n-Best Parsing Revisited*

Matthias Büchse and **Daniel Geisler** and **Torsten Stüber** and **Heiko Vogler**
Faculty of Computer Science
Technische Universität Dresden
01062 Dresden
{buechse,geisler,stueber,vogler}@tcs.inf.tu-dresden.de

Abstract

We derive and implement an algorithm similar to (Huang and Chiang, 2005) for finding the n best derivations in a weighted hypergraph. We prove the correctness and termination of the algorithm and we show experimental results concerning its runtime. Our work is different from the aforementioned one in the following respects: we consider labeled hypergraphs, allowing for tree-based language models (Maletti and Satta, 2009); we specifically handle the case of cyclic hypergraphs; we admit structured weight domains, allowing for multiple features to be processed; we use the paradigm of functional programming together with lazy evaluation, achieving concise algorithmic descriptions.

1 Introduction

In statistical natural language processing, probabilistic models play an important role which can be used to assign to some input sentence a set of analyses, each carrying a probability. For instance, an analysis can be a parse tree or a possible translation. Due to the ambiguity of natural language, the number of analyses for one input sentence can be very large. Some models even assign an infinite number of analyses to an input sentence.

In many cases however, the set of analyses can in fact be represented in a finite and compact way. While such a representation is space-efficient, it may be incompatible with subsequent operations. In these cases a finite subset is used as an approximation, consisting of n best analyses, i. e. n analyses with highest probability. For example, this approach has the following two applications.

(1) Reranking: when log-linear models (Och and Ney, 2002) are employed, some features may not permit an efficient evaluation during the computation of the analyses. These features are computed using individual analyses from said approximation, leading to a reranking amongst them.

(2) Spurious ambiguity: many models produce analyses which may be too fine-grained for further processing (Li et al., 2009). As an example, consider context-free grammars, where several leftmost derivations may exist for the same terminal string. The weight of the terminal string is obtained by summing over these derivations. The n best leftmost derivations may be used to approximate this sum.

In this paper, we consider the case where the finite, compact representation has the form of a weighted hypergraph (with labeled hyperedges) and the analyses are derivations of the hypergraph. This covers many parsing applications (Klein and Manning, 2001), including weighted deductive systems (Goodman, 1999; Nederhof, 2003), and also applications in machine translation (May and Knight, 2006).

In the nomenclature of (Huang and Chiang, 2005), which we adopt here, a derivation of a hypergraph is a tree which is obtained in the following way. Starting from some node, an ingoing hyperedge is picked and recorded as the label of the root of the tree. Then, for the subtrees, one continues with the source nodes of said hyperedge in the same way. In other words, a derivation can be understood as an unfolding of the hypergraph.

The n-best-derivations problem then amounts to finding n derivations which are best with respect to the weights induced by the weighted hypergraph.[1] Among others, weighted hypergraphs with labeled hyperedges subsume the following two concepts.

(I) probabilistic context-free grammars (pcfgs).

* This research was financially supported by DFG VO 1101/5-1.

[1] Note that this problem is different from the n-best-hyperpaths problem described by Nielsen et al. (2005), as already argued in (Huang and Chiang, 2005, Section 2).

In this case, nodes correspond to nonterminals, hyperedges are labeled with productions, and the derivations are exactly the abstract syntax trees (ASTs) of the grammar (which are closely related the parse trees). Note that, unless the pcfg is unambiguous, a given word may have several corresponding ASTs, and its weight is obtained by summing over the weights of the ASTs. Hence, the n best derivations need not coincide with the n best words (cf. application (2) above).

(II) weighted tree automata (wta) (Alexandrakis and Bozapalidis, 1987; Berstel and Reutenauer, 1982; Ésik and Kuich, 2003; Fülöp and Vogler, 2009). These automata serve both as a tree-based language model and as a data structure for the parse forests obtained from that language model by applying the Bar-Hillel construction (Maletti and Satta, 2009). It is well known that context-free grammars and tree automata are weakly equivalent (Thatcher, 1967; Ésik and Kuich, 2003). However, unlike the former formalism, the latter one has the ability to model non-local dependencies in parse trees.

In the case of wta, nodes correspond to states, hyperedges are labeled with input symbols, and the derivations are exactly the runs of the automaton. Since, due to ambiguity, a given tree may have several accepting runs, the n best derivations need not coincide with the n best trees. As for the pcfgs, this is an example of spurious ambiguity, which can be tackled as indicated by application (2) above. Alternatively, one can attempt to find an equivalent deterministic wta (May and Knight, 2006; Büchse et al., 2009).

Next, we briefly discuss four known algorithms which solve the n-best-derivations problem or subproblems thereof.

• The Viterbi algorithm solves the 1-best-derivation problem for acyclic hypergraphs. It is based on a topological sort of the hypergraph.

• Knuth (1977) generalizes Dijkstra's algorithm (for finding the single-source shortest paths in a graph) to hypergraphs, thus solving the case $n = 1$ even if the hypergraph contains cycles. Knuth assumes the weights to be real numbers, and he requires weight functions to be monotone and superior in order to guarantee that a best derivation exists. (The superiority property corresponds to Dijkstra's requirement that edge weights—or, more generally, cycle weights—are nonnegative.)

• Huang and Chiang (2005) show that the n-best-derivations problem can be solved efficiently by first solving the 1-best-derivation problem and then extending that solution in a lazy manner. Huang and Chiang assume weighted unlabeled hypergraphs with weights computed in the reals, and they require the weight functions to be monotone.

Moreover they assume that the 1-best-derivation problem be solved using the Viterbi algorithm, which implies that the hypergraph must be acyclic. However they conjecture that their second phase also works for cyclic hypergraphs.

• Pauls and Klein (2009) propose a variation of the algorithm of Huang and Chiang (2005) in which the 1-best-derivation problem is computed via an A*-based exploration of the 1-best charts.

In this paper, we also present an algorithm for solving the n-best-derivations problem. Ultimately it uses the same algorithmic ideas as the one of Huang and Chiang (2005); however, it is different in the following sense:

1. we consider labeled hypergraphs, allowing for wta to be used in parsing;

2. we specifically handle the case of cyclic hypergraphs, thus supporting the conjecture of Huang and Chiang; for this we impose on the weight functions the same requirements as Knuth and use his algorithm;

3. by using the concept of linear pre-orders (and not only linear orders on the set of reals) our approach can handle structured weights such as vectors over frequencies, probabilities, and reals;

4. we present our algorithm in the framework of functional programming (and not in that of imperative programming); this framework allows to decribe algorithms in a more abstract and concise, yet natural way;

5. due to the lazy evaluation paradigm often found in functional programming, we obtain the laziness on which the algorithm of Huang and Chiang (2005) is based for free;

6. exploiting the abstract level of description (see point 4) we are able to prove the correctness and termination of our algorithm.

At the end of this paper, we will discuss experiments which have been performed with an implementation of our algorithm in the functional programming language HASKELL.

2 The n-best-derivations problem

In this section, we state the n-best-derivations problem formally, and we give a comprehensive

example. First, we introduce some basic notions.

Trees and hypergraphs The definition of ranked trees commonly used in formal tree language theory will serve us as the basis for defining derivations.

A *ranked alphabet* is a finite set Σ (of *symbols*) where every symbol carries a *rank* (a nonnegative integer). By $\Sigma^{(k)}$ we denote the set of those symbols having rank k. The *set of trees over* Σ, denoted by T_Σ, is the smallest set T such that for every $k \in \mathbb{N}$, $\sigma \in \Sigma^{(k)}$, and $\xi_1, \ldots, \xi_k \in T$, also $\sigma(\xi_1, \ldots, \xi_k) \in T$;[2] for $\sigma \in \Sigma^{(0)}$ we abbreviate $\sigma()$ by σ. For every $k \in \mathbb{N}$, $\sigma \in \Sigma^{(k)}$ and subsets $T_1, \ldots, T_k \subseteq T_\Sigma$ we define the *top-concatenation (with σ)* $\sigma(T_1, \ldots, T_k) = \{\sigma(\xi_1, \ldots, \xi_k) \mid \xi_1 \in T_1, \ldots, \xi_k \in T_k\}$.

A Σ-*hypergraph* is a pair $H = (V, E)$ where V is a finite set (of *vertices* or *nodes*) and $E \subseteq V^* \times \Sigma \times V$ is a finite set (of *hyperedges*) such that for every $(v_1 \ldots v_k, \sigma, v) \in E$ we have that $\sigma \in \Sigma^{(k)}$.[3] We interpret E as a ranked alphabet where the rank of each edge is carried over from its label in Σ. The family $(H_v \mid v \in V)$ of *derivations of* H is the smallest family $(P_v \mid v \in V)$ of subsets of T_E such that $e(P_{v_1}, \ldots, P_{v_k}) \subseteq P_v$ for every $e = (v_1 \ldots v_k, \sigma, v) \in E$.

A Σ-hypergraph (V, E) is *cyclic* if there are hyperedges $(v_1^1 \ldots v_{k_1}^1, \sigma_1, v^1)$, …, $(v_1^l \ldots v_{k_l}^l, \sigma_l, v^l) \in E$ such that v^{j-1} occurs in $v_1^j \ldots v_{k_j}^j$ for every $j \in \{2, \ldots, l\}$ and v^l occurs in $v_1^1 \ldots v_{k_1}^1$. It is called *acyclic* if it is not cyclic.

Example 1 Consider the ranked alphabet $\Sigma = \Sigma^{(0)} \cup \Sigma^{(1)} \cup \Sigma^{(2)}$ with $\Sigma^{(0)} = \{\alpha, \beta\}$, $\Sigma^{(1)} = \{\gamma\}$, and $\Sigma^{(2)} = \{\sigma\}$, and the Σ-hypergraph $H = (V, E)$ where

- $V = \{0, 1\}$ and
- $E = \{(\varepsilon, \alpha, 1), (\varepsilon, \beta, 1), (1, \gamma, 1), (11, \sigma, 0), (1, \gamma, 0)\}$.

A graphical representation of this hypergraph is shown in Fig. 1. Note that this hypergraph is cyclic because of the edge $(1, \gamma, 1)$.

We indicate the derivations of H, assuming that $e_1, \ldots, e_5$ are the edges in E in the order given above:

[2]The term $\sigma(\xi_1, \ldots, \xi_k)$ is usually understood as a string composed of the symbol σ, an opening parenthesis, the string ξ_1, a comma, and so on.

[3]The hypergraphs defined here are essentially nondeterministic tree automata, where V is the set of states and E is the set of transitions.

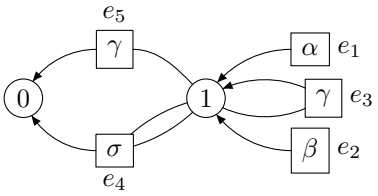

Figure 1: Hypergraph of Example 1.

- $H_1 = \{e_1, e_2, e_3(e_1), e_3(e_2), e_3(e_3(e_1)), \ldots\}$ and
- $H_0 = e_4(H_1, H_1) \cup e_5(H_1)$ where, e. g., $e_4(H_1, H_1)$ is the top-concatenation of H_1, H_1 with e_4, and thus

$$e_4(H_1,H_1) = \{e_4(e_1, e_1), e_4(e_1, e_2), e_4(e_1, e_3(e_1)), e_4(e_3(e_1), e_1), \ldots\}\,.$$

Next we give an example of ambiguity in hypergraphs with labeled hyperedges. Suppose that E contains an additional hyperedge $e_6 = (0, \gamma, 0)$. Then H_0 would contain the derivations $e_6(e_5(e_1))$ and $e_5(e_3(e_1))$, which describe the same Σ-tree, viz. $\gamma(\gamma(\alpha))$ (obtained by the node-wise projection to the second component). □

In the sequel, let $H = (V, E)$ be a Σ-hypergraph.

Ordering Usually an ordering is induced on the set of derivations by means of probabilities or, more generally, weights. In the following, we will abstract from the weights by using a binary relation $\precsim$ *directly* on derivations, where we will interpret the fact $\xi_1 \precsim \xi_2$ as "ξ_1 is better than or equal to ξ_2".

Example 2 (Ex. 1 contd.) First we show how an ordering is induced on derivations by means of weights. To this end, we associate an operation over the set $\mathbb{R}$ of reals with every hyperedge (respecting its arity) by means of a mapping θ:

$$\theta(e_1)() = 4 \qquad \theta(e_2)() = 3$$
$$\theta(e_3)(x_1) = x_1 + 1 \qquad \theta(e_4)(x_1, x_2) = x_1 + x_2$$
$$\theta(e_5)(x_1) = x_1 + 0.5$$

The weight $h(\xi)$ of a tree $\xi \in T_E$ is obtained by interpreting the symbols at each node using θ, e. g. $h(e_3(e_2)) = \theta(e_3)(\theta(e_2)()) = \theta(e_2)() + 1 = 4$.

Then the natural order $\leq$ on $\mathbb{R}$ induces the binary relation $\precsim$ over T_E as follows: for every $\xi_1, \xi_2 \in T_E$ we let $\xi_1 \precsim \xi_2$ iff $h(\xi_1) \leq h(\xi_2)$, meaning that trees with smaller weights are considered better. (This is, e. g., the case when calculating probabilites in the image of $-\log x$.) Note

that we could just as well have defined $\precsim$ with the inverted order.

Since addition is commutative, we obtain for every $\xi_1, \xi_2 \in T_E$ that $h(e_4(\xi_1, \xi_2)) = h(e_4(\xi_2, \xi_1))$ and thus $e_4(\xi_1, \xi_2) \precsim e_4(\xi_2, \xi_1)$ and vice versa. Thus, for two different trees ($e_4(\xi_1, \xi_2)$ and $e_4(\xi_2, \xi_1)$) having the same weight, $\precsim$ should not prefer any of them. That is, $\precsim$ need not be antisymmetric.

As another example, the mapping θ could assign to each symbol an operation over real-valued vectors, where each component represents one feature of a log-linear model such as frequencies, probabilities, reals, etc. Then the ordering could be defined by means of a linear combination of the feature weights. □

We use the concept of a linear pre-order to capture the orderings which are obtained this way.

Let S be a set. A *pre-order (on S)* is a binary relation $\precsim \subseteq S \times S$ such that (i) $s \precsim s$ for every $s \in S$ (*reflexivity*) and (ii) $s_1 \precsim s_2$ and $s_2 \precsim s_3$ implies $s_1 \precsim s_3$ for every $s_1, s_2, s_3 \in S$ (*transitivity*). A pre-order $\precsim$ is called *linear* if $s_1 \precsim s_2$ or $s_2 \precsim s_1$ for every $s_1, s_2 \in S$. For instance, the binary relation $\precsim$ on T_E as defined in Ex. 2 is a linear pre-order.

We will restrict our considerations to a class of linear pre-orders which admit efficient algorithms. For this, we will always assume a linear pre-order $\precsim$ with the following two properties (cf. Knuth (1977)).[4]

SP (*subtree property*) For every $e(\xi_1, \ldots, \xi_k) \in T_E$ and $i \in \{1, \ldots, k\}$ we have $\xi_i \precsim e(\xi_1, \ldots, \xi_k)$.[5]

CP (*compatibility*) For every pair $e(\xi_1, \ldots, \xi_k)$, $e(\xi'_1, \ldots, \xi'_k) \in T_E$ with $\xi_1 \precsim \xi'_1$, …, $\xi_k \precsim \xi'_k$ we have that $e(\xi_1, \ldots, \xi_k) \precsim e(\xi'_1, \ldots, \xi'_k)$.

It is easy to verify that the linear pre-order $\precsim$ of Ex. 2 has the aforementioned properties.

In the sequel, let $\precsim$ be a linear pre-order on T_E fulfilling SP and CP.

[4] Originally, these properties were called "superiority" and "monotonicity" because they were viewed as properties of the weight functions. We use the terms "subtree property" and "compatibility" respectively, because we view them as properties of the linear pre-order.

[5] This strong property is used here for simplicity. It suffices to require that for every $v \in V$ and pair $\xi, \xi' \in H_v$ we have $\xi \precsim \xi'$ if ξ is a subtree of ξ'.

Before we state the n-best-derivations problem formally, we define the operation $\min_n$, which maps every subset T of T_E to the set of all sequences of n best elements of T. To this end, let $T \subseteq T_E$ and $n \leq |T|$. We define $\min_n(T)$ to be the set of all sequences $(\xi_1, \ldots, \xi_n) \in T^n$ of pairwise distinct elements such that $\xi_1 \precsim \ldots \precsim \xi_n$ and for every $\xi \in T \setminus \{\xi_1, \ldots, \xi_k\}$ we have $\xi_n \precsim \xi$. For every $n > |T|$ we set $\min_n(T) = \min_{|T|}(T)$. In addition, we set $\min_{\leq n}(T) = \bigcup_{i=0}^{n} \min_i(T)$.

n-best-derivations problem The *n-best-derivations problem* amounts to the following.

Given a Σ-hypergraph $H = (V, E)$, a vertex $v \in V$, and a linear pre-order $\precsim$ on T_E fulfilling SP and CP,

compute an element of $\min_n(H_v)$.

3 Functional Programming

We will describe our main algorithm as a functional program. In essence, such a program is a system of (recursive) equations that defines several functions (as shown in Fig. 2). As a consequence the main computational paradigm for evaluating the application `(f a)` of a function `f` to an argument `a` is to choose an appropriate defining equation `f x = r` and then evaluate `(f a)` to `r'` which is obtained from `r` by substituting every occurrence of `x` by `a`.

We assume a *lazy* (and in particular, *call-by-need*) evaluation strategy, as in the functional programming language HASKELL. Roughly speaking, this amounts to evaluating the arguments of a function only as needed to evaluate the its body (i. e. for branching). If an argument occurs multiple times in the body, it is evaluated only once.

We use HASKELL notation and functions for dealing with lists, i. e. we denote the empty list by `[]` and list construction by `x:xs` (where an element `x` is prepended to a list `xs`), and we use the functions `head` (line 01), `tail` (line 02), and `take` (lines 03 and 04), which return the first element in a list, a list without its first element, and a prefix of a list, respectively.

In fact, the functions shown in Fig. 2 will be used in our main algorithm (cf. Fig. 4). Thus, we explain the functions `merge` (lines 05–07) and $e(\mathtt{l}_1, \ldots, \mathtt{l}_k)$ (lines 08–10) a bit more in detail.

The `merge` function takes a set $\mathcal{L}$ of pairwise disjoint lists of derivations, each one in ascending order with respect to $\precsim$, and merges them into

-- standard Haskell functions: list deconstructors, take operation

```
01  head (x:xs) = x
02  tail (x:xs) = xs
03  take n xs = []      if n == 0 or xs == []
04  take n xs = (head xs):take (n-1) (tail xs)
```

-- merge operation (lists in $\mathcal{L}$ should be disjoint)

```
05  merge L = []        if L \ {[]} = ∅
06  merge L = m:merge ({tail l | l ∈ L, l != [], head l == m} ∪
                       {l | l ∈ L, l != [], head l != m})
07        where m = min{head l | l ∈ L, l != []}
```

-- top concatenation

```
08  e(l1,...,lk) = []      if li == [] for some i ∈ {1,...,k}
09  e(l1,...,lk) = e(head l1,...,head lk):merge {e(l1^i,...,lk^i) | i ∈ {1,...,k}}
10        where lj^i = { lj          if j < i
                       { tail lj     if j = i
                       { [head lj]   if j > i
```

Figure 2: Some useful functions specified in a functional programming style.

one list with the same property (as known from the merge sort algorithm).

Note that the minimum used in line 07 is based on the linear pre-order $\precsim$. For this reason, it need not be uniquely determined. However, in an implementation this function is deterministic, depending on the the data structures.

The function $e(\mathtt{l}_1,\ldots,\mathtt{l}_k)$ implements the top-concatenation with e on lists of derivations. It is defined for every $e = (v_1 \ldots v_k, \sigma, v) \in E$ and takes lists $\mathtt{l}_1$, ..., $\mathtt{l}_k$ of derivations, each in ascending order as for `merge`. The resulting list is also in ascending order.

4 Algorithm

In this section, we develop our algorithm for solving the n-best-derivations problem. We begin by motivating our general approach, which amounts to solving the 1-best-derivation problem first and then extending that solution to a solution of the n-best-derivations problem.

It can be shown that for every $m \geq n$, the set $\min_n(H_v)$ is equal to the set of all prefixes of length n of elements of $\min_m(H_v)$. According to this observation, we will develop a function `p` mapping every $v \in V$ to a (possibly infinite) list such that the prefix of length n is in $\min_n(H_v)$ for every n. Then, by virtue of lazy evaluation, a solution to the n-best-derivations problem can be obtained by evaluating the term

```
take n (p v)
```

where `take` is specified in lines 03–04 of Fig. 2. Thus, what is left to be done is to specify `p` appropriately.

4.1 A provisional specification of `p`

Consider the following provisional specification of `p`:

$$\mathtt{p\ v} = \mathtt{merge}\ \{e(\mathtt{p\ v}_1,\ldots,\mathtt{p\ v}_k) \mid e = (v_1 \ldots v_k, \sigma, v) \in E\} \quad (\dagger)$$

where the functions `merge` and $e(\mathtt{l}_1,\ldots,\mathtt{l}_k)$ are specified in lines 05–07 and lines 08–10 of Fig. 2, respectively. This specification models exactly the trivial equation

$$H_v = \bigcup_{e=(v_1 \ldots v_k,\sigma,v)\in E} e(H_{v_1},\ldots,H_{v_k})$$

for every $v \in V$, where the union and the top-concatenation have been implemented for lists via the functions `merge` and $e(\mathtt{l}_1,\ldots,\mathtt{l}_k)$.

This specification is adequate if H is acyclic. For cyclic hypergraphs however, it can not even solve the 1-best-derivation problem. To illustrate this, we consider the hypergraph of Ex. 2 and cal-

culate[6]

$$
\begin{aligned}
&\texttt{take 1 (p 1)}\\
&= \texttt{(head (p 1)):take 0 (tail (p 1))} && (04)\\
&= \texttt{head (p 1)} && (03)\\
&= \texttt{head (merge } \{e_1(), e_2(), e_3\texttt{(p 1)}\}\texttt{)} && (\dagger)\\
&= \min\{\texttt{head } e_1(), \texttt{head } e_2(), \texttt{head } e_3\texttt{(p 1)}\} && (01, 06, 07)\\
&= \min\{\texttt{head } e_1(), \texttt{head } e_2(),\ e_3\texttt{(head (p 1))}\}. && (09)
\end{aligned}
$$

Note that the infinite regress occurs because the computation of the head element `head (p 1)` depends on itself. This leads us to the idea of "pulling" this head element (which is the solution to the 1-best-derivation problem) "out" of the merge in (†). Applying this idea to our particular example, we reach the following equation for `p 1`:

$$\texttt{p 1} = e_2\texttt{:}\quad \texttt{merge } \{e_1(), e_3\texttt{(p 1)}\}$$

because e_2 is the best derivation in H_1. Then, in order to evaluate `merge` we have to compute

$$
\begin{aligned}
&\min\{\texttt{head } e_1(), \texttt{head } e_3\texttt{(p 1)}\}\\
&= \min\{e_1,\ e_3\texttt{(head (p 1))}\}\\
&= \min\{e_1,\ e_3(e_2)\}\,.
\end{aligned}
$$

Since $h(e_1) = h(e_3(e_2)) = 4$, we can choose any of them, say e_1, and continue:

$$
\begin{aligned}
&e_2\texttt{:}\quad \texttt{merge } \{e_1(), e_3\texttt{(p 1)}\}\\
&= e_2\texttt{:}\quad e_1\texttt{:}\quad \texttt{merge } \{\texttt{tail } e_1(),\ e_3\texttt{(p 1)}\}\\
&= e_2\texttt{:}\quad e_1\texttt{:}\quad e_3(e_2)\texttt{:}\quad \texttt{merge } \{\texttt{tail } e_3\texttt{(p 1)}\}\\
&= \ldots
\end{aligned}
$$

Generalizing this example, the function `p` could be specified as follows:

$$\texttt{p 1 = (b 1) :}\quad \texttt{merge } \{exp\} \qquad (\dagger\dagger)$$

where `b 1` evaluates the 1-best derivation in H_1 and exp "somehow" calculates the next best derivations. In the following subsection, we elaborate this approach. First, we develop an algorithm for solving the 1-best-derivation problem.

4.2 Solving the 1-best-derivation problem

Using SP and CP, it can be shown that for every $v \in V$ such that $H_v \neq \emptyset$ there is a minimal derivation in H_v which does not contain any subderivation in H_v (apart from itself). In other words, it is not necessary to consider cycles when solving the 1-best-derivation problem.

[6]Please note that $e_1()$ is an application of the function in lines 08–10 of Fig. 2 while e_1 is a derivation.

We can exploit this knowledge in a program by keeping a set `U` of visited nodes, taking care not to consider edges which lead us back to those nodes. Consider the following function:

$$
\begin{aligned}
\texttt{b v U} = \min\{&e(\texttt{b } v_1\ \texttt{U'}, \ldots,\ \texttt{b } v_k\ \texttt{U'}) \mid\\
&e = (v_1 \ldots v_k, \sigma, v) \in E,\\
&\{v_1, \ldots, v_k\} \cap \texttt{U'} = \emptyset\}
\end{aligned}
$$

where $\texttt{U'} = \texttt{U} \cup \{\texttt{v}\}$

The argument `U` is the set of visited nodes. The term `b v` $\emptyset$ evaluates to a minimal element of H_v, or to $\min \emptyset$ if $H_v = \emptyset$. The problem of this divide-and-conquer (or top-down) approach is that managing a separate set `U` for every recursive call incurs a big overhead in the computation.

This overhead can be avoided by using a dynamic programming (or bottom-up) approach where each node is visited only once, and nodes are visited in the order of their respective best derivations.

To be more precise, we maintain a family $(P_v \mid v \in V)$ of already found best derivations (where $P_v \in \min_{\leq 1}(H_v)$ and initially empty) and a set C of candidate derivations, where candidates for all vertices are considered at the same time. In each iteration, a minimal candidate with respect to $\precsim$ is selected. This candidate is then declared the best derivation of its respective node.

The following lemma shows that the bottom-up approach is sound.

Lemma 3 *Let $(P_v \mid v \in V)$ be a family such that $P_v \in \min_{\leq 1}(H_v)$. We define*

$$C = \bigcup_{\substack{e=(v_1 \ldots v_k, \sigma, v) \in E,\\ P_v = \emptyset}} e(P_{v_1}, \ldots, P_{v_k})\ .$$

Then (i) for every $\xi \in \bigcup_{v \in V, P_v = \emptyset} H_v$ there is a $\xi' \in C$ such that $\xi' \preceq \xi$, and (ii) for every $v \in V$ and $\xi \in C \cap H_v$ the following implication holds: if $\xi \leq \xi'$ for every $\xi' \in C$, then $\xi \in \min_1(H_v)$.

An algorithm based on this lemma is shown in Fig. 3. Its key function `iter` uses the notion of accumulating parameters. The parameter `q` is a mapping corresponding to the family $(P_v \mid v \in V)$ of the lemma, i. e., `q v` $= P_v$; the parameter `c` is a set corresponding to C. We begin in line 01 with the function `q0` mapping every vertex to the empty list. According to the lemma, the candidates then consist of the nullary edges.

As long as there are candidates left (line 04), in a recursive call of `iter` the parameter `q` is updated with the newly found pair $(v, [\xi])$ of vertex v and (list of) best derivation ξ (expressed by

Require Σ-hypergraph $H = (V, E)$, linear preorder $\precsim$ fulfilling SP and CP.
Ensure `b v` $\in \min_1(H_v)$ for every $v \in V$ such that if `b v == [`$e(\xi_1, \ldots, \xi_k)$`]` for some $e = (v_1 \ldots v_k, \sigma, v) \in E$, then `b` v_i `== [`ξ_i`]` for every $i \in \{1, \ldots, k\}$.

01 `b = iter q0` $\{(\varepsilon, \alpha, v) \in E \mid \alpha \in \Sigma^{(0)}\}$
02 `q0 v = []`
03 `iter q` $\emptyset$ `= q`
04 `iter q c = iter (q//(v,[`ξ`])) c'`
05 where
06 ξ `=` $\min$ `c` and $\xi \in H_v$
07 `c' =` $\bigcup_{\substack{e=(v_1 \ldots v_k, \sigma, v) \in E \\ \texttt{q v == []}}} e($`q` $v_1, \ldots,$`q` $v_k)$

Figure 3: Algorithm solving the 1-best-derivation problem.

`q//(v,[`ξ`])`) and the candidate set is recomputed accordingly. When the candidate set is exhausted (line 03), then `q` is returned.

Correctness and completeness of the algorithm follow from Statements (ii) and (i) of Lemma 3, respectively. Now we show termination. In every iteration a new next best derivation is determined and the candidate set is recomputed. This set only contains candidates for vertices $v \in V$ such that `q v == []`. Hence, after at most $|V|$ iterations the candidates must be depleted, and the algorithm terminates.

We note that the algorithm is very similar to that of Knuth (1977). However, in contrast to the latter, (i) it admits $H_v = \emptyset$ for some $v \in V$ and (ii) it computes some minimal derivation instead of the weight of some minimal derivation.

Runtime According to the literature, the runtime of Knuth's algorithm is in $O(|E| \cdot \log|V|)$ (Knuth, 1977). This statement relies on a number of optimizations which are beyond our scope. We just sketch two optimizations: (i) the candidate set can be implemented in a way which admits obtaining its minimum in $O(\log|C|)$, and (ii) for the computation of candidates, each edge needs to be considered only once during the whole run of the algorithm.

4.3 Solving the n-best-derivations problem

Being able to solve the 1-best-derivation problem, we can now refine our specification of `p`. The refined algorithm is given in Fig. 4; for the functions not given there, please refer to Fig. 3 (function `b`) and to Fig. 2 (functions `merge`, `tail`, and the top-concatenation). In particular, line 02 of Fig. 4 shows the general way of "pulling out" the head element as it was indicated in Section 4.1 via an example. We also remark that the definition of the top-concatenation (lines 08–09 of Fig. 2) corresponds to the way in which $\mathrm{mult}_{\preceq k}$ was sped up in Fig. 4 of (Huang and Chiang, 2005).

Theorem 4 *The algorithm in Fig. 4 is correct with respect to its require/ensure specification and it terminates for every input.*

PROOF (SKETCH). We indicate how induction on n can be used for the proof. If $n = 0$, then the statement is trivially true. Let $n > 0$. If `b v == []`, then the statement is trivially true as well. Now we consider the converse case. To this end, we use the following three auxiliary statements.

(1) `take n (merge {`$l_1, \ldots,$`l`$_k$`}) = take n (merge {take n l`$_1, \ldots,$`take n l`$_k$`})`,

(2) `take n` $e($`l`$_1, \ldots,$`l`$_k)$ `= take n` $e($`take n l`$_1, \ldots,$`take n l`$_k)$,

(3) `take n (tail l) = tail (take (n+1) l)`.

Using these statements, line 04 of Fig. 2, and line 02 of Fig. 4, we are able to "pull" the `take` of `take n (p v)` "into" the right-hand side of `p v`, ultimately yielding terms of the form `take n (p` v_j`)` in the first line of the `merge` application and `take (n-1) (p` v'_j`)` in the second one.

Then we can show the following statement by induction on m (note that the n is still fixed from the outer induction): for every $m \in \mathbb{N}$ we have that if the tree in `b v` has at most height m, then `take n (p v)` $\in \min_n(H_v)$. To this end, we use the following two auxiliary statements.

(4) For every sequence of pairwise disjoint subsets $P_1, \ldots, P_k \subseteq \bigcup_{v \in V} H_v$, sequence of natural numbers $n_1, \ldots, n_k \in \mathbb{N}$, and lists $l_1 \in \min_{n_1}(P_1)$, …, $l_k \in \min_{n_k}(P_k)$ such that $n_j \geq n$ for every $j \in \{1, \ldots, k\}$ we have that `take n (merge {`$l_1, \ldots, l_k$`})` $\in \min_n(P_1 \cup \ldots \cup P_k)$.

(5) For every edge $e = (v_1 \ldots v_k, \sigma, v) \in E$, subsets $P_1, \ldots, P_k \subseteq \bigcup_{v \in V} H_v$, and lists $l_1 \in \min_n(P_1)$, …, $l_k \in \min_n(P_k)$ we have that `take n` $e(l_1, \ldots, l_k) \in \min_n(e(P_1, \ldots, P_k))$.

Using these statements, it remains to show that $\{e(\xi_1, \ldots, \xi_k)\} \circ \min_{n-1}\big((e(H_{v_1}, \ldots, H_{v_k}) \setminus \{e(\xi_1, \ldots, \xi_k)\}) \cup \bigcup_{e' \neq e} e'(H_{v'_1}, \ldots, H_{v'_k})\big) \subseteq \min_n(H_v)$ where `b v` $=$ `[`$e(\xi_1, \ldots, \xi_k)$`]` and $\circ$ denotes language concatenation. This can be shown by using the definition of $\min_n$.

Termination of the algorithm now follows from the fact that every finite prefix of `p v` is well defined. ∎

Require Σ-hypergraph $H = (V, E)$, linear pre-order $\precsim$ fulfilling SP and CP.
Ensure $\big($`take n (p v)`$\big) \in \min_n(H_v)$ for every $v \in V$ and $n \in \mathbb{N}$.

```
01  p v = []      if b v == []
02  p v = e(ξ1,...,ξk):merge ({tail e(p v1,...,p vk) | e = (v1...vk, σ, v) ∈ E} ∪
                             {e'(p v'1,...,p v'k) | e' = (v'1...v'k, σ', v) ∈ E, e' ≠ e})
                  if b v == [e(ξ1,...,ξk)]
```

Figure 4: Algorithm solving the n-best-derivations problem.

4.4 Implementation, Complexity, and Experiments

We have implemented the algorithm (consisting of Figs. 3 and 4 and the auxiliary functions of Fig. 2) in HASKELL. The implementation is rather straightforward except for the following three points.

(1) Weights: we assume that $\precsim$ is defined by means of weights (cf. Ex. 2), and that comparing these weights is in $O(1)$ (which often holds because of limited precision). Hence, we store with each derivation its weight so that comparison according to $\precsim$ is in $O(1)$ as well.

(2) Memoization: we use a memoization technique to ensure that no derivation occurring in `p v` is computed twice.

(3) Merge: the `merge` operation deserves some consideration because it is used in a nested fashion, yielding trees of `merge` applications. This leads to an undesirable runtime complexity because these trees need not be balanced. Thus, instead of actually computing the `merge` in `p` and in the top-concatenation, we just return a data structure describing what should be merged. That data structure consists of a best element and a list of lists of derivations to be merged (cf. lines 06 and 09 in Fig. 2). We use a higher-order function to manage these data structures on a heap, performing the merge in a nonnested way.

Runtime Here we consider the n-best part of the algorithm, i. e. we assume the computation of the mapping `b` to take constant time. Note however that due to memoization, `b` is only computed once. Then the runtime complexity of our implementation is in $O\big(|E| + |V| \cdot n \cdot \log(|E| + n)\big)$. This can be seen as follows.

By line 02 in Fig. 4, the initial heaps in the higher-order merge described under (3) have a total of $|E|$ elements. Building these heaps is thus in $O(|E|)$. By line 09 in Fig. 2, each newly found derivation spawns at most as many new candidates on the heap as the maximum rank in Σ. We assume this to be constant. Moreover, at most n derivations are computed for each node, that is, at most $|V| \cdot n$ in total. Hence, the size of the heap of a node is in $O(|E| + n)$. For each derivation we compute, we have to pop the minimal element off the heap (cf. line 07 in Fig. 2), which is in $O\big(\log(|E| + n)\big)$, and we have to compute the union of the remaining heap with the newly spawned candidates, which has the same complexity.

We give another estimate for the total number of derivations computed by the algorithm, which is based on the following observation. When popping a new derivation ξ off the heap, new next best candidates are computed. This involves computing at most as many new derivations as the number of nodes of ξ, because for each hyperedge occurring in ξ we have to consider the next best alternative. Since we pop off at most n elements from the heap belonging to the target node, we arrive at the estimate $d \cdot n$, where d is the size of the biggest derivation of said node.

A slight improvement of the runtime complexity can be obtained by restricting the heap size to n best elements, as argued by Huang and Chiang (2005). This way, they are able to obtain the complexity $O(|E| + d \cdot n \cdot \log n)$.

We have conducted experiments on an Intel Core Duo 1200 MHz with 2 GB of RAM using a cyclic hypergraph containing 671 vertices and 12136 edges. The results are shown in Table 1. This table indicates that the runtime of the n-best part is roughly linear in n.

n	total time [s]	time for n-best part [s]
1	8.713	—
25 000	10.832	2.119
50 000	12.815	4.102
100 000	16.542	7.739
200 000	24.216	15.503

Table 1: Experimental results

References

Athanasios Alexandrakis and Symeon Bozapalidis. 1987. Weighted grammars and Kleene's theorem. *Inform. Process. Lett.*, 24(1):1–4.

Jean Berstel and Christophe Reutenauer. 1982. Recognizable formal power series on trees. *Theoret. Comput. Sci.*, 18(2):115–148.

Matthias Büchse, Jonathan May, and Heiko Vogler. 2009. Determinization of weighted tree automata using factorizations. Talk presented at FSMNLP 09 in Pretoria, South Africa.

Zoltán Ésik and Werner Kuich. 2003. Formal tree series. *J. Autom. Lang. Comb.*, 8(2):219–285.

Zoltán Fülöp and Heiko Vogler. 2009. Weighted tree automata and tree transducers. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, chapter 9. Springer.

Joshua Goodman. 1999. Semiring parsing. *Comp. Ling.*, 25(4):573–605.

Liang Huang and David Chiang. 2005. Better k-best parsing. In *Parsing '05: Proceedings of the Ninth International Workshop on Parsing Technology*, pages 53–64. ACL.

Dan Klein and Christopher D. Manning. 2001. Parsing and hypergraphs. In *Proceedings of IWPT*, pages 123–134.

Donald E. Knuth. 1977. A Generalization of Dijkstra's Algorithm. *Inform. Process. Lett.*, 6(1):1–5, February.

Zhifei Li, Jason Eisner, and Sanjeev Khudanpur. 2009. Variational decoding for statistical machine translation. In *Proc. ACL-IJCNLP '09*, pages 593–601. ACL.

Andreas Maletti and Giorgio Satta. 2009. Parsing algorithms based on tree automata. In *Proc. 11th Int. Conf. Parsing Technologies*, pages 1–12. ACL.

Jonathan May and Kevin Knight. 2006. A better n-best list: practical determinization of weighted finite tree automata. In *Proc. HLT*, pages 351–358. ACL.

Mark-Jan Nederhof. 2003. Weighted deductive parsing and Knuth's algorithm. *Comp. Ling.*, 29(1):135–143.

Lars Relund Nielsen, Kim Allan Andersen, and Daniele Pretolani. 2005. Finding the k shortest hyperpaths. *Comput. Oper. Res.*, 32(6):1477–1497.

Franz Josef Och and Hermann Ney. 2002. Discriminative training and maximum entropy models for statistical machine translation. In *ACL*, pages 295–302.

Adam Pauls and Dan Klein. 2009. k-best a* parsing. In *Proc. ACL-IJCNLP '09*, pages 958–966, Morristown, NJ, USA. ACL.

J. W. Thatcher. 1967. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *J. Comput. Syst. Sci.*, 1(4):317–322.