Improved *N*-Best Extraction with an Evaluation on Language Data

Johanna Björklund
Umeå University
Department of Computing Science
johanna@cs.umu.se

Frank Drewes
Umeå University
Department of Computing Science
drewes@cs.umu.se

Anna Jonsson
Umeå University
Department of Computing Science
aj@cs.umu.se

We show that a previously proposed algorithm for the N*-best trees 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 with the original algorithm, the modifications increase the laziness of the evaluation strategy, which makes the new algorithm asymptotically more efficient than its predecessor. The algorithm is implemented in the software* BETTY*, and compared to the state-of-the-art algorithm for extracting the* N *best runs, implemented in the software toolkit* TIBURON*. The data sets used in the experiments are wtas resulting from real-world natural language processing tasks, as well as artificially created wtas with varying degrees of nondeterminism. We find that* BETTY *outperforms* TIBURON *on all tested data sets with respect to running time, while* TIBURON *seems to be the more memory-efficient choice.*

1. Introduction

Trees are standard in natural language processing (NLP) to represent linguistic analyses of sentences. Similarly, tree automata provide a compact representation for a set of such analyses. *Bottom–up tree automata* act as recognizing devices and process their input trees in a step-wise fashion, working upwards from the leaf nodes towards the root of the tree. For memory, they have a finite set of **states**, some of which are said to be **accepting**, and their internal logic is represented as a finite set of **transition rules**. A **run** of a tree automaton M on an input tree t is a mapping from the nodes of t to the states, which is

Submission received: 18 March 2021; revised version received: 31 August 2021; accepted: 5 December 2021.

https://doi.org/10.1162/COLI_a_00427

compatible with the transition rules. In general, M can have several distinct runs on t, and **accepts** t if one of these runs maps the root of t to an accepting state. By equipping the transition rules with weights, M can be made to associate t with a likelihood or score: The weight of a run of M on t is the product of the weights of the transition rules used in the run, and the weight of t is the sum of the weights of all runs on t. This type of automaton is called a weighted-tree automaton (wta) and is popular in, for example, dependency parsing and machine translation.

A central task is the extraction of the highest ranking trees with respect to a weighted tree automaton. For example, when the automaton represents a large set of intermediate solutions, it may be desirable to prune these down to a more manageable number before continuing the computation. This problem is known as the **best trees problem**. It is related to the **best runs problem** that asks for highest-ranking runs of the automaton on not necessarily distinct trees. The computational difficulty of the N-best trees problem depends on the algebraic domain from which weights are taken. Here we assume this domain to be the 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 (Benesty, Sondhi, and Huang 2008), since probabilistic devices can be modeled using negative log likelihoods. Moreover, the semiring 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 the non-extremal case, the problem is NP-complete even for strings [Lyngsø and Pedersen 2002].) The N-best trees problem for a given wta M over an extremal semiring can be solved indirectly by computing a list of N' best runs for M, for a sufficiently large number N', and outputting the corresponding trees while discarding previously outputted trees. A complicating factor with this approach, however, is that M can have exponentially many runs on a single tree, so N' may have to be very large to guarantee that the output contains N distinct trees.

Best trees extraction is useful in any application that includes some type of reranking of hypotheses. One example that makes use of best trees extraction is the work by Socher et al. (2013) on syntactical language analysis. The team of authors improve the Stanford parser by composing a probabilistic context-free grammar (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, e.g., the work by Zhao, Zhang, and Tu [2018]), and underlines the value of the N-best problem in language processing.

In previous work, Björklund, Drewes, and Zechner (2019) generalized an N-best algorithm by Mohri and Riley (2002) from strings to trees, resulting in the algorithm BEST TREES V.1. Intuitively, the algorithm performs a lazy implicit determinization and uses a priority queue to output N best trees in the right order. The running time of BEST TREES V.1 was shown to be in $\mathcal{O}(\max(Nmn \cdot (Nr + r \log r + N \log N), N^2 n^3, mr^2))$, 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. BEST TREES V.1 was evaluated empirically in Björklund, Drewes, and Jonsson (2018) against the *N* best runs algorithm by Huang and Chiang (2005), which represents the state of the art. Although Büchse et al. (2010) proved that the algorithm by Huang and Chiang works for cyclic input wtas and generalized it by extending it to structured weight domains, the core idea of the algorithm remains the same. The algorithm by Huang and Chiang (2005) is implemented in the widely referenced Tiburon toolkit (May and Knight 2006). From here on, we refer to this implementation simply as TIBURON, even though the best runs procedure is only one out of many that the toolkit has to offer. The conclusion of Björklund, Drewes, and Jonsson (2018) was that BEST TREES V.1 is faster if the input wtas exhibit a high degree of nondeterminism, whereas TIBURON is the better option when the input wtas are large but essentially deterministic.

We now improve BEST TREES V.1 by exploring the search space in a more structured way, resulting in the algorithm BEST TREES. In BEST TREES V.1, all assembled trees were kept in a single queue. In this work, 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 that are instantiations of τ, that is, trees with a run that applies τ at the root. This makes it possible to improve the strategy to prune the queue that was used by Björklund, Drewes, and Zechner (2019), and avoid 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 BEST TREES and derive an upper bound on its running time, namely, $\mathcal{O}(Nm(\log(m) + r^2 + r\log(Nr)))$.

In addition to solving the best trees problem, BEST TREES can also solve the best runs problem by removing the control structure that makes it discard duplicate trees (see Section 5). In this article, we make use of this possibility to compare this algorithm, implemented as BETTY, with TIBURON on the home turf of the latter, that is, with respect to the computation of best runs rather than best trees. For our experiments, we use both largely deterministic wtas from a machine translation project and from Grammatical Framework (Ranta 2011), and more nondeterministic wtas that were artificially created to expose the algorithms to challenging instances. Our results show that BETTY is generally more time efficient than TIBURON, despite the fact that the former is more general as it can also compute best trees (with almost the same efficiency as it computes best runs). Moreover, we perform a limited set of experiments measuring the memory usage of the applications, and can conclude that overall, the memory efficiency of TIBURON is slightly better than that of BEST TREES.

1.1 Related Work

The proposed algorithm adds to a line of research that spans two decades. It originates with an algorithm by Eppstein (1998) that finds the *N* best paths from one source node to the remaining nodes in a weighted directed graph. When applied to graphs representing weighted string automata, the list returned by Eppstein's algorithm may in case of nondeterminism contain several paths that carry the same string, that is, the list is not guaranteed to be free from duplicate strings. Four years later, Mohri and Riley (2002) presented an algorithm that computes the *N* best strings with respect to a weighted string automaton and thereby creates duplicate-free lists. To reduce the amount of redundant computation, consisting in the exploration of alternative runs on one and the same substring, they incorporate the *N* shortest paths algorithm by Dijkstra (1959). Moreover, Mohri and Riley work with on-the-fly determinization of the input automaton *M*, which avoids the problem that the determinized automaton can be exponentially larger than M, but has at most one run on each input string. Jiménez and Marzal (2000) lift the problem to the tree domain by finding the N best parse trees with respect to a context-free grammar in Chomsky normal form for a given string. Independently, Huang and Chiang (2005) published an algorithm that computes the best runs for weighted hypergraphs (which is equivalent to weighted tree automata and weighted regular tree grammars), and that is a generalization of the algorithm by Jiménez and Marzal in that it does not require the input to be in normal form. Huang and Chiang combine dynamic programming and lazy evaluation to keep the number of intermediate computations small, and derive a lower bound on the worst-case running time of their algorithm than Jiménez and Marzal do.

As previously mentioned, the algorithm by Huang and Chiang (2005) is implemented in the Tiburon toolkit by May and Knight (2006). Its initial usage was as part of a machine-translation pipeline, to extract the N best trees from a weighted tree automaton. In connection with this work, Knight and Graehl (2005) noted that there was no known efficient algorithm to solve this problem directly. As an alternative way forward, Knight and Graehl thus enumerate the best runs and discard duplicate trees, until sufficiently many unique trees have been found. Since TIBURON is highly optimized and the current state-of-the-art tool for best runs extraction, it is a natural choice of reference implementation for an empirical evaluation of our solution. The algorithm by Huang and Chiang (2005) was later generalized by Büchse et al. (2010) to allow a linear pre-order on the weights, as opposed to a total order. Büchse et al. (2010) also prove that the algorithm is correct on cyclic input hypergraphs, provided that the Viterbi algorithm for finding an optimal run (Jurafsky and Martin 2009) is replaced by Knuth's algorithm (Knuth 1974).[1]

Also, Finkel, Manning, and Ng (2006) remark on the lack of sub-exponential algorithms for the best trees problem. They propose an algorithm that approximates the solution when the automaton is expressed as a cascade of probabilistic tree transducers. The authors model the cascade as a Bayesian network and consider every step as a variable. This allows them to sample a set of alternative labels from each prior step, to propagate onward in the current step. The approximation algorithm runs in polynomial time in the size of the input device and the number of samples, but the convergence rate to the exact solution is not analyzed. To explain why their approach is preferable to finding N best runs, they extract the $N = 50$ best runs from the Stanford parser and observe that about half of the output trees are actually duplicates—enough to affect the outcome of the processing pipeline. Thus, they argue, extracting the highest ranking trees rather than the highest ranking runs is not only theoretically better, but is also of practical significance.

Finally, we note that the best trees problem also has applications outside of NLP. In fact, whenever we are considering a set of objects, each of which can be expressed uniquely by an expression in some particular algebra, and the set of expressions is the language of a wta, then the best trees algorithm can be used to produce the best objects. For instance, Björklund, Drewes, and Ericson (2016) propose a restricted class of hypergraphs that are uniquely described by expressions in a certain graph algebra, so the best trees algorithm makes it possible to find the optimal such graphs with respect to a wta. The reason why the representation needs to be unique is that otherwise we will have to check equivalence between objects as an added step. In the case of graphs,

[1] This replacement had already been done in TIBURON, without an explicit remark.

this would mean deciding graph isomorphism, which is not known to be tractable in general.

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.

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

A (commutative) semiring is a structure $(\mathbb{D}, \oplus, \otimes, 0, 1)$ such that both $(\mathbb{D}, \oplus, 0)$ and $(\mathbb{D}, \otimes, 1)$ are commutative monoids, the semiring multiplication $\otimes$ distributes over the semiring addition $\oplus$ from both left and right, and 0 is an annihilator for $\otimes$, that is, $0 \otimes d = 0 = d \otimes 0$ for all $d \in \mathbb{D}$. In this article, we will exclusively consider the *tropical semiring*. Its domain is $\mathbb{R}_+^\infty$, with min serving as semiring addition and ordinary plus as semiring multiplication.

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

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

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 simplified to 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 $\mathrm{rank}(f)$. The set T_Σ of ranked trees over Σ consists of all Σ-labeled trees t in which the rank of every node $v \in \mathrm{dom}(t)$ 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 $\Box \notin \Sigma$ 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 $v \in \mathrm{dom}(c)$ with $c(v) = \Box$. We define the **depth** of c to be $depth(c) = |v|$, i.e., the depth of c is the distance of $\Box$ from the root of c. The **substitution** of another tree t into c results in the tree $c[\![t]\!]$ given by $\mathrm{dom}(c[\![t]\!]) = \mathrm{dom}\, c \cup \{vu \mid u \in \mathrm{dom}(t)\}$ and, for all $w \in \mathrm{dom}\,(c[\![t]\!])$,

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

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. Weighted tree languages can be specified in a number of equivalent ways. Three of the standard ones, mirroring the ways in which regular string languages are traditionally specified, are weighted regular tree grammars, weighted tree automata, and weighted finite-state diagrams formalized as hypergraphs. The equivalence of the second and the third is shown explicitly in Jonsson (2021). All three have been used in the context of N-best problems: weighted regular tree grammars by May and Knight (2006), weighted tree automata by Björklund, Drewes, and Zechner (2019), and hypergraphs by Huang and Chiang (2005) and Büchse et al. (2010). In this article, we use weighted tree automata.

Definition 1
A weighted tree automaton (wta) over the tropical semiring 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 $f[q_1, \ldots, q_k] \overset{w}{\to} q$ 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 $\tau\colon f[q_1, \ldots, q_k] \overset{w}{\to} q$ will also be viewed as a symbol of rank k, turning R into a ranked alphabet. We let $tar(\tau)$ denote the target state q of τ, $src(\tau)$ denotes the sequence of source states $q_1 \ldots q_k$, and rank $(\tau) = \mathrm{rank}(f)$. In addition, we view every state $q \in Q$ as a symbol of rank 0.

Definition 2
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 that $q \in runs_M$ with $input_M(q) = q$, $wt_M(q) = 0$, and $tar(q) = q$.
2. For every transition rule $\tau\colon f[q_1, \ldots, q_k] \overset{w}{\to} q$ 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

$$input_M(\rho) = f[input_M(\rho_1), \ldots, input_M(\rho_k)]$$

$$wt_M(\rho) = w + \textstyle\sum_{i\in[k]} wt_M(\rho_i)\ \text{, and}$$

$$tar(\rho) = q$$

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

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

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

$$M(t) = \min\{M(\rho) \mid \rho \in runs_M \text{ and } input_M(\rho) = t\}$$

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 – which is the sum of all weights of t in the tropical semiring. 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 article, 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 as input a wta M and an integer $N \in \mathbb{N}$, 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 $\rho_1, \rho_2, \ldots$ of N pairwise distinct runs such that there do not exist $i \in [N]$ and $\rho \in runs \setminus \{\rho_1, \ldots, \rho_i\}$ with $M(\rho) < M(\rho_i)$.

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 $tar(\rho) = q_f$. 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.

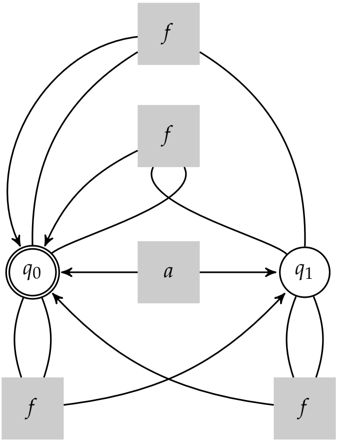
Similarly to the N-best runs problem, the *N-best* **trees problem** for the wta M consists in computing a sequence of pairwise distinct trees $t_1, t_2, \ldots$ in T_Σ of minimal weight. In other words, we seek a sequence of trees such that there do not exist $i \in [N]$ and $t \in T_\Sigma \setminus \{t_1, \ldots, t_i\}$ 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.

Example 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 automaton. Because 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 obtain the following recursive equations for the number $\#_i(t)$ of runs ρ on a tree t ending in state $tar(\rho) = q_i$:

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

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



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

Figure 1
A finite-state diagram representing 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.

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

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

We end this section by discussing the choice of our particular weight structure, the tropical semiring. In the literature on wta, Definitions 1 and 2 are generalized to wta over arbitrary commutative semirings, and their resulting weighted tree languages, simply by replacing $+$ and min by $\oplus$ and $\otimes$, respectively. The tropical semiring $(\mathbb{R}_+^\infty, \min, +, \infty, 0)$ used in Definitions 1 and 2 is frequently used in natural language processing. Equally popular is the *Viterbi semiring* $([0,1], \max, \cdot, 0, 1)$ that acts on the unit interval of probabilities, with maximum and standard multiplication as operations. In the setting discussed here, both semirings are equivalent. To see this, transform a wta M over the Viterbi semiring to a wta M' over the tropical semiring by simply mapping every weight p of a transition rule of M to $-\ln p$, that is, taking negative logarithms everywhere. Since $-\ln p + -\ln p' = -\ln(p \cdot p')$ and $-\ln p < -\ln p' \iff p > p'$, it holds that $M(t)$ (now calculated using the Viterbi semiring operations) is equal to $\exp(-M'(t))$ for all trees t. It follows that the trees $t_1, \ldots, t_N$ form an N-best list according to M (now looking for trees with *maximal* weights) if and only if they form an N-best list according to M' in the sense defined above.

3. The Improved Best Trees Algorithm

In this section, we explain how the algorithm in Björklund, Drewes, and Zechner (2019) can be made lazier, and hence more efficient, by exploring the search space with respect to transitions rather than states. From here on, let $M = (Q, \Sigma, R, q_f)$ be a wta with m transition rules, n states, and a maximum rank of r among the symbols in Σ.

3.1 BEST TREES V.1

We first summarize the approach of Björklund, Drewes, and Zechner (2019). 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,

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

Because $M_q(c)$ is independent of t, it is possible to compute in advance a **best context** $\mathbb{c}_q$ that minimizes it. The technique will be discussed in Section 3.2.

Definition 3
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: $opt(t) = \text{argmin}_{q \in Q} \mathbb{c}_q[\![t]\!]$. Throughout the article, we shall make use of the following weight functions derived from M, where $t \in T_\Sigma$:

$$\Delta_q(t) = \mathbb{w}_q + M^q(t)$$
$$\Delta(t) = M(\mathbb{c}_{opt_{(t)}}[\![t]\!])$$

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

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

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

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

```
 1: procedure BEST TREES V.1($M, N$)
 2:     compute best contexts
 3:     $T \leftarrow \emptyset$
 4:     $K \leftarrow \emptyset$
 5:
 6:     for $f \in \Sigma_0$ do
 7:         enq($K, f$)
 8:     end for
 9:
10:
11:
12:     $c \leftarrow 0$
13:     while $c < N$ do
14:         $t \leftarrow$ deq($K$)
15:
16:         $T \leftarrow T \cup \{t\}$
17:
18:
19:         if $M(t) = \Delta(t)$ then
20:             output($t$)
21:             $c \leftarrow c + 1$
22:         end if
23:         enq($K$, expand($T, t$))
24:
25:     end while
26: end procedure
```

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

```
 1: procedure BEST TREES($M, N$)
 2:     compute best contexts
 3:     output $t_{q_f}^{\text{best}}$
 4:     $T_q \leftarrow$ mkList($t_q^{\text{best}}$) for every $q \in Q$
 5:     for ($\tau\colon f[q_1, \dots, q_k] \overset{w}{\to} q) \in R$ do
 6:         $U \leftarrow \begin{cases} inc(1^{\text{rank}(\tau)}) & \text{if } \tau = \mathbb{p}_q(\lambda) \\ \{1^{\text{rank}(\tau)}\} & \text{otherwise} \end{cases}$
 7:         $K_\tau \leftarrow$ mkQueue($U$)
 8:     end for
 9:     $K' \leftarrow$ mkQueue($\{K_\tau \mid \tau \in R\}$)
10:     $c \leftarrow 0$
11:     while $c < N$ do
12:         $K_\tau \leftarrow$ deq($K'$)
13:         $u \leftarrow$ deq($K_\tau$)
14:         if $|T_{tar(\tau)}| < N \wedge \tau[u] \notin T_{tar(\tau)}$ then
15:             append $\tau[u]$ to $T_{tar(\tau)}$
16:         end if
17:         if $tar(\tau) = q_f \wedge \tau[u]$ is new then
18:             output($\tau[u]$)
19:             $c \leftarrow c + 1$
20:         end if
21:         enq($K_\tau, inc(u)$) [2]
22:         enq($K', K_\tau$)
23:     end while
24: end procedure
```

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 Björklund, Drewes, and Zechner (2019) in Algorithm 1. Given a wta M and $N \in \mathbb{N}$, it solves the N-best trees problem. After outputting $i \in [N]$ trees, the set of trees enqueued in line 23 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 Björklund, Drewes, and Zechner (2019).

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 t_q^{best} is computed for each state $q \in Q$. The desired property of t_q^{best} is that it is a 1-best tree of M^q, that is, it is a tree $t \in T_\Sigma$ that minimizes $M^q(t)$. After that, the second

[2] We assume that enq(K_τ, U) enqueues all $u \in U$ in K_τ except those already in it, thus skipping duplicates.

phase computes the actual best context $\mathbb{c}_q$ for every $q \in Q$, in other words, 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 (1977) 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 $\tau\colon f[q_1, \ldots, q_k] \overset{w}{\to} q$ is ∞ if any $\mathbb{p}_{q_i}$ ($i \in [k]$) is still undefined. Otherwise, it is $w + \sum_{i \in [k]} wt(\mathbb{p}_{q_i})$, that is, the weight of the run $\tau[\mathbb{p}_{q_1}, \ldots, \mathbb{p}_{q_k}]$. The algorithm repeatedly dequeues the highest priority element $\tau\colon f[q_1, \ldots, q_k] \overset{w}{\to} q$ 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 $t_q^{\text{best}} = f[t_{q_1}^{\text{best}}, \ldots, t_{q_k}^{\text{best}}]$. 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 t_q^{best} are discovered by the algorithm in the order of ascending weight. This observation will soon become important for the initialization phase of Algorithm 2.

As a side remark, we note that 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 t_q^{best} is itself one of the trees $\mathbb{p}_{q'}$ and $t_{q'}^{\text{best}}$, respectively. It follows that the entire set of these trees can be stored as a maximally shared directed acyclic graph with $|Q|$ nodes.

The second phase applies Dijkstra's shortest paths algorithm (Dijkstra 1959) to M, where M is viewed as a weighted edge-labeled graph. More precisely, consider the graph with node set Q such that, for every transition rule $(\tau\colon f[q_1, \ldots, q_k] \overset{w}{\to} q) \in R$ 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 $\mathbb{c}_{q'} = \mathbb{c}_q[\![f[t_{q_1}^{\text{best}}, \ldots, t_{q_{i-1}}^{\text{best}}, \square, t_{q_{i+1}}^{\text{best}}, \ldots, t_{q_k}^{\text{best}}]]\!]$ (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 the weight that needs to be added to $\mathbb{w}_q$ to obtain $\mathbb{w}_{q'}$ (under the assumption that, indeed, $\mathbb{c}_{q'} = \mathbb{c}_q[\![f[t_{q_1}^{\text{best}}, \ldots, t_{q_{i-1}}^{\text{best}}, \square, t_{q_{i+1}}^{\text{best}}, \ldots, t_{q_k}^{\text{best}}]]\!]$).

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 $\mathbb{c}_{q'} = \mathbb{c}(\pi)$, where $\mathbb{c}(\pi)$ is defined recursively as follows:

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

$$\mathbb{c}(\pi) = \mathbb{c}(\pi')[\![f[t_{q_1}^{\text{best}}, \ldots, t_{q_{i-1}}^{\text{best}}, \square, t_{q_{i+1}}^{\text{best}}, \ldots, t_{q_k}^{\text{best}}]]\!]$$

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 $t_{q'}^{\text{best}}$ for the required states q'.

3.3 Transition-Based Best Trees Computation

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, we build sequences T_q of $N' \leq N$ best trees for each state q. Most importantly, a separate priority queue K_τ is kept for each transition rule $\tau = f[q_1, \ldots, q_k] \xrightarrow{w} q$ in R. Every tree in this queue is of the form $f[(T_{q_1})_{i_1}, \ldots, (T_{q_k})_{i_k}]$ and is hence uniquely determined by the tuple $(i_1, \ldots, i_k)$, each i_j working as a pointer into the sequence T_{q_j}.[3] 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.

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 organize 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 organization is schematically illustrated in Figure 2.

The algorithm itself is outlined in Algorithm 2. The sequences T_q mentioned above are the previously dequeued best instantiations of rules τ with $tar(\tau) = q$. Thus, as mentioned before, T_q is a prefix of a solution of the N-best problem for the wta M^q.

For every $\tau\colon f[q_1, \ldots, q_k] \xrightarrow{w} q$ and every index tuple $u = (i_1, \ldots, i_k) \in \mathbb{N}^k$, the instantiated transition rule is denoted by $\tau[u]$. As previously mentioned, it is defined as the tree $f[(T_{q_1})_{i_1}, \ldots, (T_{q_k})_{i_k}]$, but since this is well defined only if $i_j \leq |T_{q_j}|$ for every $j \in [k]$, we let $\tau[u]$ be undefined otherwise.

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. Formally, for every tuple $u = (i_1, \ldots, i_k) \in \mathbb{N}^k$,

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

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 can be shown to be 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 more goal-oriented one. The priority of every element u of K_τ is primarily determined by a variant of Δ, denoted by $\Delta_\tau(u)$. Recall that, for $q \in Q$, $\Delta_q(t)$ is the

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

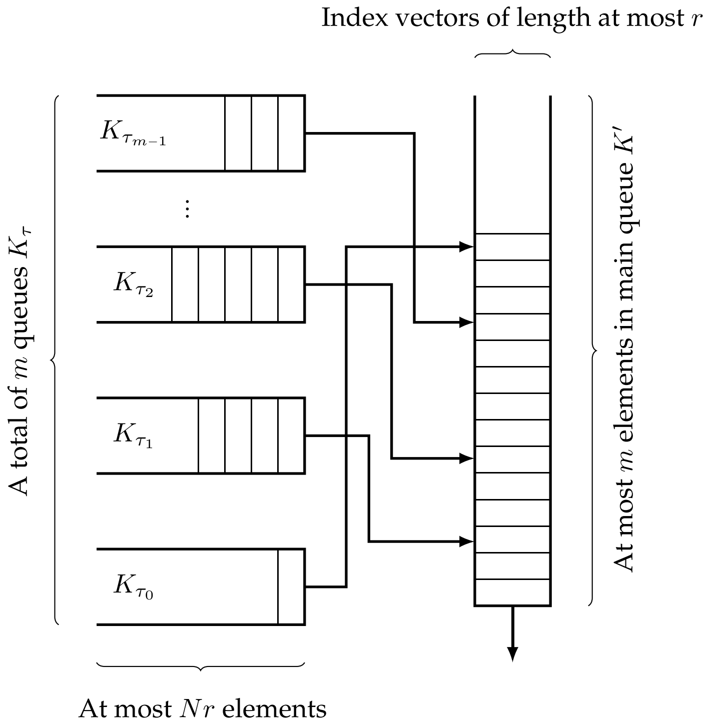

Index vectors of length at most r
$K_{\tau_{m-1}}$
⋮
K_{τ_2}
K_{τ_1}
K_{τ_0}
A total of m queues K_τ
At most m elements in main queue K'
At most Nr elements

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).

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 specializes this further by assuming that the specific transition rule applied at the root of the subtree t is τ. Moreover, because 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

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

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 $\mathtt{w}_q$, $\Delta_\tau(u)$ 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 $v \in \operatorname{dom} \rho$. 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, namely, $(i,j) < (i',j')$ 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 initialized to contain only the best tree t_q^{best} for q. The queue K_τ is initialized to contain only 1^k if $\tau \neq \mathbb{p}_q(\lambda)$, since the latter means that τ was not used to build t_q^{best}, 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 initialized to contain all successors of 1^k because, by the way in which t_q^{best} was constructed and the definition of $\tau[u]$, we have $(T_q)_1 = t_q^{\text{best}} = \tau[1^k]$ and are now looking for the next best instantiation of τ.

After initializing 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 $\Delta_\tau(u) \in \mathbb{N}$, that is, the tree $\tau[u]$ is defined (see the next section). 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," that is, 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.

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 $|T_{tar(\tau)}| < N$ from the condition on line 14. In the following lemma, we say that $T_q = t_1 \cdots t_m$ is **appropriate** if $t_1, \ldots, t_m$ is a solution of the m-best trees problem for M^q.

Lemma 1
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 appropriate.

(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. Because it holds before the first execution of the loop (as each T_q consists of a single best tree with respect to M^q), 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 $\mathbb{c}_{q'}(v_0) = \square$, that is, v_0 is the node such that $s/v_0 = t$. By the definition of $\Delta_{q'}$, there is a run ρ with $\mathit{input}(\rho) = s$, $\mathit{tar}(\rho/v_0) = q'$, and $\mathit{wt}(\rho) = \Delta_{q'}(t)$. For every $v \in \mathrm{dom}(s)$, let $q_v = \mathit{tar}(\rho/v)$ be the state the run is in after having processed s/v. There is at least one node $v \in \mathrm{dom}(s)$ such that $s/v \notin T_{q_v}$ (namely, $v = v_0$). Now, choose $v \in \mathrm{dom}(s)$ 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 $p = \mathit{opt}(s/v)$ it follows that

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

Because $s/v \notin T_{q_v}$, 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 appropriate. 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). Because $\tau[u]$ 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 the appropriateness of T_q and the second holds because $\mathit{tar}(\tau) = q$. Furthermore, by (2) there is no tree $t' \in T_\Sigma \setminus T_q$ such that $M^q(t') < M^q(\tau[u])$. Thus, T_q is still appropriate when $\tau[u]$ has been appended to it.

Lemma 2
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 $\tau\colon f[q_1, \dots, q_k] \xrightarrow{w} q$. 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 1, 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

$$\mathbb{c}_{q'}[\![g[t^{\mathrm{best}}_{p_1}, \dots, t^{\mathrm{best}}_{p_{i-1}}, \square, t^{\mathrm{best}}_{p_{i+1}}, \dots, t^{\mathrm{best}}_{p_\ell}]]\!]$$

for a transition rule $\tau'\colon g[p_1, \dots, p_\ell] \xrightarrow{w'} q'$ 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

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

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

1. there is no u'' in any of the queues $K_{\tau''}$ with $\Delta_{\tau''}(u'') < \Delta_\tau(u)$ (Lemma 1(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 $K_{\tau''}$ from which a tuple is 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. Because, 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 1
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 color, red or black, where black is the default color. (The color attribute of a subtree may differ from that of the tree itself.) Suppose that, in an iteration of the main loop, we have $\tau\colon f[q_1, \dots, q_k] \xrightarrow{w} q$ and $u = (i_1, \dots, i_k)$, and thus $\tau[u] = f(t_1, \dots, t_k)$ 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 color $\tau[u]$ red while its subtrees $t_1, \dots, t_k$ keep their colors as given by their positions in $T_{q_1}, \dots, T_{q_k}$.

Now, assume that some T_q contains a tree $t = f[t_1, \dots, t_k]$ 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 $u = (i_1, \dots, i_k)$ 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 $u^i = (i_1, \dots, i_{j-1}, i, i_{j+1}, \dots, i_k)$ for $i = 1, \dots, N$. (This is an immediate consequence of Lemma 2 because, by Lemma 1(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.

Because 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 trees in T_{q_f} are black as well. As subtrees inherit their color 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 2. □

3.5 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 along the lines illustrated in Figure 2, with priority queues based on heaps (Cormen et al. 2009). This enables us to implement the following details efficiently:

- Consider a transition $\tau\colon f[q_1, \dots, q_k] \xrightarrow{w} q$. At a given stage of the algorithm, some of the elements $u = (u_1, \dots, u_k)$ in K_τ may contain elements u_i such that $|T_{q_i}| < u_i$, that is, $\tau[u]$ 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 2
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.

Because 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 (Björklund, Drewes, and Zechner 2019). By handling instantiations of transition rules rather than states, we reduce the running times of the algorithm roughly by a factor of $O(Nn)$.

We have not performed a detailed space complexity analysis, but because we know that the logarithmic factors are due to heap operations, we can conclude that the memory space consumption of the algorithm is in $O(Nm)$.

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. (2013). 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 recognized 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. The Algorithm BEST RUNS

We now recall the algorithm by Huang and Chiang (2005), which we henceforth will refer to as BEST RUNS. To facilitate comparison, we express BEST RUNS in terms of wta.

The type of wta used as input to BEST RUNS differs slightly from the one used in Definition 1 in that the admissible weight structures are not restricted to the tropical semiring. Instead, each transition rule $\tau\colon f[q_1, \ldots, q_k] \to q$ is equipped with a weight function $\mathrm{wt}_\tau\colon \mathbb{R}_+^k \to \mathbb{R}_+$. The definition of the weight of a run $\rho = \tau[\rho_1, \ldots, \rho_k]$ is then changed to $\mathrm{wt}(\rho) = \mathrm{wt}_\tau(\mathrm{wt}(\rho_1), \ldots, \mathrm{wt}(\rho_k))$.

For the algorithm to work, these weight functions wt_τ are required to be monotonic: $\mathrm{wt}_\tau(w_1, \ldots, w_k) \geq \mathrm{wt}_\tau(w'_1, \ldots, w'_k)$ whenever $w_i \geq w'_i$ for all $i \in [k]$. Definition 1, which BEST TREES is based on, corresponds to the special case where each of these weight functions is of the form $\mathrm{wt}_\tau(w_1, \ldots, w_k) = w + \sum_{i \in [k]} w_i$ for a constant w. In other words, the weight functions BEST TREES can work with are a restriction of those BEST RUNS works on. This will be discussed in Section 5.

The input to BEST RUNS is a pair (M, N), where M is a wta with m transition rules and n states, and $N \in \mathbb{N}$. The algorithm is outlined in Algorithm 3. Line 2 is a preprocessing step that can be performed in $O(m)$ time using the Viterbi algorithm, given that the rank of the alphabet used is considered a constant. A list *input*$[q]$ is used to store, for each state $q \in Q$, the at most N discovered best runs arriving at q. The search space of candidate runs is represented by an array of heaps, here denoted *cands*. For each state q, *cands*$[q]$ holds a heap storing the (at most N) best, so far unexploited, runs arriving at q. That is, if we have already picked the N' best runs arriving at a node, the heap lets us pick the next best unpicked candidate efficiently when so requested by the recursive call.

To expand the search space, the N'-th best run ρ is used as follows: if $\tau = ([q_1, \ldots, q_k] \to q)$, we obtain a new candidate by replacing the i-th direct subtree of ρ with the next (and thereby minimally worse) run in *input* arriving at q_i. Note that this is equivalent to the increment method used in BEST TREES.

As shown by Huang and Chiang (2005), the worst case running time of BEST RUNS is $O(m \log |V| + s_{\max} N \log N)$ where $s_{\max}$ is the size of the largest run among the N results.[4] Using similar reasoning for the memory complexity as for BEST TREES, BEST RUNS achieves a $O(m + s_{\max} N)$ memory complexity bound.

5. Comparison

A major difference between BEST RUNS and BEST TREES is that the former solves the N-best runs problem whereas the latter solves the N-best trees problem: On lines 14 and 17 of Algorithm 2, duplicate trees are discarded. If these conditions are removed, BEST TREES solves the best runs problem. (Provided, of course, that the objects outputted are changed to being runs rather than trees.)

For the sake of comparison, we adopt the view of Huang and Chiang (2005) that the ranked alphabet Σ can be considered fixed. This yields that the running time of BEST TREES is $O(Nm(\log m + \log N))$. Since $m \in O(n^{r+1})$, where r is the (now fixed) maximal rank of symbols in Σ, it follows that $\log m \in O(\log n)$, so the second expression

[4] In fact, the running time obtained by Huang and Chiang (2005) is $O(m + s_{\max} N \log N)$, but this assumes (the graph representation of) M to be acyclic. When M is cyclic, line 2 must be implemented by using Knuth's algorithm, resulting in an additional factor $\log n$ in the first term.

simplifies to $Nm(\log n + \log N) \leq m \log n \cdot N \log N$. Moreover, recall that the worst case running time of BEST RUNS is $O(m \log n + s_{\max} N \log N)$. If N is large enough to make the second term the dominating one, the difference between both running times is thus a factor of $(m \log n)/s_{\max}$ (assuming for the sake of the comparison that the given bounds are reasonably tight). A further comparison between the running times does not appear to be all that meaningful because $s_{\max}$ depends on both N and the structure of the input wta, and the algorithms are specialized for different problems.

Algorithm 3 Compute $N \in \mathbb{N}^{\infty}$ runs arriving at $q_{\mathrm{f}} \in Q = [n]$ of minimal weight according to a wta $M = (Q, \Sigma, R, q_{\mathrm{f}})$ (Huang and Chiang 2005).

```
procedure BEST RUNS($M, N$)
    Compute 1-best run $\rho_q^{\mathrm{best}}$ arriving at $q$, for every $q \in Q$
    $input[q] \leftarrow$ LIST-EMPTY() for all $q \in Q$
    LIST-ADD($input[q], \rho_q^{\mathrm{best}}$) for all $q \in V$
    BEST RUNS$_0$($q_{\mathrm{f}}, N$)
    for $\rho \in input[q_{\mathrm{f}}]$ do
        output($\rho$)
    end for
end procedure

procedure BEST RUNS$_0$($q, N$)
    if $|input[q]| \geq N$ then
        return
    end if
    if $cands[q]$ is undefined then
        $tmp \leftarrow$ SORT-BY-WEIGHT($\{\tau[\rho_{q_1}^{\mathrm{best}}, \ldots, \rho_{q_k}^{\mathrm{best}}] \mid \tau = ([q_1, \ldots, q_k] \to q) \in R\}$)
        $cands[q] \leftarrow$ HEAP-IFY($tmp[1] \ldots tmp[N]$)
        LIST-ADD($input[q]$, HEAP-EXTRACT-MIN($cands[q]$))
    end if
    while LIST-SIZE($input[q]$) $< N$ and HEAP-SIZE($cands[q]$) $> 0$ do
        $s \leftarrow$ LIST-SIZE($input[q]$)
        $\rho \leftarrow$ LIST-GET($input[q], s$)                  ▷ the $s$-best run arriving at $q$
        NEXT RUNS($cands[q], \rho$)                         ▷ build new candidates from $\rho$
        LIST-ADD($input[q]$, HEAP-EXTRACT-MIN($cands[q]$))
    end while
end procedure

procedure NEXT RUNS($qcands, \rho = \tau[\rho_1, \ldots, \rho_k]$ where $\tau = ([q_1, \ldots, q_k] \to q)$)
    for $i \leftarrow 1, \ldots, k$ do
        $list \leftarrow input[q_i]$
        $N' \leftarrow$ index of $\rho$ in $input[q]$
        BEST RUNS$_0$($q_i, N'$)                         ▷ make sure $list$ has $N' \leq N$ elements
        if $N' \leq$ LIST-SIZE($list$) then
            $\rho_i' \leftarrow$ LIST-GET($list, N'$)
            $\rho' \leftarrow \tau[\rho_1, \ldots, \rho_{i-1}, \rho_i', \rho_{i+1}, \ldots, \rho_k]$
            if $\rho' \notin qcands$ then
                HEAP-INSERT($qcands, \rho'$)
            end if
        end if
    end for
end procedure
```

A conceptual comparison of the two algorithms may be more insightful. The algorithms differ mainly in two ways. The first difference is that, while BEST RUNS enumerates candidates of best runs using one priority queue per node of G, BEST TREES uses a more fine-grained approach, maintaining one priority queue per hyperedge. Since the upper bound on the length of queues is $O(N)$ in both cases, in total BEST TREES may need to handle $O(Nm)$ candidates whereas BEST RUNS needs only $O(Nn)$. This ostensible disadvantage of BEST TREES is not a real one as it occurs only when solving the N-best trees problem. The difference vanishes if the algorithm is used to compute best runs (by changing lines 14 and 17). To see this, consider a given state $q \in Q$. Each time a queue element $(i_1, \ldots, i_k)$ is dequeued from a queue K_τ with $\tau\colon f[q_1, \ldots, q_k] \to q$, the corresponding run is appended to the list T_q on line 15, and k queue elements are inserted on line 21. As this can only happen at most N times per state q, in total only $O(Nn)$ queue elements are ever created in the worst case. In other words, if BEST TREES is set to solve the N-best runs problem, the splitting of queues does not result in a disadvantage compared to BEST RUNS.

The second conceptual difference between the algorithms is that BEST TREES adopts an optimization technique known from the N-best strings algorithm by Mohri and Riley (2002). It precomputes and uses (the weight and depth of) a best context c_q for every state q in order to explore the search space in a more goal-oriented fashion. The possibility of using this optimization depends on the use of the tropical semiring. As mentioned in the Introduction, Büchse et al. (2010) 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 the case for the tropical semiring, but not for structured weight domains in general. Let ℓ be the maximum of the distances of states in Q to q_{f}, that is,

$$\ell = \max_{q \in Q} \mathrm{dist}(q, q_{\mathrm{f}})$$

where $dist(q, q_{\mathrm{f}})$ denotes the length of the shortest sequence of transition rules $\tau_0 \cdots \tau_n$ such that $q \in src(\tau_0)$, $tar(\tau_{i-1}) \in src(\tau_i)$ for all $i \in [n]$, and $q_{\mathrm{f}} = tar(\tau_n)$. The argument used to show Claim 1 in the proof of Lemma 2 yields that, at every point in time at most ℓ further loop executions are made before the next best run is outputted. To see this, consider an execution of the main loop, in which a run $\rho = \tau[\cdots]$ arriving at a state q is constructed. The priority of the corresponding queue element on line 13 is given by $(\mathrm{wt}(\rho) + w, d)$, where w and d are the weight and depth of c_q. If $q \neq q_{\mathrm{f}}$, then $c_q \neq \square$, and thus it is of the form $c_{q'}[\![\tau'[\rho_1, \ldots, \rho_{i-1}, \square, \rho_{i+1}, \ldots, \rho_k]]\!]$, where the ρ_j are the 1-best runs arriving at their respective nodes (see Figure 3).

It follows that line 21 inserts the element into K_τ that represents the run $\rho' = \tau'[\rho_1, \ldots, \rho_{i-1}, \rho, \rho_{i+1}, \ldots, \rho_k]$. Because each of the runs ρ_i, arriving at some state q_i, is already in the respective list T_{q_i}, the run ρ' immediately becomes a current candidate arriving at $q = tar(\tau)$. The corresponding pair $(w_q, d_q) = (\mathrm{wt}(c_q), depth(c_q))$ satisfies $w_q + \mathrm{wt}(\rho') = w + \mathrm{wt}(\rho)$ and $d_q = d - 1$. Hence, ρ' (or another run with the same priority) will be picked in the next loop execution, meaning that after at most ℓ steps the node arrived at by the constructed run will be q_{f}, that is, the run will be outputted. This also shows that queues the elements of which are not needed for generating output trees will never become filled beyond their initial element.

We end the comparison by looking at Table 2, which summarizes the discussion above and provides comparison data for the other N-best algorithms discussed in this

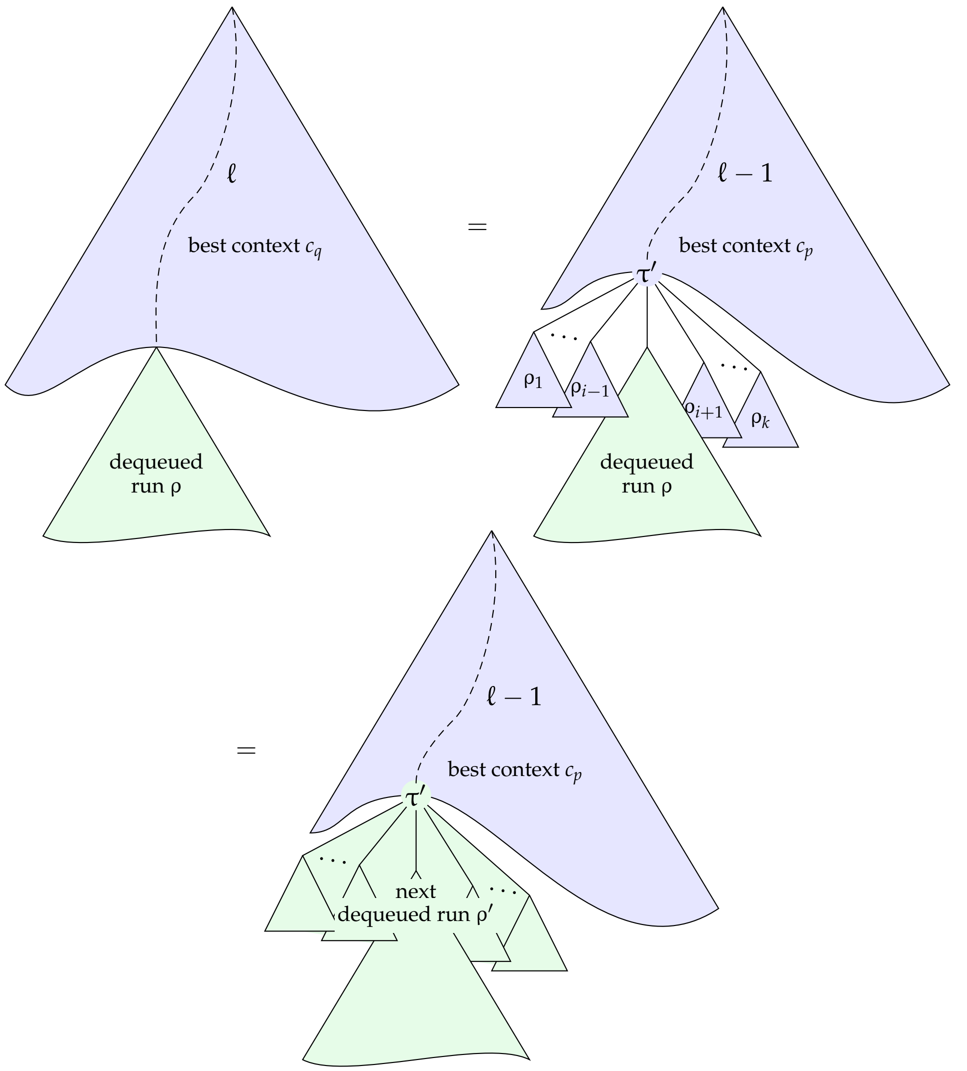


Figure 3
When a run ρ reaching state q is dequeued with $\ell > 0$ (top left), then its best context c_q is of the form $c_p[\![\tau'[\rho_1, \ldots, \rho_{i-1}, \Box, \rho_{i+1}, \ldots, \rho_k]]\!]$, where c_p is the best context for $p = tar(\tau')$ and the ρ_i are 1-best trees reaching the remaining states of τ′ (top right). Hence, $\rho' = \tau'[\rho_1, \ldots, \rho_{i-1}, \rho, \rho_{i+1}, \ldots, \rho_k]$ (or a run which likewise is less than ℓ steps away from an output run) will be the next run dequeued (bottom).

article. Keep in mind that N must be interpreted differently depending on the problem at hand. For example, the output of BEST RUNS is not equivalent to the output of BEST TREES (unless the input is deterministic), which is why we cannot directly compare the time complexities. This output inequivalence should also be considered in Section 7 where, for simplicity, we plot data for BEST RUNS and BEST TREES side by side.

Table 2
Summary of characteristics of N-best algorithms. For the time complexities, the alphabet is taken to be constant and the input is represented as a wta. The two right-most columns indicate whether the algorithms compute best contexts as a preprocessing step, and what optimization methods are used to find new candidate objects.

Algorithm	Objects	Time complexity	Best contexts	Search-space expansion
Eppstein (1998)	Paths	$O(n \log n + Nn + m)$	No	Adding sidetracks to implicit heap representations of paths
Mohri and Riley (2002)	Strings	No formal analysis provided	Yes	On-the-fly determinization
Huang and Chiang (2005)	Runs	$O(m \log n + s_{\max} N \log N)$[5]	No	Increment
BEST TREES V.1	Trees	$O(N^2 n(n^2 + m \log N))$	Yes	Eppstein's algorithm
BEST TREES	Trees Runs	$O(Nm(\log m + \log N))$ $O(N(\log m + \log N))$[6]	Yes Yes	Increment Increment

6. Implementation Details

In the upcoming section, we experimentally compare BEST RUNS and BEST TREES. In preparation of that, we want to make a few comments on the implementation of BEST TREES. As previously mentioned, BEST RUNS is implemented in the Java toolkit TIBURON by May and Knight, and this is the implementation we use in our experiments. Therefore, we simply refer to the TIBURON GitHub page[7] for implementation details.

We have extended our code repository BETTY,[8] which originally provided an implementation of BEST TREES V.1, to additionally implement the improved BEST TREES as its standard choice of algorithm. A flag `-runs` can be passed on as an argument to compute best runs instead of best trees. Below follow a number of central facts about the BETTY implementation.

First, recall Algorithm 2, and in particular that the best tree t_q^{best} is inserted into T_q for all $q \in Q$ prior to the start of the main loop rather than letting T_q be empty and initializing K_τ to $1^{\operatorname{rank} \tau}$ for all $q \in Q$ and $\tau \in R$. The reason is that the latter would not guarantee that at most ℓ iterations are made until a run is outputted because some ρ_i may still not be in T_q. If this happens, transition rules adding the weight 0 may repeatedly be picked because some other τ' is not yet enabled. However, with a trick the initialization can nevertheless be simplified as indicated. The idea is to delay the execution of line 22 for every queue K_τ until τ has actually appeared in an output tree. Thus, until this has happened, K_τ is disabled from contributing another tree to $T_{tar(q)}$. This variant turned out to have efficiency advantages in practice and is therefore the variant implemented in BETTY. Another advantage is that it allows BETTY to handle a *set*

[5] Allows for cyclic input wta; $s_{\max}$ is the size of the largest output.
[6] Note that a factor m is removed, compared with when the same algorithm is used for finding the best trees. This is because all runs originating at the same rule queue are distinct (and naturally the same also holds for different rule queues).
[7] `https://github.com/isi-nlp/tiburon`.
[8] `https://github.com/tm11ajn/betty`.

of final states rather than a single one. This is not possible with the original initialization of Algorithm 2, because line 3 would have to be generalized to outputting the best trees of all final states, which cannot be done while maintaining the correctness of the algorithm, since the second best tree for a state q may have a lesser weight than the best tree for another state q'.

In lines 14 and 17 of Algorithm 2, trees are checked for equivalence. To perform the comparisons efficiently, we make use of hash tables. We use immutable trees, which need to be hashed only once when they are created; we then save the hash code together with the tree to make the former accessible in constant time for each tree. To compare trees for inequality, their hash codes are compared. If equal (which seldom happens if the trees are not equal), the comparison is continued recursively on the direct subtrees. This is theoretically less efficient than the method of representing trees uniquely in memory (as described in the last paragraph of the proof of Theorem 2), but practically sufficient and much easier to implement.

When creating new tuples for a transition rule τ as given by line 21, we add the tuples that can be instantiated directly to the corresponding queue K_τ. The tuples that cannot be instantiated must, however, be stored until they can. We want to be able to efficiently access the tuples that are affected when adding a tree t to $T_{q'}$ for some $q' \in Q$ (line 15). Therefore, we connect each tuple to the memory locations that will contain the data needed by the tuple. In more detail: Let $\tau = (f[q_1 \cdots q_k] \to q)$ be any transition rule in R and let $(i_1, \ldots, i_k)$ be a tuple originating from τ. For every $i_j \in \{i_1, \ldots, i_k\}$, the tuple is saved in a list of tuples affected by $T_{q_j}(i_j)$ and marked with a counter that shows how many trees remain until it can be instantiated. Thus, when t is added to $T_{q'}(i_j)$ for $i_j \in \{i_1, \ldots, i_k\}$ and $q' = q_j$, we can immediately access all tuples that can possibly be instantiated. If a tuple cannot be instantiated, its counter is decreased appropriately.

7. Experiments

Let us now describe our experiments. For each problem instance (i.e., combination of input file and value of N), we perform a number of test runs, measure the elapsed time, and compute the average over the test runs. To avoid noise in our data caused by, for example, garbage collection, we only measure the time consumed by the thread the actual application runs in. Also, we disregard the time it takes to read the input files.

The number of test runs that are performed per problem instance is decided by the relationship between the mean μ and the standard deviation σ of the recorded times—these values are computed every fifth test run, and to finish the testing of the current problem instance, we require that $\sigma < 0.01\mu$. However, five test runs per problem instance has turned out to be sufficient for fulfilling the requirement in practically all instances seen.

The memory usage is measured in terms of the maximum resident set size of the application process by using the Linux `time` command. Moreover, the strategy described above for computing the average over several test runs is used here as well.

All test scripts have been written in Python, in contrast to the tested implementations, which use Java. We run the experiments on a computer with a 3.60 Hz Intel Core i7-4790 processor.

7.1 Corpora

The corpora that were used in our experiments (see the list below) contain both real-world and synthetic data. The first corpus is derived from an actual machine-translation system and is thus representative for real-world usage. The second corpus consists of manually engineered grammars for a set of natural languages, used in a range of research and industry applications. The last two corpora are artificially created for the purpose of investigating the effect of increasing degrees of nondeterminism. Let us now present each in closer detail.

`MT-data` This data set consists of tree automata resulting from an English-to-German machine-translation task, described in the doctoral thesis of Quernheim (2017). The data consists of 927 files, each file containing a wta corresponding to one sentence; the files are indexed from 0 to 926. The smallest file has 24 lines and the largest one has 338,937 lines; all lines but the first hold a transition rule. Moreover, these wtas have a large number of states and are essentially deterministic in the sense that each state corresponds to a particular part-of-sentence structure and input node.

`GF-data` Unweighted context-free grammars from the Grammatical Framework (GF) by Ranta (2011) provided the basis for this corpus. GF is, among other things, a programming language and processing platform for multilingual grammar applications. It provides combinatory categorial grammars (Steedman 1987) for more than 60 natural languages, out of which we export a subset to context-free grammars in Backus–Naur form with the help of the built-in export tool, and assign every grammar rule the weight 1. The number of production rules varies between languages: The Latin grammar has, for example, 1.6 million productions, whereas the Italian grammar has 5.8 million. (These differences are due to the level of coverage chosen by the grammar designers, and do not necessarily reflect inherent complexities of the languages.)

`PolyNonDet` This is a family of automata of increasing size indexed by i. The states of member i are $q_0, \ldots, q_i$, where q_i is the final state and the rules are:

- $a \xrightarrow{0} q_j$ for $j = 0, \ldots, i$
- $f[q_j, q_j] \xrightarrow{1} q_j$ for $j = 0, \ldots, i$
- $f[q_j, q_{j-1}] \xrightarrow{1} q_{j-1}$ for $j = 1, \ldots, i$

There are then $\Theta(n^i)$ runs for a tree of size n (and the number of rules grows linearly). Therefore, we call these *polynomially nondeterministic*.

`ExpNonDet` Finally, we use another family of automata of increasing size, also indexed by i and with states $q_0, \ldots, q_i$ and a final state q_f. The rules for member i of the family for $j, k = 0, \ldots, i$ and $j \neq k$ are:

- $a \xrightarrow{0} q_j$
- $f[q_j, q_k] \xrightarrow{1} q_j$
- $f[q_k, q_j] \xrightarrow{1} q_j$

- $q_j \xrightarrow{0} q_f$

Thus, the number of rules grows quadratically in i and the number of runs for a tree of size n is $\Omega((i+1)^n)$; we say that these are *exponentially nondeterministic*.

The variables in all result-displaying plots in this article are either N (the number of trees or runs to output), m (the number of transition rules, i.e., lines in the input file), or both. For the artificially created corpora `PolyNonDet` and `ExpNonDet`, we exchange m for the variable i with which they are indexed. Note that for the former, m and i are interchangeable, but for the latter, m grows quadratically with increasing i.

7.2 Comparison with BEST TREES V.1

First, we verify that the algorithm BEST TREES is at least as efficient as its predecessor BEST TREES V.1. Therefore, we run experiments on the `ExpNonDet` data set for both of the algorithms—both solving the best trees problem. The `ExpNonDet` data set was chosen because it has the largest degree of nondeterminism achievable, which should challenge both of the algorithms maximally. The results are presented in figures 4 and 5 for varying N and i, respectively. Note that both of these plots have logarithmic y axes. We see that BEST TREES outperforms BEST TREES V.1 considerably for both increasing N and increasing i.

More interestingly, in Figure 4, BEST TREES displays a step-like behavior at $N \approx 100$, 200, and 600. The nature of the `ExpNonDet` corpus is the explanation of this: When we have found all of the distinct trees of size s (all of the same weight), then the algorithm has to discard all of the duplicates of size s that are still in the queue, before arriving at a tree of size $s+1$ with higher weight.

7.3 Comparison with TIBURON

Before turning to the comparison between BETTY and TIBURON, recall from Section 5 that when using BETTY to find the best trees, the input N cannot be equated with the N

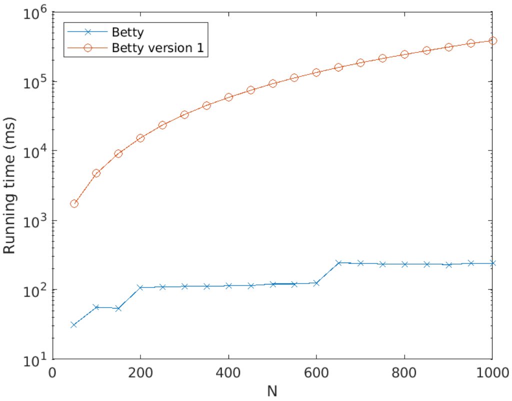


Figure 4
Comparison on the `ExpNonDet` file with $i = 7$.

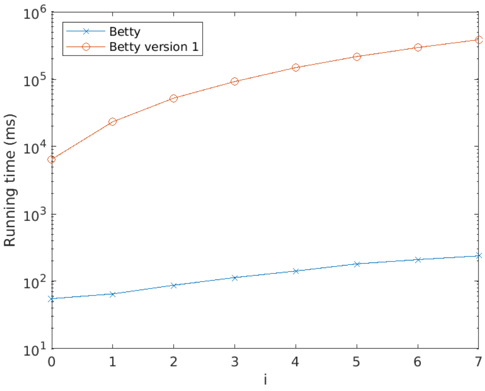


Figure 5
Comparison for $N = 1000$ on the `ExpNonDet` corpus.

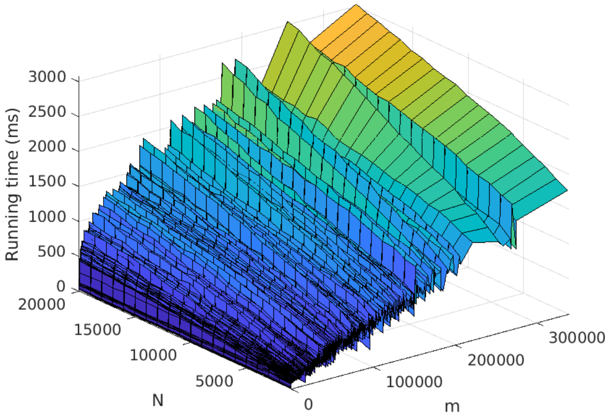


Figure 6
TIBURON solves the best runs problem for the `MT-data` corpus.

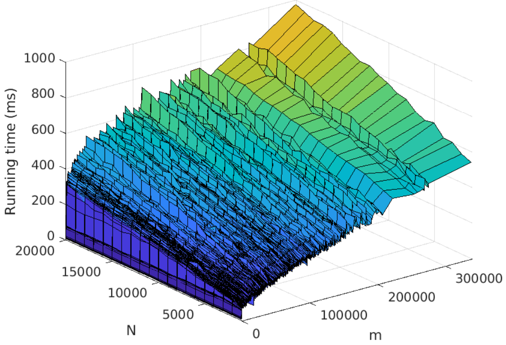


Figure 7
BETTY solves the best runs problem for the `MT-data` corpus.

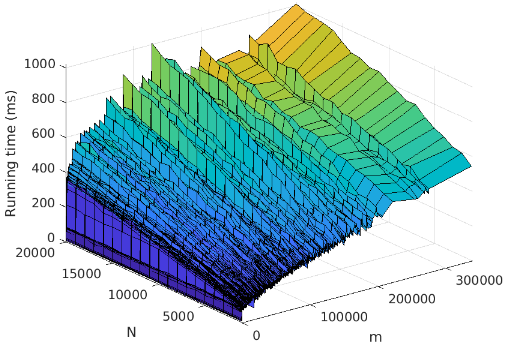


Figure 8
BETTY solves the best trees problem for the `MT-data` corpus.

that is input to TIBURON and BETTY to find the best runs. When the practical application actually demands best trees, a best runs implementation like TIBURON will potentially have to compute many more runs than trees needed, thus by necessity resulting in larger N. We have, however, for simplicity chosen to show the best trees data in the same plot as the data for best runs. It should also be noted that this comparison between BETTY and TIBURON is essentially a "stress test" for BETTY, which was designed to solve best trees but is here reduced to compute best runs.

Next, we compare BETTY with TIBURON for all data sets, starting with the `MT-data` corpus. Figure 6 shows the running times for finding the N best runs using TIBURON for every file of the `MT-data` corpus and $N = 1{,}000, 2{,}000, \ldots, 20{,}000$; recall that m is the number of transition rules in the input file. The corresponding results for BETTY can be seen in Figure 7. When we instead let BETTY solve the best trees problem for the same input, we achieve the result in Figure 8. Figures 9 and 10 show the results for all three implementations in the same plots for a fixed file and value of N, respectively. It seems that BETTY is faster than TIBURON on both tasks, but to be certain, we perform a

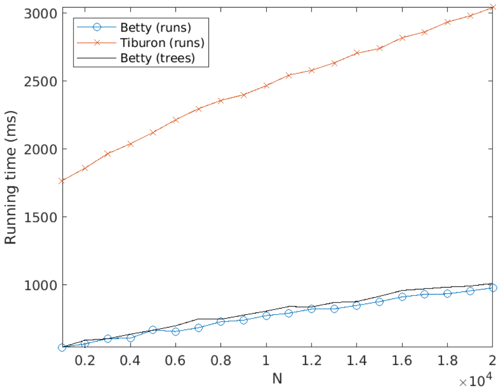


Figure 9
Comparison of all three implementations on the largest `MT-data` file.

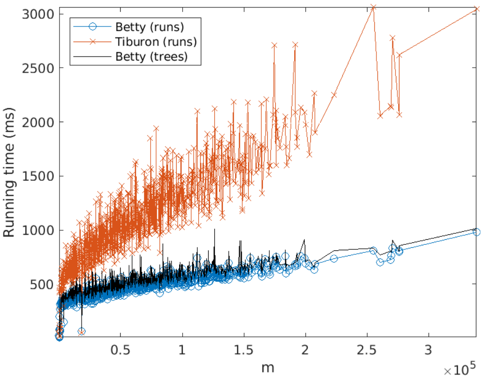


Figure 10
Comparison of all three implementations for $N = 25{,}000$ on all `MT-data` files.

statistical test. Because we have paired results from two populations, we run a Wilcoxon signed rank test on the results. We prepare both result sets for statistical testing by splitting them into 9 vectors of size 103 for each value of N; the splitting is done to allow us to find patterns in which implementation performs better given the parameters. Let v_{BETTY} and v_{TIBURON} be such result vectors from the test runs of BETTY and TIBURON, respectively. Our null hypothesis H_0 is that $v_{\text{DIFF}} := v_{\text{BETTY}} - v_{\text{TIBURON}}$ comes from a distribution with median 0, and our alternative hypothesis H_1 is that v_{DIFF} comes from a distribution with median less than 0. For our statistical tests, we use a significance level of 0.01. The result has the form of probabilities of observing values as or more extreme than the data under the null hypothesis H_0, and it shows that each probability is smaller than 10^{-18}. (Because the numbers are insignificantly small, we exclude a table of detailed results.) We can therefore conclude that H_0 is rejected for all partitions of our data. This implies that BETTY is statistically significantly faster than TIBURON on the `MT-data` corpus for both tasks. For the rest of the experiments, we omit similar statistical tests.

Next, we apply the three algorithms to the `GF-data` corpus. We run tests for 42 different languages, and present a selection of typical results along with a single atypical one. The typical results are shown in figures 11, 12, and 13, and concern the languages Persian, Thai, and Somali. The Thai result shows one of the extremes among the typical results with the three implementations starting at about the same running time but then diverging, while the Somali plot shows the other extreme, that is, when the running times differ significantly already at start. In all of these plots, BETTY displays a better performance than TIBURON, regardless of the task solved. The one atypical result is found in Figure 14, which is the result from the Latin file. Here we see that TIBURON is faster than BEST TREES for all values of N tested, although the slope of the TIBURON curve indicates that it will eventually surpass both BETTY curves with increasing N. To verify this, we ran the same experiment for $N = 120{,}000$, and confirm that it has indeed happened: TIBURON runs in 3,105 ms whereas BETTY runs in 2,681 and 3,055 ms for best runs and best trees, respectively. We conjecture that this unusual result depends on certain characteristics of the Latin corpus: The Latin and the Somali file are approximately the same size, but the Latin output trees are small and many represent

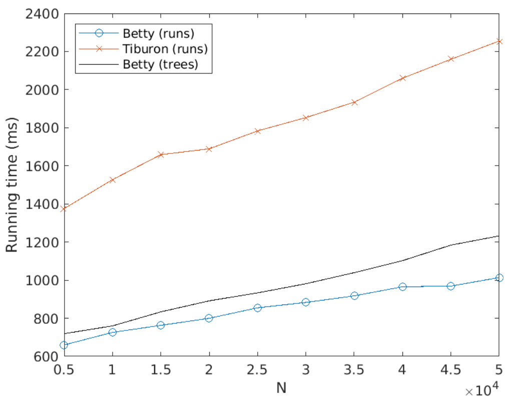


Figure 11
Comparison on the Persian file from `GF-data`.

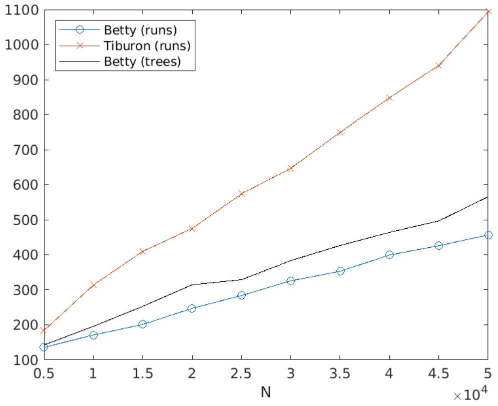


Figure 12
Comparison on the Thai file from `GF-data`.

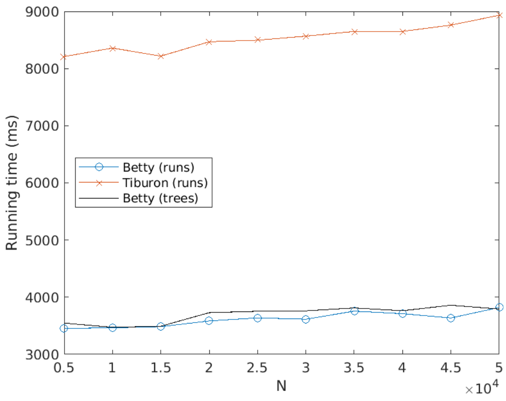


Figure 13
Comparison on the Somali file from `GF-data`.

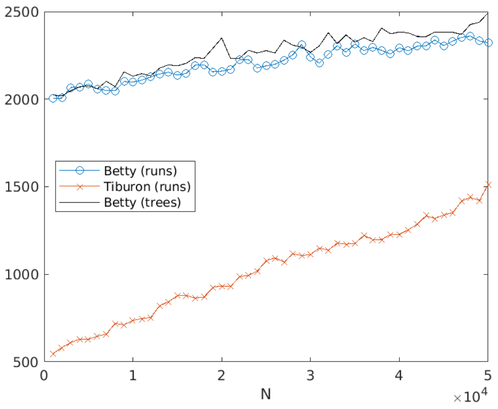


Figure 14
Comparison on the Latin file from `GF-data`.

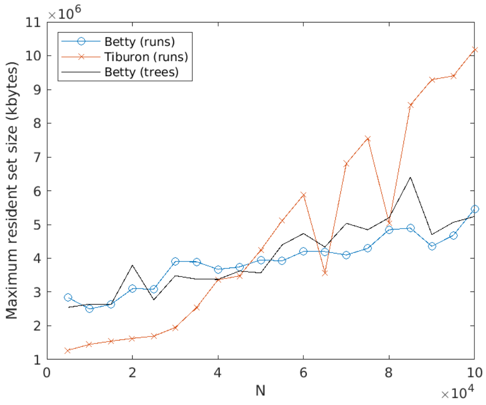


Figure 15
Comparison of the memory usage of all three implementations on the largest `MT-data` file.

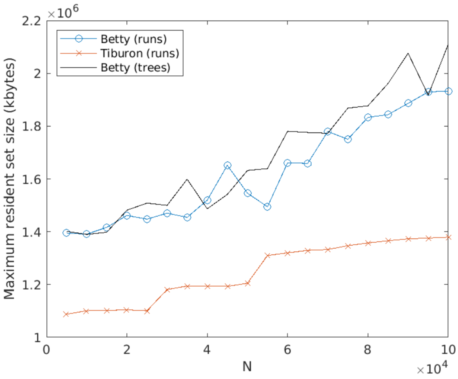


Figure 16
Comparison of the memory usage of all three implementations on the Persian `GF-data` file.

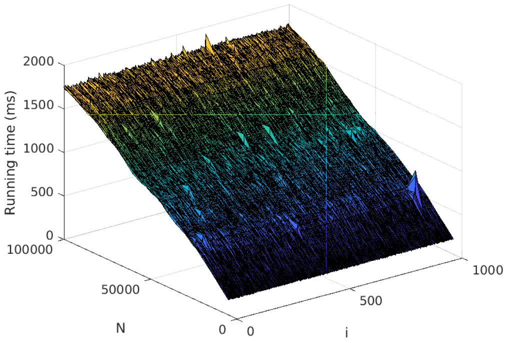


Figure 17
TIBURON solves the best runs problem for the `PolyNonDet` corpus.

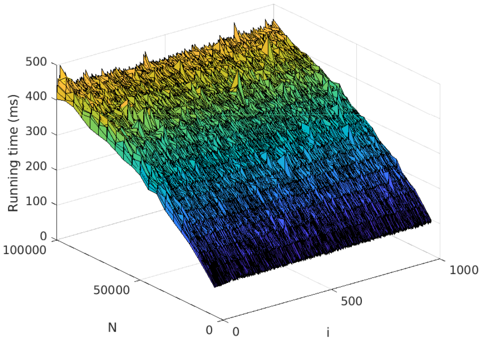


Figure 18
BETTY solves the best runs problem for the `PolyNonDet` corpus.

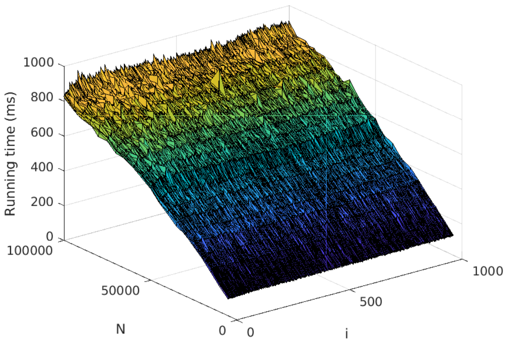


Figure 19
BETTY solves the best trees problem for the `PolyNonDet` corpus.

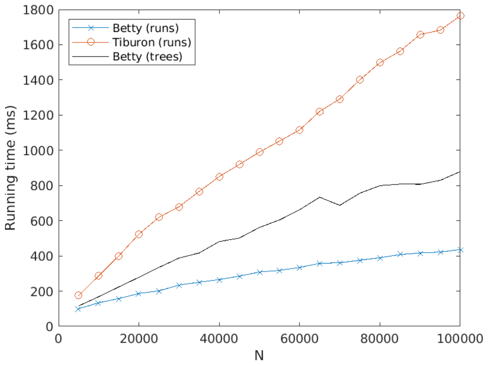


Figure 20
Comparison on the $i = 999$ `PolyNonDet` file.

one-word utterances, whereas the Somali output trees are larger and represent more complex sentence structures. The combination of a large file and small output trees makes the computation of the best contexts a large overhead, which is also visible in the plot: Even for zero output runs BETTY seems to require about 2 seconds of computation time. Even though this is a combination that we rarely see in practice, it is an example of when computing the best contexts could be considered superfluous and thus detrimental to efficiency.

Finally, we measure the memory usage for the various implementations on one file from each natural language corpus. The results for the largest `MT-data` file and the Persian `GF-data` file can be seen in figures 15 and 16, respectively. For the former, the memory usage of TIBURON seems to grow faster than the one for BETTY, but for the latter, the roles are reversed. To explain why this happens, the memory usage of the implementations needs to be investigated in greater detail, a task left for future work.

Now we have arrived at the artificial corpora that were created to investigate the effect of nondeterminism on the algorithms. Running TIBURON on the `PolyNonDet` corpus yields the running times in Figure 17, and using BETTY to solve the best runs and the best trees problems results in the numbers in figures 18 and 19, respectively.

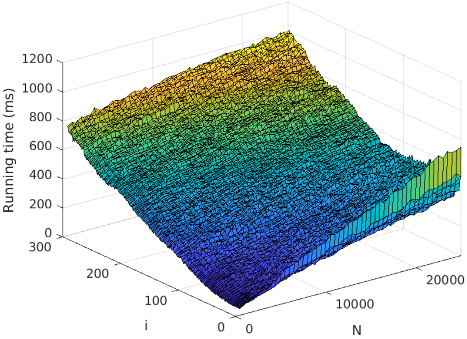


Figure 21
TIBURON solves the best runs problem for the `ExpNonDet` corpus.

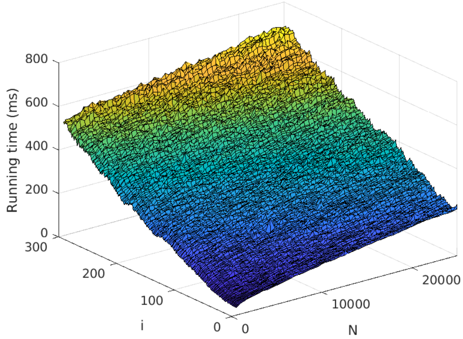


Figure 22
BETTY solves the best runs problem for the `ExpNonDet` corpus.

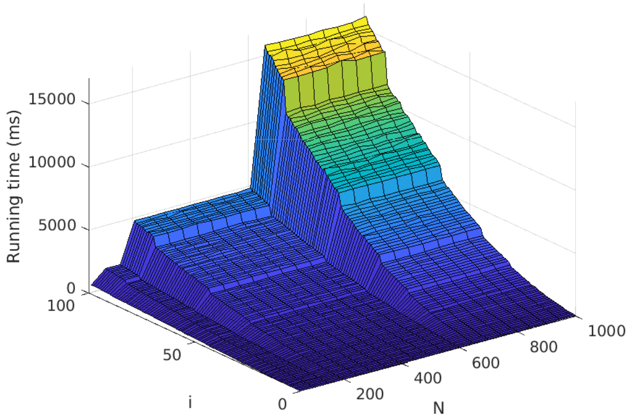


Figure 23
BETTY solves the best trees problem for the `ExpNonDet` corpus.

We observe that TIBURON's running times are significantly larger than those of BETTY. When solving the BEST TREES problem, BETTY does not seem to be affected by increasing i, which means it can all handle increasing degrees of polynomial nondeterminism well. By inspection of Figure 20, which compares the algorithms with respect to N, it is possible to conclude that they are all in $O(N)$ on this particular example.

We now use the `ExpNonDet` corpus to expose the algorithms to exponential nondeterminism. The expectation is that the best runs algorithms will continue to do their job well, simply because the exponential nondeterminism does not play a significant role in this case. Indeed, this turns out to be the case, with BETTY (Figure 22) being slightly faster than TIBURON (Figure 21). However, the best trees problem is no longer that easily solvable. As can be seen in Figure 23, we had to restrict the intervals for both i and N to make the task feasible for the computer used. This outcome is expected since there is now an exponential number of runs on each tree, and these duplicates have to be discarded before a larger tree can be found: The plateaus in Figure 23 indicate where we go from including trees of sizes smaller than s to including trees of size s. Asymptotic

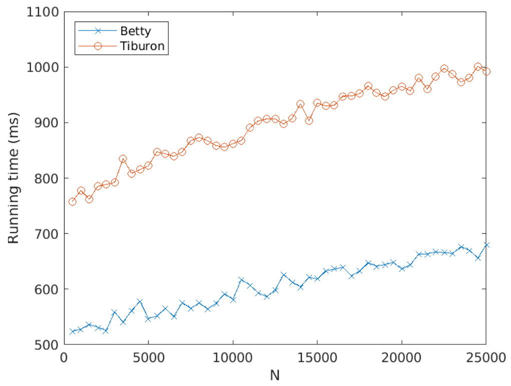


Figure 24
Comparison between TIBURON and BETTY for the best runs problem on the ExpNonDet corpus for $i = 299$.

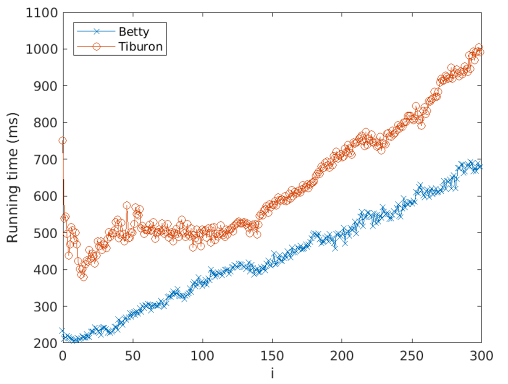


Figure 25
Comparison between TIBURON and BETTY for the best runs problem on the ExpNonDet corpus for $N = 25{,}000$.

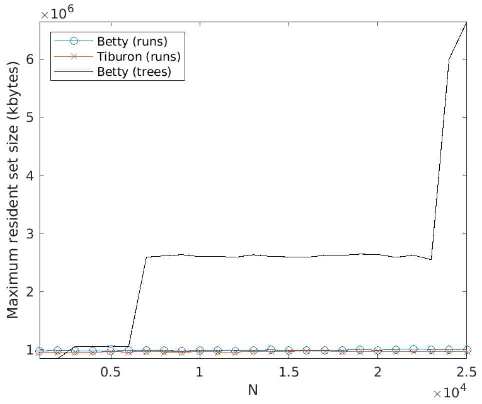


Figure 26
Comparison of the memory usage of the best runs implementations for the ExpNonDet file with $i = 299$, and with $i = 19$ for the best trees one.

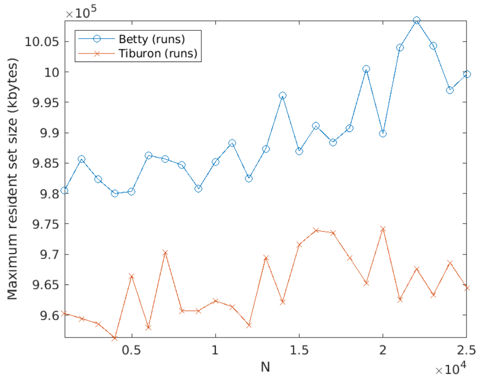


Figure 27
Comparison of the memory usage of the two best runs implementations for the $i = 299$ ExpNonDet file.

testing verifies that the best trees curve grows quadratically with i (meaning that it is linear in m) and is not worse than $N \log N$ for the second dimension. Figures 24 and 25 make a closer comparison of TIBURON and BETTY for the best runs task with respect to the variables N and i. The effect of increasing N is linear as for the PolyNonDet data, but when considering i, the previously constant behavior is now linear.

We conclude the experiments by presenting a smaller number of results for memory usage: Figures 26 and 27 show the memory usage measured in kbytes for varying N while figures 28 and 29 show the corresponding numbers for varying i. As expected, the memory usage is much larger for the best trees task than for the best runs task. Moreover, for the latter, TIBURON seems to have a slight advantage over BETTY with respect to i but a clear advantage with respect to N.

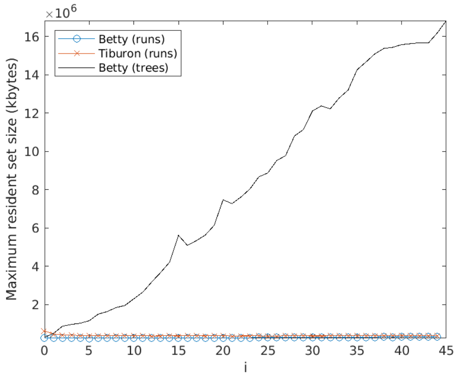


Figure 28
Comparison of the memory usage of all three implementations for $N = 25\,000$ on the $i = 0, \ldots, 45$ ExpNonDet files.

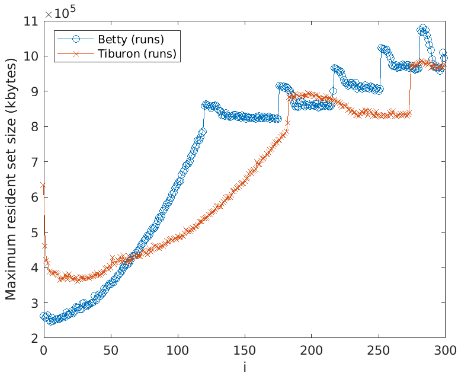


Figure 29
Comparison of the memory usage of the two best runs implementations for $N = 25\,000$ on the $i = 0, \ldots, 299$ ExpNonDet files.

8. Conclusion

We have presented an improved version of the algorithm by Björklund, Drewes, and Zechner (2019) 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 derived an upper bound on its running time—a bound that is smaller than that of the previous algorithm. We believe that this speed-up makes it superior for usage in typical language-processing applications.

Moreover, we have complemented the theoretical work with an experimental evaluation. Because BEST TREES can be easily modified to produce the best runs instead of the best trees, we considered both tasks in our evaluation. To achieve a large coverage, we used two types of data: data from real-world language processing tasks and artificially created data. The real-world data consist of machine translation output and corpus-based rule sets for natural languages; these corpora are meant to display the kind of behavior one can expect when applying our algorithm to language processing tasks. The artificial corpora were designed to expose BEST TREES to its worst-case scenario for the best trees task: the case when we have an exponential number of duplicate trees in a best runs list. We also covered the more moderate case where the nondeterminism only gives rise to a polynomial number of duplicates.

In the experiments focusing on running time, we first used the exponentially nondeterministic data to show that BEST TREES is better at the best trees task than its predecessor BEST TREES V.1 that uses a less efficient pruning scheme. Then, we compared BEST TREES with the state-of-the-art best-runs algorithm of Huang and Chiang (2005), implemented in TIBURON by May and Knight (2006) for both tasks on all data sets. The results made it clear that BEST TREES is preferable when extracting *N*-best lists of both runs and trees: BETTY outperforms TIBURON for the best runs task on all 2,269 input wtas except one. The single exception revealed a corner case where the Huang and Chiang algorithm is faster, namely, when there are millions of rules and only a small percentage of them are used to produce *N* very small (height 0 or 1) runs. Additionally, we performed a smaller number of experiments to measure the memory usage of the three applications, and, while no final conclusion could be made, it seemed as though TIBURON had an advantage over BEST TREES with respect to memory usage in total.

Prior to the experiments, we compared the two algorithms at a conceptual level and discussed the expected effects on their running time. The allocation of queues to transition rules instead of states mainly serves to structure the implementation. As we saw, it does not have a disadvantage with respect to running time. The use of best contexts, generalizing the idea of Mohri and Riley (2002) to trees, has a positive effect: it ensures that the maximum distance of a state to the final state is an upper bound on the maximum number of main loop iterations before the next run is outputted. This is because the best contexts guide the algorithm to take the shortest route in constructing the next best run that reaches a final state.[9]

The disadvantage of using best contexts is that it limits which semirings can be used. The technique is compatible with the tropical semiring and, as discussed in Section 2, with the equivalent Viterbi semiring, but seemingly not with semirings that are not extremal. It is currently unclear to us whether an appropriate extension is possible, and we leave this question for future work. However, such extensions seem only relevant for the *N*-best runs problem, because it appears highly unlikely that the *N*-best trees problem could ever be solved with reasonable efficiency in cases where the weight semiring is not extremal. The reason is that, in that case, the best run on a tree does not determine the weight of that tree, making it unclear how the problem can be solved even if disregarding efficiency aspects. However, because the tropical semiring and the Viterbi semiring are the most prominent ones used to rank hypotheses in NLP, the computational advantages seen in this article seem to justify the restriction to these semirings, even when looking only at the *N*-best runs problem.

Acknowledgments

We are thankful to Andreas Maletti for providing the machine translation data set used in our experiments; to Jonathan May for his support in the application of TIBURON to the *N*-best problem; to André Berg for sharing his expertise in mathematical statistics; to Aarne Ranta, Peter Ljunglöf, and Krasimir Angelov for introducing us to Grammatical Framework; and to the reviewers for suggesting numerous improvements to the article.

References

Benesty, Jacob, M. Mohan Sondhi, and Yiteng Huang, editors. 2008. *Springer Handbook of Speech Processing*. Springer. https://doi.org/10.1007/978-3-540-49127-9

Björklund, Henrik, Frank Drewes, and Petter Ericson. 2016. Between a rock and a hard place—parsing for hyperedge replacement DAG grammars. In *10th International Conference on Language and Automata Theory and Applications*, pages 521–532. https://doi.org/10.1007/978-3-319-30000-9_40

Björklund, Johanna, Frank Drewes, and Anna Jonsson. 2018. 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, pages 197–108. https://doi.org/10.1007/978-3-319-94812-6_9

Björklund, Johanna, Frank Drewes, and Niklas Zechner. 2019. Efficient enumeration of weighted tree languages over the tropical semiring. *Journal of Computer and System Sciences*, 104:119–130. https://doi.org/10.1016/j.jcss.2017.03.006

Büchse, Matthias, Daniel Geisler, Torsten Stüber, and Heiko Vogler. 2010. n-best

[9] We have conducted a small experiment not mentioned in the previous section, by switching off that feature. It showed that the use of best contexts (in that particular, randomly chosen case) reduced the size of rule queues by a factor of 10.

parsing revisited. In *Proceedings of the 2010 Workshop on Applications of Tree Automata in Natural Language Processing*, pages 46–54.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*, 3rd edition. The MIT Press.

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

Eppstein, David. 1998. Finding the *k* shortest paths. *SIAM Journal on Computing*, 28(2):652–673. https://doi.org/10.1137/S0097539795290477

Finkel, Jenny Rose, Christopher D. Manning, and Andrew Y. Ng. 2006. Solving the problem of cascading errors: Approximate Bayesian inference for linguistic annotation pipelines. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 618–626. https://doi.org/10.3115/1610075.1610162

Huang, Liang and David Chiang. 2005. Better *k*-best parsing. In *Proceedings of the Conference on Parsing Technology 2005*, pages 53–64. https://doi.org/10.3115/1654494.1654500

Jiménez, Víctor M. and Andrés Marzal. 2000. Computation of the N best parse trees for weighted and stochastic context-free grammars. In *Advances in Pattern Recognition*, pages 183–192. Springer. https://doi.org/10.1007/3-540-44522-6_19

Jonsson, Anna. 2021. *Best Trees Extraction and Contextual Grammars for Language Processing*. Ph.D. thesis, Umeå University.

Jurafsky, Dan and James H. Martin. 2009. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Pearson Prentice Hall.

Knight, Kevin and Jonathan Graehl. 2005. An overview of probabilistic tree transducers for natural language processing. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 1–24. https://doi.org/10.1007/978-3-540-30586-6_1

Knuth, Donald E. 1974. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673. https://doi.org/10.1145/361604.361612

Knuth, Donald E. 1977. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6:1–5. https://doi.org/10.1016/0020-0190(77)90002-3

Lyngsø, Rune B. and Christian N. S. Pedersen. 2002. The consensus string problem and the complexity of comparing hidden Markov models. *Journal of Computer and System Sciences*, 65(3):545–569. Special Issue on Computational Biology 2002. https://doi.org/10.1016/S0022-0000(02)00009-0

May, Jonathan and Kevin Knight. 2006. Tiburon: A weighted tree automata toolkit. In *International Conference on Implementation and Application of Automata*, pages 102–113. https://doi.org/10.1007/11812128_11

Mohri, Mehryar and Michael Riley. 2002. An efficient algorithm for the *n*-best-strings problem. In *Proceedings of the Conference on Spoken Language Processing (ICSLP 02)*, pages 1313–1316.

Quernheim, Daniel. 2017. *Bimorphism Machine Translation*. Ph.D. thesis, Universität Leipzig.

Ranta, Aarne. 2011. *Grammatical Framework: Programming with Multilingual Grammars*, CSLI Publications, Stanford.

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

Steedman, Mark. 1987. Combinatory grammars and parasitic gaps. *Natural Language and Linguistic Theory*, 5:403–439. https://doi.org/10.1007/BF00134555

Zhao, Yanpeng, Liwen Zhang, and Kewei Tu. 2018. Gaussian mixture latent vector grammars. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Volume 1: Long Papers*, pages 1181–1189. https://doi.org/10.18653/v1/P18-1109