# Chapter 1
# Introduction

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

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

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

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

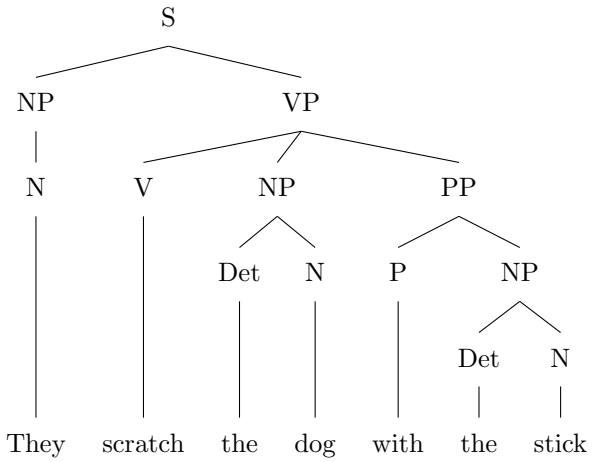



Figure 1.1: Analysis of "They scratch the dog with the stick" representing the interpretation that the stick is used for scratching the dog.

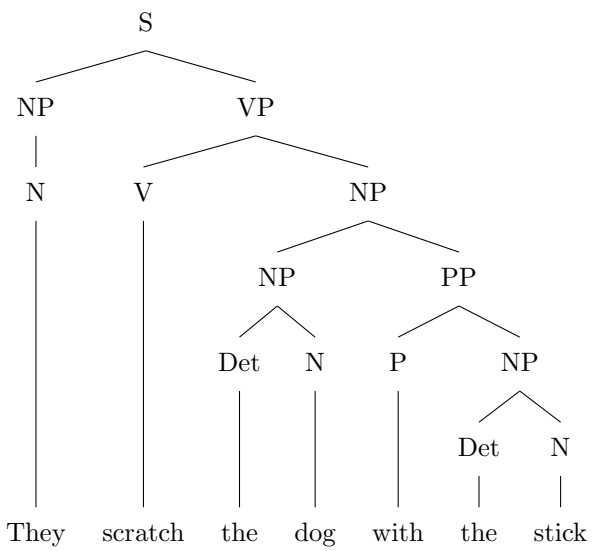



Figure 1.2: Analysis of "They scratch the dog with the stick" representing the interpretation that the scratched dog has a stick.

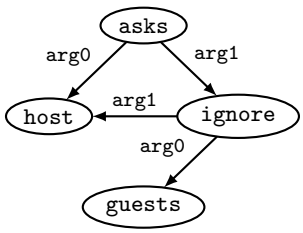
second one is perhaps more likely. Given the context of the sentence, we can learn which interpretation has the highest probability. This is exactly what machine learning is for: teaching a model to value sentence analyses based on the contexts of the sentences. Machine learning of a model reduces to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

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

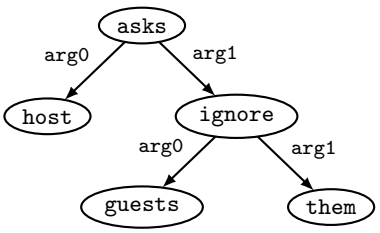
Every wta defines a potentially infinite *weighted tree language*, which means that a wta can be seen as a compact representation of its weighted tree language since its set of *transition rules* is finite. Given an input tree, a wta applies transition rules to it until the entire tree is processed,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

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

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




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




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

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

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

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

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

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

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

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

## 1.1 Outline

Part One of this thesis handles the $N$-best problem and is outlined as follows. Chapter 2 lays the theoretical foundation needed to formally define and 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 with 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 ranked hypothesis space. As we have seen, common hypotheses for language processing tasks take the form of strings or trees – we call the hypothesis type the *structure*. Then, we have the *devices* that define the hypothesis space compactly such as automata and grammars. Finally, there has to be something that ranks the structures, some criterion that decides if a structure is better than another, and this is where weights and semirings enter the picture. Let us now provide formal definitions for each of these components, starting with weights and semirings.

## 2.1 Weights and Semirings

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

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

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

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

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

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

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

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

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

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

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

## 2.2 Structures and Devices

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.1: Example wta.

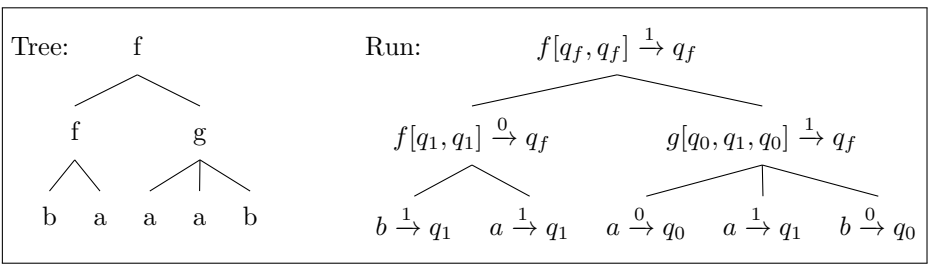



Figure 2.2: Left: A graphic representation of the tree $f[f[b, a], g[a, a, b]]$. Right: An example run for the wta given in Figure 2.1 on the tree to the left.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 2.3 General Problem Statement

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

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

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

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

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

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

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

# Chapter 3
# Related Work

To start our review of research on the $N$-best problem, we go back to 1977 when Knuth [Knu77] presented an algorithm that finds the best (smallest) derivation for a weighted context-free grammar (cfg, Definition 2.4). Recall that in order to make a cfg $G = (N, \Sigma, S, R)$ weighted, we assign weights from $\mathbb{R}_+$ to each of its production rules. Knuth, however, takes a more general approach: Every production rule $r = (A \to \alpha)$ is assigned a function $f_r$. The variables of $f_r$ are given by $\mathrm{nt}(\alpha) = \{x_1, \ldots, x_{|\mathrm{nt}(\alpha)|}\}$, and if $\mathrm{nt}(\alpha) = \emptyset$, $f_r$ is constant-valued. The weight of a derivation of $s \in \mathcal{L}(G)$ is the result of recursively applying the functions for every production rule used in the generation of $s$. Moreover, $f_r$ has to fulfil the *superiority property*, meaning that $f_r$ is non-decreasing in every variable and that $f_r(x_1, \ldots, x_{|\mathrm{nt}(\alpha)|})$ cannot be smaller than any individual value $x_i$. In other words: applying a rule cannot decrease the total weight of the derivation. Using 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 for nonterminals $A$ that occur in productions $A \xrightarrow{w} \alpha$ with $\mathrm{nt}(\alpha) = \lambda$, it holds that $A \Rightarrow \alpha$. Furthermore, the $\alpha$ that minimises the rule weight $w$ over all production rules $(A \xrightarrow{w} \alpha) \in R$ with $\mathrm{nt}(\alpha) = \lambda$ is optimally derived from its corresponding nonterminal $A$. Due to the superiority property of the production rules, there cannot be an $\alpha' \neq \alpha$ for which $A \Rightarrow \alpha'$ with smaller weight than the minimal $w$. Thus, $w$ can be stored as a result in the array $res[A]$ (given that the nonterminals can be internally represented as integers, we allow that arrays are indexed using nonterminals). To mark that the weight of $A$'s best derivation has now been determined, $A$ is added to the formerly empty set $D$. Now knowing the best derivation for the nonterminal in $D$, we are no longer restricted to rules that have only terminals in their right-hand sides, but these can now also contain the nonterminal in $D$ since its smallest weight is already determined. This allows us to pick the smallest-weighted derivation for a new nonterminal in the same way as we did before. We do this iteratively and when $D$ contains all nonterminals, we are done, and we return $res$ containing the result.

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 also the actual derivations, which we can keep track of during the computation of the weights. The 1-best derivation is then the derivation corresponding to $res[S]$ (which also yields the best string in $\mathcal{L}(G)$).

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

## 3.1 The Best Runs Problem

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

In the original paper, each hyperedge has its own weight function that takes the weights of the tail's nodes to a weight that is then assigned the head node. Moreover, those weight functions have to be monotonic in the sense that increasing one of the tail weights does not decrease the head's weight. (This is

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 3.2 The Best Trees Problem

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 3.3 Applications

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

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

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

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

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

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

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

# Chapter 4
# Contributions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Part Two

# Semantic Graph Grammars

# Chapter 5
# Background

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

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

## 5.1 Semantic Graphs

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

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

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

occurs ~~two times~~ twice: as the `destination` and as `Tudor`'s `home`. Sentence plans are in fact directed graphs in which the concepts are node labels and the attributes are edge labels.

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

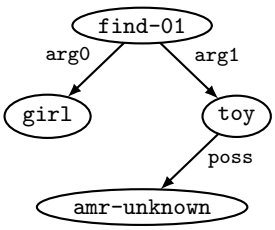
- $V$ is a finite set of *nodes*,
- $E$ is a finite set of *edges*,
- $\mathrm{src}\colon E \to V$ is the *assignment of source nodes to edges*,
- $\mathrm{tar}\colon E \to V$ is the *assignment of target nodes to edges*,
- $\mathrm{lab}_V\colon V \to \Sigma_V$ is the *labelling of nodes*, and
- $\mathrm{lab}_E\colon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

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

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

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

(a) SPL/AMR format.




(b) Graph format.

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

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

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

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

## 5.2 Modelling Semantic Graph Languages

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Normally, we separate the recognition problem into two problems: the *uniform membership problem* that considers both $M$ and $G$ as input, and the *non-uniform membership problem* that only considers $G$ as input. The existence of an efficient algorithm that solves the recognition problem is central to the usability of a rule-based formalism in natural language processing applications: if we cannot efficiently check if a piece of natural language is correct according to our model, then our model is of no practical use. Since the algorithm of Chiang et al. solves a problem that is more general than the recognition problem for dag automata, their result holds for the recognition of dag automata as well. In general, the non-uniform membership problem for dag automata is NP-complete (and thereby also the harder uniform membership problem), but the result of Chiang et al. shows that when $G$ has bounded treewidth, the recognition problem can be solved in polynomial time.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

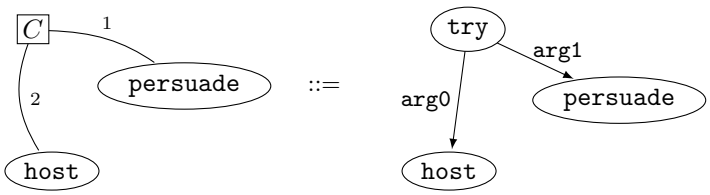



Figure 5.3: A context-free rule adding the control verb "try" to an already existing semantic graph containing the concepts "persuade" and "host". For appropriate usage, the rule requires that `host` is the agent of persuade.

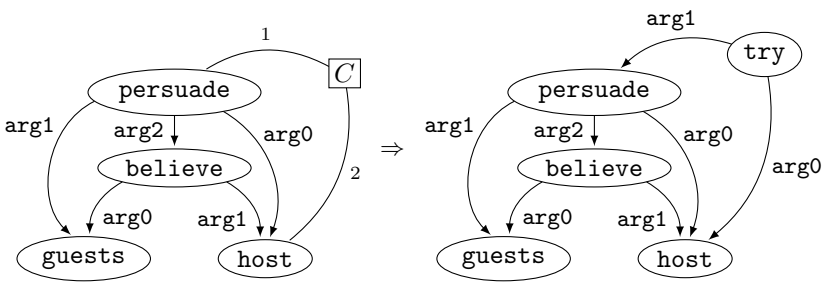



Figure 5.4: An application of the rule in Figure 5.3 to a semantic graph, resulting in the addition of a verb that requires subject control. The resulting graph can be read as "The host tries to persuade the guests to believe them."

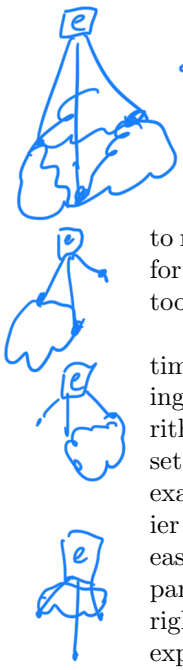
**Definition 5.10** Given a graph grammar $\Gamma$ and a graph $G$, the *parsing problem* asks if $G$ is in $\mathcal{L}(\Gamma)$.

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

Lautemann [Lau90] proves by the construction of an algorithm that uniform parsing of an HRG $\Gamma$ can be done in polynomial time given two conditions:

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

Example language that violates Condition #2 ?



- For every graph $G \in \mathcal{L}(\Gamma)$ and nonterminal $e$ labelled by some $A \in \Sigma_N$, $G$ can be derived from $e$ if and only if the graph induced by $G$ on $C \cup \{v_1, \ldots, v_r\}$, for every connected component $C$ of $G \setminus \{v_1, \ldots, v_r\}$, can be derived from $e$.

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

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

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

linear?

decide

When modelling semantic graph languages, we want to be able to express all semantic graphs over a domain of concepts and relations that represent a sensical meaning. At the same time, we want to have the ability to leave out semantic graphs with nonsensical meaning. A strength of hyperedge replacement is that it provides the structural control that some language constructions require. To exemplify this, again consider Figure 5.3 which depicts a context-

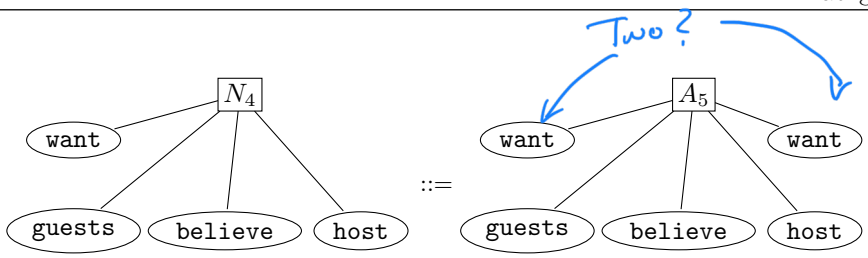



Figure 5.5: Context-free rule for adding the concept `want` to a semantic graph.

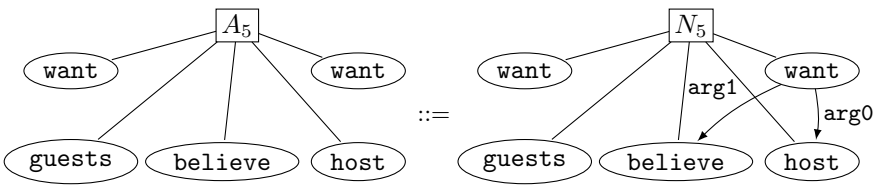



Figure 5.6: Context-free rule for adding outgoing edges from the concept `want` to other nodes of a semantic graph.

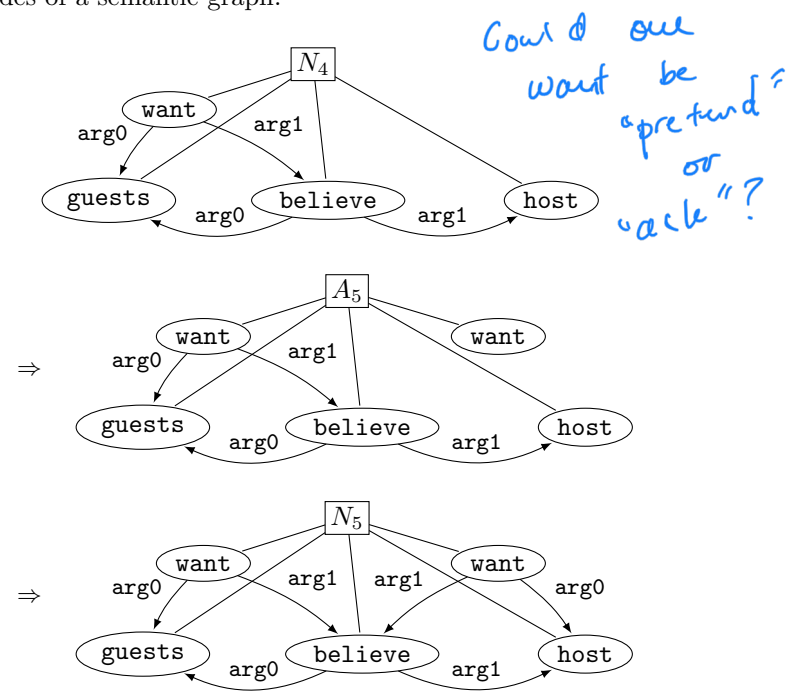



Figure 5.7: A derivation of a verb that does not require any control using the context-free rules in Figures 5.5 and 5.6, in that order. The first hypergraph can be interpreted as "The guests want to believe the host" and the last one reads "The guests want to believe the host and the host wants the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

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

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

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

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

**Definition 5.12** A *contextual hyperedge replacement grammar (CHRG)* is an HRG $\Gamma = (\Sigma, \mathcal{R}, Z)$ for ~~which~~ $\mathcal{R}$ can contain ~~in addition to~~ the context-free rules and contextual rules.

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

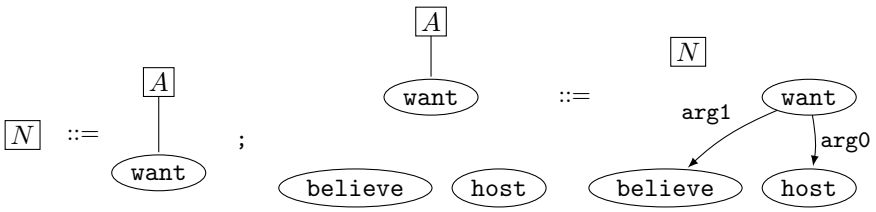



Figure 5.8: Two semicolon separated rules. To the left: a context-free rule. To the right: a contextual rule whose left-hand side contains two contextual nodes (`believe` and `host`).

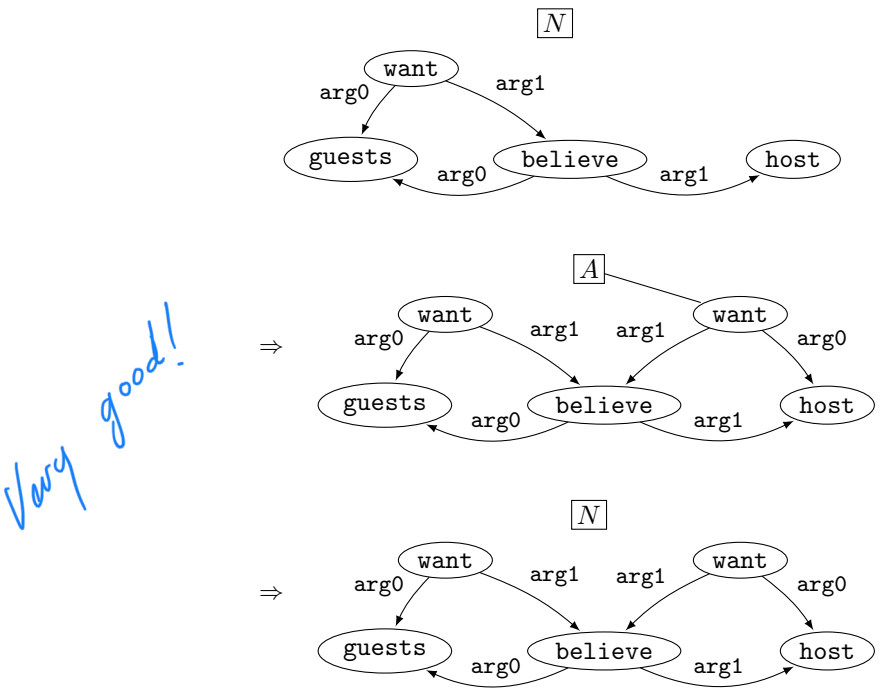



Figure 5.9: The same derivation as in Figure 5.7 but this time using the rules from Figure 5.8 out of which one is contextual.

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

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

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

## 5.3 Mapping Between Language Representations

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

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

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

1. A synchronous HRG (SHRG), which generates a string and a graph in parallel using the same rule set. The SHRG can thus easily be used for mapping in both directions.

2. A ~~cascade~~ of two devices: a bottom-up dag-to-tree transducer (d2t) that creates a tree from the AMR, and a top-down tree-to-string transducer (LNT) that takes the yield of the tree to extract the English string.

3. A cascade of four devices: a d2t as in the previous cascade, two extended tree-to-tree transducers (xLNT) – one that introduces verbs and one that introduce pronouns – and finally an LNT, again for extracting the yield of the resulting tree.

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

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

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

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

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

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

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

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

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

# Chapter 6
# Contributions

## Paper V: Contextual Hyperedge Replacement Grammars for Abstract Meaning Representations

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

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

## Paper VI: Polynomial Graph Parsing with Non-Structural Reentrancies

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

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

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

# References


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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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