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

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

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

Abstract

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

1. Introduction

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

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

[1]Declarations of interest: none.

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

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

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

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

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

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

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

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

2. Preliminaries

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

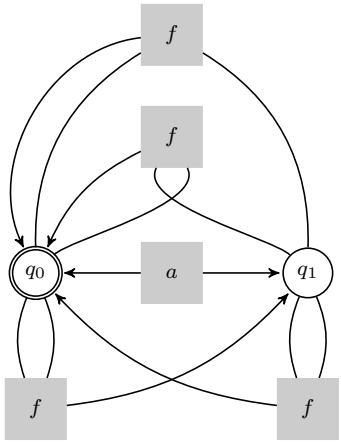
Example 2.1. *Figure 1 shows an example wta, and Table 1 contains a side-by-side comparison of the input trees of the* 10 *best runs and the* 10 *best trees of the example wta. Since the weight of every transition rules is* 1*, the weight of every run on an input tree t is its size $|t|$. Looking at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

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

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

3. Algorithm

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



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

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

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

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

3.1. Best Trees

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2. Computation of Best Contexts

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

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

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

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

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

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

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

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

  end while
end procedure

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3. Transition-Based Best Trees

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

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

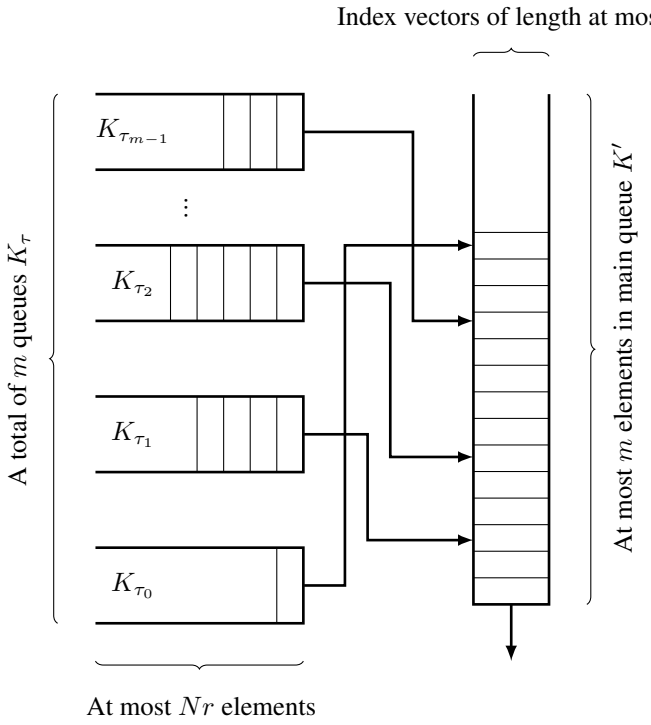


Figure 2: The algorithm keeps a dedicated priority queue K_τ for each transition rule τ, and the collection of these queues is itself arranged in a main priority queue K' (drawn on the right).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4. Correctness

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We finally show that Algorithm 2 is correct as well.

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

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

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

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

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

3.5. Time complexity

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

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

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

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

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

Theorem 3.6. *Algorithm 2 runs in time*

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

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

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

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

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

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

as claimed.

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

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

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

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

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

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

4. Conclusion

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

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

Acknowledgements

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

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

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

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

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

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

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

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

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

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

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

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

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

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