queue of trees that prioritises a tree t that is in the state q based on the sum of t's current weight at q and the best context weight of q. However, Best Trees additionally uses a second level of priority queues, one for each transition rule of the input wta, which allows us to prune the tree candidates with respect to the transitions. The second-level priority queues are in fact elements in the main priority queue, and they are prioritised based on their top element – the currently highest-prioritised tree for the corresponding rule. New tree candidates are created as in the Huang and Chiang algorithm [HC05]: For every tree t that is dequeued from the main queue via the rule queue corresponding to the rule $r = (\sigma[q_1, \ldots, q_k] \xrightarrow{w} q)$, we successively exchange each direct subtree t' of t that thus is in some state q_i for $i \in \{1, \ldots, k\}$ by the next best tree for q_i, yielding as many candidates as the rank of r, namely k. (We might have seen some of the candidates before for the rule queue of r, and in that case, the already seen candidates are disregarded.) When a tree t is dequeued from a rule queue, only that particular rule queue has to be updated with new candidates. If a candidate cannot be instantiated immediately since at least one of its direct subtrees has not been dequeued from the main queue yet, the candidate is marked as pending. For the rest of the rule queues, we at most have to instantiate the pending candidate trees that t has now made available and add them to their corresponding rule queues. The time complexity of Best Trees is $O\big(Nm(\log m + r^2 + r \log(Nr))\big)$ where m is the number of transition rules in the input wta and r is the maximum rank over all of its transitions.

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

Paper IV:

Paper IV consists of an experimental evaluation of Best Trees in relation to both the previous version Best Trees v.1 and the Huang and Chiang algorithm [HC05] in the form of Tiburon [MK06] – just as for Paper II. However, this time we do not only use artificial data for the sake of investigating the effect of nondeterminism, but also wta that result from tasks on natural language data. We use Betty to solve both the best trees and the best runs problem, and Tiburon to solve the best runs problem. Betty outperforms Tiburon for the best runs task on all input wtas except one. The exception is a natural language-based wta with more than a million transitions whose language contains many very small trees with low weights. This shows how extreme a situation we need to provoke the case where the computation of the best contexts actually makes a visible overhead. Furthermore, using Betty for solving the best trees task yields only slightly worse running times than using Betty for the best runs task. The exception here is the artificial corpus with exponentially many duplicates which as expected slows Betty down greatly when used for finding the best trees.

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

We believe that the success of Best Trees is due to both the best context weights and the two levels of priority queues. This is supported by a quick test we did by removing the best contexts part after which we could observe that the number of candidates in the second-level priority queues was increased by roughly a factor 10. The exact effect of the various parts would however have to be investigated more thoroughly to know for sure.

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