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 can be made more efficient by changing how it arranges and prunes the explored search space. Given an integer N and a weighted tree automaton (wta) M over the tropical semiring, the algorithm computes N trees of minimal total weight with respect to M. Compared to the original algorithm, the modifications increase both the effectiveness of pruning and 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. Given a probabilistic context-free grammar (pcfg) G and a string w, we can construct a weighted tree automaton (wta) that assigns a probability to each parse tree of w with respect to G. That way, the computation of the N most likely parses for w with respect to G is reduced to an instance of the N best trees problem for wta.

Socher et al. [1] 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, [2]), and underlines the importance of the N-best problem in the context of language processing.

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 language recognised by M. The N-best trees problem is similar to that

[1]Declarations of interest: none.

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.

The computational difficulty of the N-best problem therefore 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 [3]. 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 [4], Björklund et al. generalised an N-best algorithm by Mohri and Riley [5] 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 bounded from above by a low polynomial in N, m, n, and r, 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 [6], a comparative study of our algorithm and an algorithm by Huang and Chiang that extracts N best runs, which is implemented in the Tiburon wta toolkit [7] and taken to represent the state of the art. Tiburon can thus be used to compute N best trees indirectly by first computing a large enough set of best runs and then removing duplicate trees. Our conclusion 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 improve the algorithm in [4] by organizing 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 [4]. 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.

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 $|S|$, and the powerset of S is denoted by $pow(S)$. The n-fold Cartesian product of a set S with itself is denoted by S^n.

For a set S and a mapping $\Gamma\colon S \to \mathbb{R}_+^\infty$ we denote by $\min_\Gamma S$ the minimal elements of S with respect to Γ, i.e. $\min_\Gamma S = \{s \in S \mid \Gamma(s) \leq \Gamma(s') \text{ for all } s' \in S\}$.

As usual, the set of all finite sequences over S is denoted by S^*, and the empty sequence by λ. Given a sequence of distinct elements $S = s_1, \ldots, s_n$ of n elements and an integer $i \in [n]$, we write S_i for the i:th element of S, that is, for s_i. Simlarly, we write $index(S, s_i)$ for the index of s_i in S, that is, for i.

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 $t(v)$. We generally assume that T_Σ is infinite, i.e., that there exists a $k > 0$ such that $\Sigma_{(0)} \neq \emptyset \neq \Sigma_{(k)}$. For a set T of trees we denote by $\Sigma(T)$ the set of trees which have a symbol from Σ in their root, with direct subtrees in T, 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 $\Box \notin \Sigma$ be a special symbol of rank 0. The set of *contexts over* Σ is the set C_Σ of trees $c \in T_{\Sigma\cup\{\Box\}}$ containing exactly one node $v \in dom(c)$ with $c(v) = \Box$. 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

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

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

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

- a finite set Q of symbols of rank 0 called *states*;
- a ranked alphabet Σ of *input symbols* disjoint with Q;
- a finite set R of *transition rules* in the form $f[q_1, \ldots, q_k] \xrightarrow{w} q$, where we have $q, q_1, \ldots, q_k \in Q$, $f \in \Sigma_{(k)}$, and $w \in \mathbb{R}_+^\infty$; and

- a set $Q_{\mathrm{f}} \subseteq Q$ of *final states*.

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 of τ, that is, q, 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 $runs_M^q \subseteq T_{R \cup Q}$ of *q-runs of M*, their resulting trees $result_M(\rho)$, and their weights $wt_M(\rho)$ (for $\rho \in runs_M^q$) inductively, as follows:

1. For every $q \in Q$, we have $q \in runs_M^q$ with $result_M(q) = q$ and $wt_M(q) = 0$.
2. For every rule $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$ in R and all runs $\rho_1 \in runs_M^{q_1}, \ldots, \rho_k \in runs_M^{q_k}$, we have the run $\rho = \tau[\rho_1, \ldots, \rho_k] \in runs_M^q$ with the resulting tree $result_M(\rho) = f[result_M(\rho_1), \ldots, result_M(\rho_k)]$, and the weight $wt_M(\rho) = w + \sum_{i \in [k]} wt_M(\rho_i)$.

The set of *accepting* runs of M is the set $runs_M = \{runs_M^q \mid q \in F\}$.

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

$$M(t) = \min\{wt_M(\rho) \mid \rho \in runs_M^q \text{ is accepting and } result_M(\rho) = t\}$$

for all $t \in T_\Sigma$ (where, by convention, $\min \emptyset = \infty$). In other words, $M(t)$ is the minimal weight of an accepting run of t. Note that we, by a slight abuse of notation, denote by M both the wta and the weighted tree language it computes.

Given a wta M and an integer $N \in \mathbb{N}^\infty$, the *N-best runs problem* consists in computing a sequence of N accepting runs of minimal weight according to M. More precisely, an algorithm solving the problem will output a sequence $\rho_1, \rho_2, \ldots$ of N pairwise distinct accepting runs such that there do not exist $i \in [N]$ and $\rho \in runs_M \setminus \{\rho_1, \ldots, \rho_i\}$ with $wt_M(\rho) < wt_M(\rho_i)$.

General assumption. To make sure that the N-best runs problem always possesses a solution, and also in order to avoid technical problems in our algorithms, we assume from now on that all considered wtas M have infinitely many accepting runs. Since T_Σ is infinite, this can be guaranteed by choosing (or adding) an accepting state q and symbols $a \in \Sigma_{(0)}, f \in \Sigma_{(k)}$ with $k > 0$ and adding the transition rules $a \xrightarrow{\infty} q$ and $f[q, \ldots, q] \xrightarrow{\infty} q$. Obviously, this modification does not affect the weighted tree language recognized by M.

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 $t_1, t_2, \ldots$ in T_Σ of minimal weight, in other words, a sequence of trees such that there do not exist $i \in [N]$ and $t \in T_\Sigma \setminus \{t_1, \ldots, t_i\}$ 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. Moreover, by the assumption above at least one such solution consists of the results of N accepting runs.

3. Algorithm

In this section, we explain how the algorithm in [4] can be made lazier, and hence more efficient, by pruning with respect to transitions rather than states. From here on, let $M = (Q, \Sigma, R, Q_{\mathrm{f}})$ be a wta over the tropical semiring with m transition rules, n states, and a maximum rank of r among its symbols.

3.1. Best Trees

We first recall the approach of [4] 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 of $M(c[\![t]\!])$, where c ranges over all possible contexts. Let M^q denote the wta obtained from M by replacing its set of final states by $\{q\}$. Then, for every context c and all trees t,

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

Since the expression $M(c[\![q]\!])$ is independent of t, it is possible to compute in advance a *cheapest context* c_q that minimises it [4].

Definition 1. *A* cheapest context *of a state $q \in Q$ is a context c_q with $M_q(c_q) = \min_{c \in C_\Sigma} M_q(c)$.*

The algorithm uses the cheapest contexts to compute an *optimal state* $opt(t)$ for each tree t that it encounters, such that $opt(t)$ is the state that minimises the weight of $c_q[\![t]\!]$, among all states $q \in Q$.

The priority queue K is initialised with the trees in Σ_0. Let

$$\Delta(s) = M(c_{opt(s)}[\![s]\!])$$

for all $s \in T_\Sigma$. Then the ordering $<_K$ of K is defined as follows, for all trees t and t' in K:

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

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 [4] in Algorithm 1. After outputting $i \in [N]$ trees, the set of trees enqueued in line 19 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 [4].

3.2. Transition-based pruning

The improved algorithm 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, refraining from assembling a new instantiation of a rule τ until all previous instantiations of τ have appeared as subtrees of outputted trees.

To this end, a separate priority queue is kept for each transition rule in R. These priority queues do not store the trees themselves directly. Instead, for a rule τ as above,

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

```
1: procedure BEST TREES V.1(M, N)
2:     compute cheapest contexts
3:     T ← ∅
4:     K ← ∅
5:
6:     for f ∈ Σ_0 do
7:         enq(K, f)
8:     end for
9:     c ← 0
10:    while c < N do
11:        t ← deq(K)
12:
13:        T ← T ∪ {t}
14:
15:        if M(t) = Δ(t) then
16:            output(t)
17:            c ← c + 1
18:        end if
19:        enq(K, expand(T, t))
20:    end while
21: end procedure
```

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

```
1: procedure BEST TREES V.2(M, N)
2:     compute cheapest contexts
3:     for q ∈ Q do
4:         T_q ← ∅
5:     end for
6:     for τ ∈ R do
7:         enq(K_τ, 1^rank(τ))
8:     end for
9:     c ← 0
10:    while c < N do
11:        (τ, u) ← deq({K_τ | τ ∈ R})
12:        if |T_tar(τ)| < N ∧ τ[u] ∉ T_tar(τ) then
13:            append τ[u] to T_tar(τ)
14:        end if
15:        if tar(τ) final ∧ τ[u] is new then
16:            output(τ[u])
17:            c ← c + 1
18:        end if
19:        enq(K_τ, inc(u))
20:    end while
21: end procedure
```

sequences of k-tuples of indices are stored, each index i_j at position j pointing the i_jth element of a queue that holds the 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 3.4, an efficient implementation of the algorithm can make this assembly "just in time", so as to avoid unnecessary work.

The transition-based algorithm is outlined in Algorithm 2. It maintains, for each state $q \in Q$, a sequence T_q of at most N trees in T_Σ. T_q is initially empty. The purpose of T_q is to list trees that reach the state q with minimal cost. 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 $\{K_\tau \mid \tau \in R\}$: For every $\tau \in R$, the queue K_τ contains index sequences in $\mathbb{N}^{rank(\tau)}$. These sequences represent trees as follows: Given a transition rule $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$ in R and an integer sequence $u \in \mathbb{N}^k$, we denote by $\tau[u]$ the tree $f[(T_{q_1})_{u_1}, \ldots, (T_{q_k})_{u_k}]$ if $u_i \leq |T_{q_i}|$ for every $i \in [k]$, and let $\tau[u]$ be undefined otherwise. Here, $(T_{q_i})_{u_i}$ denotes the u_i-th element of the sequence T_{q_i}.

The queue K_τ is a min-priority queue in which the priority of an element u is determined by the value $\Lambda_\tau(u)$. The latter is defined similarly to $\Delta(\tau[u])$, but using the weight of the rule τ at the root of $\tau[u]$. Hence, $\Lambda_\tau(u)$ may be greater than $\Delta(\tau[u])$ but is equal to it for some τ. Formally, for a rule $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$ and $u = (u_1, \ldots, u_k)$,

$\Lambda_\tau(u) = \top > \infty$ if $\tau[u]$ is undefined. Otherwise,

$$\Lambda_\tau(u) = M(c_q[\![q]\!]) + w + \sum_{i=1}^{k} M^{q_i}((T_{q_i})_{u_i}).$$

Initially, each queue K_τ, $\tau \in R$, contains the single element $1^{rank(\tau)}$. After initializing T_q, $q \in Q$, and K_τ, $\tau \in R$, 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 pair (τ, u) from the queues $\{K_\tau,\ \tau \in R\}$. The algorithm chooses this pair so that $\Lambda_\tau(\tau[u])$ is minimal across all K_τ. Thanks to our assumption that M possesses infinitely many (accepting) runs, it can be shown that $\Lambda_\tau(\tau[u]) \neq \top$, i.e., the tree $\tau[u]$ is defined. The tree is appended at the end of the list of the cheapest trees that reach the state $tar(\tau)$, the target state of the rule τ. If $tar(\tau)$ is final (i.e., the corresponding run is accepting) and $\tau[u]$ is "new", i.e., has not been outputted before, it is outputted, and the counter c tracking the number of outputted 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.

Finally, Algorithm 2 uses the function inc to record what rule instantiations to potentially construct later. The idea is that a next-best instantiation of the rule τ to $\tau[u]$ where $u = u_1 \cdots u_k$ can be found by increasing exactly one of u_i, $i \in [k]$, by 1.

Definition 2 (The function inc)**.** *The function* $inc(u) : \mathbb{N}^+ \to pow(\mathbb{N}^+)$ *maps every sequence* $u \in \mathbb{N}^k$ *to the set*

$$\{v_1 \cdots v_k \in \mathbb{N}^+ \mid \begin{array}{l} \exists i \in [k] : v_i = u_i + 1 \textit{ and} \\ \forall j \in [k] \setminus \{i\} : v_j = u_j\} \ . \end{array}$$

On line 11 of Algorithm 2, we use $\mathrm{deq}(\{K_\tau \mid \tau \in R\})$ as a shorthand for dequeuing a sequence u from one of the queues K_τ, $\tau \in R$, such that $\Lambda_\tau(u)$ is a minimal among all τ and u, again using a lexical order to break ties. As mentioned before, and shown in the next section, there always exist τ and u such that $\Lambda_\tau(u) \leq \infty$, i.e., $\tau[u]$ is defined.

3.3. Correctness

Let us now show that Algorithm 2 is correct. We begin with the observation that the weight of the cheapest tree that contains a given tree t as a subtree, is at least as large as the weight of the cheapest tree that contains as a subtree any subtree s of t:

Lemma 3. *For every* $s, t \in T_\Sigma$ *such that* s *is a subtree of* t*, we have* $\Delta(s) \leq \Delta(t)$.

Proof. Since s is a subtree of t, there is a context c_0 such that $t = c_0[\![s]\!]$. By the

definition of Δ

$$\begin{aligned}\Delta(t) &= \min_{c \in C_\Sigma} M(c[\![t]\!]) \\ &= \min_{c \in C_\Sigma} M(c[\![c_0[\![s]\!]]\!]) \\ &= \min_{c \in C_\Sigma} M(c[\![c_0]\!][\![s]\!]) \\ &\geq \min_{c \in C_\Sigma} M(c[\![s]\!]) \\ &= \Delta(s).\end{aligned}$$

□

Lemma 3 shows that it is possible to search for N best trees with respect to M by first considering trees of size one, and then looking at larger trees that have previously investigated smaller trees as their subtrees.

For $i \in [N]$, let $(\tau_1, u_1), \ldots, (\tau_i, u_i)$ be the sequence of elements dequeued (but not necessarily outputted) by Algorithm 2 during the first i iterations of the main loop (starting at line 10), and let $\Theta_i = \{\tau_1[u_1], \ldots, \tau_i[u_i]\}$.

Observation 4. *For every $i \in \mathbb{N}$, Θ_i is subtree closed, i.e., if $t \in \Theta_i$ then each of its subtrees is also in Θ_i.*

Lemma 5. *Consider K_τ ($\tau \in R$) at the point in time when Algorithm 2 has reached line 10 for the i-th time, and let*

$$K_{\tau,i} = \{u \in K_\tau \mid \tau[u] \text{ is defined and not in } \Theta_i\} \text{ and}$$
$$T = \{\tau[u] \mid \tau \in R, u \in \min_{\Lambda_\tau}(K_{\tau,i})\} .$$

Then $\min_\Delta(T_\Sigma \setminus \Theta_i) \cap T \neq \emptyset$.

Proof. Let $i \in [N]$ and $s \in \min_\Delta(T_\Sigma \setminus \Theta_i)$. If $s \in T$, then the assertion of the lemma holds. Otherwise, let $\rho \in runs_M^{opt(q)}$ be a run of minimal weight such that $result_M(\rho) = s$. Since $s \notin \Theta_i$, there is a $v \in dom(s)$, say with $\rho(v) = f[q_1, \ldots, q_k] \xrightarrow{w} q$ and $s/v = t = f[t_1, \ldots, t_k]$, such that $t_j \in T_{q_j}$, for every $j \in [k]$, but $t \notin T_q$. By Lemma 3 we have $\Delta(t) \leq \Delta(t)$, so $t \in \min_\Delta(T_\Sigma \setminus \Theta_i)$.

Now, let i_j be the position of t_j in T_{q_j} for all $j \in [k]$. Then there is a $u \in \mathbb{N}^k$ such that (i) $u_j \leq i_j$, for every $j \in [k]$, and (ii) u has been enqueued on K_q, but not yet dequeued. Such a u must exist, since for $k = 0$, K_ρ contains $()$ due to the initialization, and for $k > 0$ the function inc adds tuples in monotonically increasing order, which means that $K_{\rho(v)}$ is always non-empty. Moreover, we know that $\rho(v)[u]$ is defined, since $u_j \leq i_j$ and T_{q_i} contains at least i_j elements.

Suppose that there is a u' in K_τ such that $\Lambda_\tau(u') < \Lambda_\tau(u)$. Let s' be the tree obtained from s by substituting $\tau[u']$ for the subtree t. If $M(s') < M(s)$, then either $s' \in \Theta_i$, so from Observation 4 it follows that $\tau[u'] \in \Theta_i$, or $s' \notin \Theta_i$, in which case $t \notin \min_\Delta(T_\Sigma \setminus \Theta_i)$. □

Theorem 6. *Algorithm 2 computes a solution to the N-best problem.*

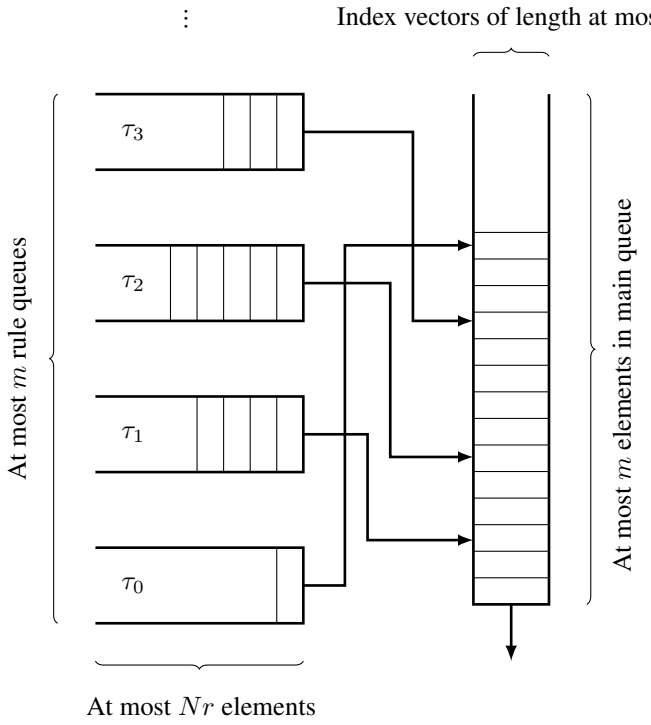


Figure 1: The algorithms keeps a dedicated priority queue for each transition rule (here named τ_i, $i \in \mathbb{N}$), and these queues feed elements in to a main priority queue (drawn on the right).

Proof. By Lemma 5 we know that when Algorithm 2 reaches line 10 for the i-th time, there is a $\tau \in R$ and and $u \in K_\tau$, such that $\tau[u]$ is defined, and $\tau[u]$ is only preceded in K_τ by elements in Θ_i.

Since no tuple is ever enqueued twice on the same queue K_τ (by the construction of inc), there can be at most m copies of each previously outputted tree among the $\tau[u]$ for which u still is to be dequeued from K_τ. Hence, after $m \cdot |\Theta_i|$ dequeueing operations, the algorithm is guaranteed to dequeue one of the trees in $\min_\Delta(T_\Sigma \setminus \Theta_i)$. □

3.4. Time complexity

Recall that the input $M = (Q, \Sigma, R, Q_\mathrm{f})$ 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 schematically illustrated in Figure 1. It uses priority queues based on heaps [8], which 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 $\Lambda_\tau(u) = \top$. Thus, $\Lambda_\tau(u)$ must be decreased from $\top$ 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 $\Lambda_\tau(u)$ (which is either still $\top$ or has reached its final value).

- We add an auxiliary priority queue K that contains pairs of elements (τ, u) such that $\tau \in R$ and u is the top priority element in K_τ. The priority of (τ, u) in K is simply $\Lambda_\tau(u)$. The advantage of K is that we avoid iterating over all queues K_τ, $\tau \in R$, every time Algorithm 2 reaches line 11. To make sure that the length of K does not exceed m, we keep a pointer for each τ to the record in K that represents the top priority element of K_τ. Whenever another element of K_τ turns out to have a higher priority than the one currently represented in K, the record is updated and the priority is adjusted accordingly.

With these implementation details in place, we are now ready to establish an upper bound on the running time of the algorithm.

Theorem 7. *Algorithm 2 runs in time*

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

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 one instantiation of each rule, there are at most m items on K at any time, and insertion into K is thus in $\mathcal{O}(\log(m))$. Furthermore, each rule is limited to N instantiations which means that the maximum number of items that will ever pass by K is Nm. This yields $\mathcal{O}(Nm\log(m))$ for the management of K.

Next, we have the rule-specific queues K_τ. For each instantiation of a rule τ of rank r_τ that is dequeued from K_τ, at most $r_\tau \leq r$ new instantiations are created; the creation of these instantiations takes $r_\tau^2 \leq r^2$ operations. Thus, there will be at most Nr elements in K_τ for any τ, which gives us a time of $\mathcal{O}(\log(Nr))$ per queue operation, a total time of $\mathcal{O}(N(r^2 + r\log(Nr)))$ for the management of K_τ, and thus a total of $\mathcal{O}(Nm(r^2 + r\log(Nr)))$ for the m queues K_τ altogether.

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

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

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 are represented in memory by the same object. 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 *outputted* 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 value of *outputted*. □

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

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

of Algorithm 1 [4]. By pruning instantiations of transition rules rather than states, we reduce running times roughly by a factor of $\mathcal{O}(Nn)$.

In practical applications, we expect to see Algorithm 2 used in two ways. The first is that discussed in the introduction, in which Algorithm 2 is invoked with a relatively small value of N, for example, $N = 200$ as suggested by Socher et al. [1]. Here, 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 improved pruning is even more valuable, as it saves redundant computation.

4. Conclusion

We have presented an improved version of the algorithm in [4] that solves the N-best problem for weighted tree automata over the tropical semiring. The main difference between the original version and that presented here, is the pruning of instantiations of transition rules, rather than of instantiations of states, and the lazy assembly of these instantiations. We have proved the new algorithm to be correct and asymptotically faster than the original, and we believe that this speed-up makes the algorithm superior for usage in typical language-processing applications.

[1] 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.

[2] 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.

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

[4] 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.

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

[6] 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.

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

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