A Comparative Evaluation of the Efficiency of N-Best Algorithms on Language Data

Johanna Björklund*
Umeå University

Frank Drewes**
Umeå University

Anna Jonsson†
Umeå University

The N-best extraction problem consists in selecting the N highest ranking hypotheses from a set of hypotheses, with respect to a given ranking system. In our setting, the hypotheses and ranking are jointly represented by a weighted tree automaton (wta) over the tropical semiring: the hypotheses are trees, or runs on trees, and the ranking is decided by the weight assigned to them. In previous work, we presented an algorithm for N-best extraction that combines techniques to restrict the search space, and proved it to be correct and efficient. The algorithm is now implemented in the software BETTY*, allowing us to complement the deductive study with an empirical investigation. In particular, we compare our algorithm to the state-of-the-art algorithm for extracting the N best runs, implemented in 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.*

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, that is 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, e.g., 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

* E-mail: johanna@cs.umu.se.
** E-mail: drewes@cs.umu.se.
† E-mail: aj@cs.umu.se.

computation. This problem is known as the *best trees problem*. The additive operator $\oplus$ that combines weights of runs is *extremal* if $a \oplus b \in \{a, b\}$ for all weights a and b. When the operator has this property, the weight of a tree coincides with the weight of one of its runs, and there is a *natural order* on the weights, given by $a \leq b$ if and only if $a \oplus b = a$. This is for example the case when performing a Viterbi analysis: the transition rules are equipped with probabilities, which are multiplied over a given run to determine the probability of that run, and weights of alternative runs on a tree t are combined using $\oplus = \max$ to obtain the weight of t. An equivalent setting is the one in which transition weights are summed up to obtain the weight of a run, and the run with the minimal weight determines the weight of t.

In these cases, the best trees problem can be solved by computing a list of $N \cdot c$ best runs for M, for some constant c, and discarding runs made on previously outputted trees. A complicating factor is that M can have exponentially many runs on a single tree, so c may have to be very large to guarantee that the output contains N distinct trees.

We note that the best trees problem has applications also 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 this would mean deciding graph isomorphism, which is in general not known to be tractable.

In this work, we report on an empirical comparison of a recent algorithm capable of extracting both the best trees and best runs, and a state-of-the-art algorithm by Huang and Chiang (2005) for best runs. This adds to a line of research that spans two decades. It originates with an algorithm by Mohri and Riley (2002) to compute the N best strings with respect to a weighted string automaton. 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 a *determinized* version of the input automaton M, which can be exponentially larger than M, but has at most one run on each input string. Independently, Huang and Chiang (2005) published an algorithm that computes the best runs for tree automata. The algorithm combines dynamic programming and lazy evaluation to keep the number of intermediate computations small.

The algorithm by Huang and Chiang (2005) is implemented in the widely referenced Tiburon toolkit[1] (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 initial application for the implementation 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. The algorithm by Huang and Chiang (2005) is 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)[2].

1 https://github.com/isi-nlp/tiburon

2 This replacement had already been done in TIBURON but it had not been further commented upon.

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 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.

Björklund, Drewes, and Zechner (2019) generalize the best strings algorithm by Mohri and Riley (2002), resulting in the algorithm BEST TREES V.1 that solves the best trees problem in polynomial time. The algorithm includes an on-the-fly determinization that serves to keep the object automaton small, and is presented together with formal correctness and complexity proofs. In contrast to the current paper, which focuses mainly on the N-best runs problem, Björklund, Drewes, and Jonsson (2018) compare BEST TREES V.1 with TIBURON on the N-best trees problem, letting TIBURON enumerate runs until N distinct trees have been found. The conclusion is that TIBURON performs better when the input automaton is largely deterministic, that is, when there is a small number of runs on each input tree, but when the input automaton shows a higher degree of nondeterminism, meaning that number of runs grows rapidly in the size of the input tree, BEST TREES V.1 is faster.

Björklund, Drewes, and Jonsson (2019) continue to improve BEST TREES V.1 by queuing intermediate results with respect to transition rules rather than states, leading to the algorithm BEST TREES V.2. Henceforth, we will refer to the latter by the name BEST TREES. In addition to solving the best trees problem, BETTY can also solve the best runs problem. In this paper, we make use of this possibility in order to compare BEST TREES, implemented as BETTY[3], with TIBURON with respect to the computation of best runs. For our experiments, we use both largely deterministic wta from a machine translation project and from a grammatical framework, and more nondeterministic wta that were artificially created to expose the algorithms to challenging instances. Our results show that BETTY is generally more 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 that it computes best runs with).

2. Preliminaries

In this paper, $\mathbb{N}$ denotes the set of nonnegative integers (including zero), and $\mathbb{R}_+$ stands for the set of non-negative reals. We furthermore let $\mathbb{R}_+^\infty$ denote $\mathbb{R}_+ \cup \{\infty\}$. A sequence of 1's of length l is denoted by 1^l. For $n \in \mathbb{N}$, the set $\{i \in \mathbb{N} \mid 1 \leq i \leq n\}$ is abbreviated by $[n]$; in particular, $[0] = \emptyset$. As usual, the set of all strings (or: finite sequences) over a set A is denoted by A^*, and the empty string by λ.

A *ranked alphabet* is a finite set Σ of symbols partitioned into disjoint subsets $\Sigma_{(k)}$, $k \in \mathbb{N}$, i.e., $\Sigma = \biguplus_{k\in\mathbb{N}} \Sigma_{(k)}$. The *rank* of $f \in \Sigma$, denoted by $\mathrm{rank}(f)$, is the unique $k \in \mathbb{N}$ for which $f \in \Sigma_{(k)}$. Given such a ranked alphabet Σ and any set T, we denote by $\Sigma(T)$ the set of formal expressions

$$\Sigma(T) = \{f[t_1, \ldots, t_k] \mid k \in \mathbb{N}, f \in \Sigma_{(k)}, \text{ and } t_1, \ldots, t_k \in T\} \ .$$

3 https://github.com/tm11ajn/betty

The set of *trees over* Σ is the smallest set T_Σ such that $T_\Sigma = \Sigma(T_\Sigma)$.

Throughout the paper, let $\square \notin \Sigma$ be a special symbol of rank 0. The set C_Σ of *contexts over* Σ is the set of trees in $T_{\Sigma\cup\{\square\}}$ containing exactly one occurrence of $\square$. We define the *depth* of $c \in C_\Sigma$, denoted by $\mathrm{depth}(c)$, and the *substitution* of a tree t for $\square$ in c, denoted by $c[\![t]\!]$, as follows: If $c = \square$, then $\mathrm{depth}(c) = 0$ and $c[\![t]\!] = t$. Otherwise, if $c = f[t_1, \ldots, t_{i-1}, c', t_{i+1}, \ldots, t_k]$ with $c' \in C_\Sigma$, then $\mathrm{depth}(c) = \mathrm{depth}(c') + 1$ and $c[\![t]\!] = f[t_1, \ldots, t_{i-1}, c'[\![t]\!], t_{i+1}, \ldots, t_k]$.

A *weighted tree language* (over the tropical semiring) is a mapping $L\colon T_\Sigma \to \mathbb{R}_+^\infty$. Such languages can be specified by *weighted tree automata*. Here, we take a slightly different but equivalent view corresponding to the one of Huang and Chiang (2005) and Büchse et al. (2010) in order to facilitate comparison.

Recall that ordinary finite automata can be represented as state-transition diagrams, i.e., directed edge-labeled graphs whose nodes and edges correspond to states and transitions, respectively. A path from a (node representing an) initial state to a final one corresponds to an accepting run of the automaton on the string of edges labels on that path. If the string automaton is weighted, additional weights are assigned to the edges, and those are summed up along a path, yielding the weight of the run. For a given string, the minimal weight of a run on that string is the weight of the string (and ∞ if there is no such run).

In a similar way, (weighted) tree automata can be represented as hypergraphs whose hyperedges generalize ordinary binary edges in that they may have any number of source nodes, but still only one target node. A hyperedge labeled with an input symbol of rank k must have k source nodes. The formal definition reads as follows.

Definition 1
Let Σ be a ranked alphabet. A *weighted hypergraph* over Σ and $\mathbb{R}_+$ (for short: hypergraph) is a triple $G = (V, E, \mathrm{wt})$ where:

- V is a finite sets of *nodes*,
- $E \subseteq V^* \times \Sigma \times V$ is the set of *hyperedges*, and
- $\mathrm{wt}\colon E \to \mathbb{R}_+$ assigns a weight to every hyperedge.

For $e = (v_1 \cdots v_k, f, v) \in E$, we define $\mathrm{rank}(e) = k$ (thus turning E into a ranked alphabet) and require that $\mathrm{rank}(e) = \mathrm{rank}(f)$. We denote the three components of e by $\mathrm{tail}(e)$, $\mathrm{lab}(e)$, and $\mathrm{head}(e)$, i.e., $\mathrm{tail}(e) = v_1 \cdots v_k$, $\mathrm{lab}(e) = f$ and $\mathrm{head}(e) = v$. The homomorphisms $\mathrm{wt}\colon T_E \cup C_E \to \mathbb{R}_+$ and $\mathrm{result}\colon T_E \cup C_E \to T_\Sigma$ are given as follows:

1. $\mathrm{wt}(\square) = 0$ and $\mathrm{result}(\square) = \square$.
2. For $\rho = e[\rho_1, \ldots, \rho_k] \in T_E \cup C_E$, where $e \in E$,

$$\mathrm{wt}(\rho) = \mathrm{wt}(e) + \sum_{i\in[k]} \mathrm{wt}(\rho_i) \text{ and } \mathrm{result}(\rho) = \mathrm{lab}(e)[\mathrm{result}(\rho_1), \ldots, \mathrm{result}(\rho_k)] \ .$$

Next, we define *runs*, which in the hypergraph setting become trees over hyperedges.

Definition 2
Let $G = (V, E, \mathrm{wt})$ be a hypergraph. For $v \in V$, the sets $\mathrm{runs}_G(v)$ of *runs arriving at* v are the smallest subsets of T_E such that, for every edge $e \in E$ with $\mathrm{tail}(e) = v_1 \cdots v_k$, if $\rho_1 \in \mathrm{runs}_G(v_1), \ldots, \rho_k \in \mathrm{runs}_G(v_k)$ then $e[\rho_1, \ldots, \rho_k] \in \mathrm{runs}_G(\mathrm{head}(e))$.

For a final node[4] $v_{\mathrm{f}} \in V$, the *weighted tree language accepted at* v_{f} *in* G is given by

$$L_G^{v_{\mathrm{f}}(t)} = \min\{\mathrm{wt}(\rho) \mid \rho \in \mathrm{runs}_G(v_{\mathrm{f}}) \text{ and } \mathrm{result}(\rho) = t\}$$

for every tree $t \in T_\Sigma$.

Given a hypergraph G as above, a final node $v_{\mathrm{f}} \in V$, and some $N \in \mathbb{N}$, the *N-best runs problem* asks to compute a sequence of N runs of smallest possible weight in $\mathrm{runs}_G(v_{\mathrm{f}})$. More precisely, the goal is to compute a sequence of pairwise distinct runs $\rho_1, \rho_2, \ldots, \rho_N \in \mathrm{runs}_G(v_{\mathrm{f}})$ such that there do not exist $i \in [N]$ and $\rho \in \mathrm{runs}_G(v_{\mathrm{f}}) \setminus \{\rho_1, \ldots, \rho_i\}$ with $\mathrm{wt}(\rho) < \mathrm{wt}(\rho_i)$. Similarly, the *$N$-best trees problem* asks to compute a sequence of pairwise distinct trees $t_1, t_2, \ldots, t_N$ in T_Σ of smallest possible weight, i.e., 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 $L_G^{v_{\mathrm{f}}(t)} < L_G^{v_{\mathrm{f}}(t_i)}$. To make sure that the two problems always have a solution, we assume in the following that $\mathrm{runs}_G(v_{\mathrm{f}})$ is infinite. Further, we assume that every $e \in E$ occurs in some run of $\mathrm{runs}_G(v_{\mathrm{f}})$. In automata terminology, this latter condition means that the automaton does not contain useless states, and it can be ensured in the usual way.

We end this section by discussing the choice of our particular weight structure, the tropical semiring. In general, 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, $\otimes$ distributes over $\oplus$ from both left and right, and 0 is an annihilator for $\otimes$, i.e., $0 \otimes d = 0 = d \otimes 0$ for all $d \in \mathbb{D}$. With this, Definitions 1 and 2 can be generalized to weighted hypergraphs over arbitrary commutative semirings, and their resulting weighted tree languages, simply by replacing $+$ and $\max$ by $\oplus$ and $\otimes$, respectively.

The semiring used in Definitions 1 and 2 is the *tropical semiring* $(\mathbb{R}_+^\infty, \min, +, \infty, 0)$, which is frequently used in NLP. 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 hypergraph G over the Viterbi semiring to a hypergraph G over the tropical semiring by simply mapping every hyperedge weight p to $-\ln p$, i.e., taking negative logarithms everywhere. Since $-\ln p + -\ln p' = -\ln(p \cdot p')$ and $-\ln p < -\ln p' \iff p > p'$ it holds that $L_G^{v_{\mathrm{f}}(t)}$ (now calculated using the Viterbi semiring operations) is equal to $\exp(-L_H^{v_{\mathrm{f}}(t)})$ for all trees t, and trees $t_1, \ldots, t_N$ form an N-best list according to $L_G^{v_{\mathrm{f}}}$ (now looking for trees with *maximal* weights) if and only if they form an N-best list according to $L_H^{v_{\mathrm{f}}}$ in the sense defined above.

3. Algorithms

In this section we relate BEST TREES to the algorithms of Huang and Chiang (2005) and Büchse et al. (2010).

3.1 The algorithm BEST TREES

Let us first describe the algorithm BEST TREES by Björklund, Drewes, and Jonsson (2019) in the hypergraph setting. Its pseudocode is outlined in Algorithm 1. As input it takes a triple (G, v_{f}, N), where $G = (V, E, \mathrm{wt})$ is a hypergraph, $v_{\mathrm{f}} \in V$ is a final node, and $N \in \mathbb{N}$. The objective is to solve the N-best trees problem or, for the sake of comparison with the TIBURON implementation of the algorithm by Huang and Chiang (2005), the N-best runs problem. From the point of view of computational complexity, the latter is the easier problem of the two. Essentially, the difference is that an algorithm solving the former must discard duplicate trees

4 We use this terminology because v_{f} corresponds to the *final state* of a tree automaton represented by G.

Algorithm 1 Compute $N \in \mathbb{N}$ best trees given a hypergraph $G = (V, E, \mathrm{wt})$ and a final node $v_\mathrm{f} \in V$ of final nodes

procedure BEST TREES(G, v_f, N)
  compute $(c_v, t_v^{\mathrm{best}})$ for all $v \in V$
  $T_v \leftarrow$ LIST-MAKE(t_v^{best}) for all $v \in V$
  for $e \in E$ **do**
    $U \leftarrow \begin{cases} \mathrm{inc}(\langle i_1, \ldots, i_{\mathrm{rank}(e)} \rangle) & \text{if } e = c_{\mathrm{head}(e)}(\lambda) \\ \{1^{\mathrm{rank}(e)}\} & \text{otherwise} \end{cases}$
    $K_e \leftarrow$ HEAP-MAKE(U) for all $e \in E$
  end for
  $K \leftarrow$ HEAP-MAKE($\{K_e \mid e \in E\}$)
  output($t_{v_\mathrm{f}}^{\mathrm{best}}$)
  repeat
    $K_e \leftarrow$ HEAP-EXTRACT-MIN(K)
    $(i_1, \ldots, i_{\mathrm{rank}(e)}) \leftarrow$ HEAP-EXTRACT-MIN(K_e)
    HEAP-INSERT(K, K_e)
    $t_i \leftarrow$ LIST-GET($T_{\mathrm{head}(e)}, i_j$) for $j \in [\mathrm{rank}(e)]$
    $t \leftarrow \mathrm{lab}(e)[t_1, \ldots, t_{\mathrm{rank}(e)}]$
    if t is not yet in $T_{\mathrm{head}(e)}$ **then**
      LIST-ADD($T_{\mathrm{head}(e)}, t$)
    end if
    if $\mathrm{head}(e) = v_\mathrm{f}$ **then**
      if t has not yet been outputted **then**
        output(t)
      end if
    end if
    HEAP-INSERT($K_e, \mathrm{inc}(\langle i_1, \ldots, i_{\mathrm{rank}(e)} \rangle)$)
    HEAP-UPDATE-PRIORITIES(K)
  until N trees have been outputted
end procedure

encountered, that is, equal results of distinct runs. For the sake of efficiency, this should be done as early as possible in the computation.

A central fact underlying Algorithm 1 is that, for every $v \in V$, there is a *best context* $c_v \in C_E$ such that

1. $c_v[\![\rho]\!] \in \mathrm{runs}_G(v_\mathrm{f})$ for every run $\rho \in \mathrm{runs}_G(v)$ and
2. among all contexts that satisfy the preceding condition, c_v minimizes $(\mathrm{wt}(c_v), \mathrm{depth}(c_v))$ with respect to the lexicographic order[5].

Suitable contexts c_v can be efficiently computed by a procedure based on an algorithm by Knuth (1974). Here, the role of Knuth's algorithm is to compute, for every node $v \in V$, a best tree t_v^{best}, that is, a tree t which minimizes $L_G^{v(t)}$. In fact, Algorithm 1 does not make much use of the contexts c_v themselves but mainly uses the pairs $(\omega_v, \delta_v) = (\mathrm{wt}(c_v), \mathrm{depth}(c_v))$ for every $v \in V$. Intuitively, c_v is the most economical way to extend any run ρ arriving at v to a run ρ' arriving at v_f, minimizing the weight of ρ'. Note that this depends only on v and not on ρ itself. As we will see below, we can use the pairs (ω_v, δ_v) to guide the search for best trees.

To understand Algorithm 1, let us first consider the easier N-best runs problem. For every $v \in V$, let $T_v(1), T_v(2), \ldots$ be an enumeration (without repetition) of $\mathrm{runs}_G(v)$ according

5 A pair $(w, n) \in \mathbb{R}_+ \times \mathbb{N}$ is lexicographically less than another such pair (w', n') if $w < w'$ or $w = w'$ and $n < n'$.

to increasing weight. Then, for a hyperedge $e \in E$ with $\mathrm{tail}(e) = v_1 \cdots v_k$, every run $\rho = e[\rho_1, \ldots, \rho_k]$ can uniquely be specified by e and the tuple $(i_1, \ldots, i_k)$ such that $\rho_j = T_{v_j}(i_j)$ for all $j \in [k]$. We write $\rho = e\langle i_1, \ldots, i_k \rangle$. Now, in order to *construct* the lists T_v run by run, we keep, for every hyperedge e as above, a queue K_e of those tuples $(i_1, \ldots, i_k)$ for which $e\langle i_1, \ldots, i_k \rangle$ is not yet in $T_{\mathrm{head}(e)}$. For a given $(i_1, \ldots, i_k)$ in K_e, $e\langle i_1, \ldots, i_k \rangle$ may not yet be available because some T_{v_j} does not yet contain i_j runs. Call those which *are* available the *current candidates*, and consider, for every run $\rho \in \mathrm{runs}_G(v)$, the pair $\Pi(\rho) = (\omega_v + \mathrm{wt}(\rho), \delta_v)$, where ω_v and δ_v are given by the best context c_v for v as discussed above. It turns out that if we, among the set of all current candidates ρ, always choose the one that minimizes $\Pi(\rho)$ and append it to the appropriate T_v, then this ρ is the one that minimizes $\Pi(\rho)$ among *all* runs not yet in any T_v. Hence, ρ is indeed an appropriate next element of T_v. In addition, if $v = v_{\mathrm{f}}$, ρ is written to the output.

To make the procedure efficient, we make two central observations (see (Björklund, Drewes, and Jonsson 2019) for details). First, in order to find N best runs, no more than the first N elements of each T_v need to be stored; any additional ones can be discarded should they be encountered as candidates. Second, $i_1 \leq j_1, \ldots, i_k \leq j_k$ implies $\mathrm{wt}(e\langle i_1, \ldots, i_k \rangle) \leq \mathrm{wt}(e\langle j_1, \ldots, j_k \rangle)$, which means that at each moment in time only the minimal still unused elements of $\mathbb{N}^k$ need to be kept in K_e, for each $e \in E$ of rank k. (Here, *minimal* refers to the partial order given by $(i_1, \ldots, i_k) \leq (j_1, \ldots, j_k)$ if and only if $i_\ell \leq j_\ell$ for all $\ell \in [k]$.)

Consider a hyperedge e with $\mathrm{tail}(e) = v_1 \cdots v_k$ and $\mathrm{head}(e) = v$. We need quick access to a tuple $(i_1, \ldots, i_k)$ in K_e that determines a current candidate $\rho = e\langle i_1, \ldots, i_k \rangle$ which minimizes $\Pi(\rho) = (\omega_v + \mathrm{wt}(\rho), \delta_v)$. We do this by defining the priority of $(i_1, \ldots, i_k)$ in K_e to be $\Pi(\rho)$ if $|T_{v_j}| \geq i_j$ for all $j \in [k]$ (i.e., if ρ is indeed a current candidate). Otherwise, the priority of $(i_1, \ldots, i_k)$ is $(\infty, 0)$, in effect disabling this potential candidate until it eventually becomes available. Finally, to be able to access the run $\rho = e\langle i_1, \ldots, i_k \rangle$ that, across all K_e, minimizes $\Pi(\rho)$, we also need quick access to the queue K_e whose highest-priority element has the highest priority across all K_e. To facilitate this, we keep a *main queue* K whose elements are the individual queues K_e, the priority of each K_e in K being the priority of its highest priority element $(e, \langle i_1, \ldots, i_k \rangle)$.

To obtain an algorithm computing best trees rather than best runs, we discard runs resulting in duplicate trees as soon as possible. That is, a candidate run ρ is appended to T_v at position $i + 1 \leq N$ only if $\mathrm{result}(\rho) \neq \mathrm{result}(T_v(j))$ for all $j \in [i]$. Moreover, $\mathrm{result}(\rho)$ is written to the output only if $v = v_{\mathrm{f}}$ and $\mathrm{result}(\rho)$ has not been outputted before (as the result of another run arriving at v_{f}).

The pseudocode in Algorithm 1 shows the main steps of the procedure, using min-heaps to implement priority queues. The initialization phase prior to the main loop computes a best context c_v and a best tree t_v^{best} for every node v. The tree t_v^{best} is inserted in T_v. Every queue K_e initially either contains the element $1^{\mathrm{rank}(e)}$ or, if e happens to be the edge used to arrive at v in the best tree for v, all of the successors of $1^{\mathrm{rank}(e)}$. (See the end of Section 3.3 for an alternative that allows us to avoid the initialization of each T_v with t_v^{best}, instead initializing T_v as the empty list and K_e as the queue containing only $1^{\mathrm{rank}(e)}$ for all $v \in V$ and $e \in E$.)

We note that line 13 re-inserts queue K_e into K, which is needed because K_e has just been extracted from K as its top-most element, and must thus be re-inserted with a lower priority as its own top-most element $(e, \langle i_1, \ldots, i_{\mathrm{rank}(e)} \rangle)$ was extracted. Line 24 inserts the set of all $(e, \langle j_1, \ldots, j_{\mathrm{rank}(e)} \rangle$ into e such that $j_\ell = i_\ell + 1$ for exactly one $\ell \in [\mathrm{rank}(e)]$ and $j_{\ell'} = i_{\ell'}$ for all $\ell' \in [\mathrm{rank}(e)] \setminus \{\ell\}$.[6] Finally, line 25 increases the priorities of all elements of K (i.e., of all

6 To avoid duplicates to appear in K_e, we implement inc in such a way that it returns only those $\langle j_1, \ldots, j_{\mathrm{rank}(e)} \rangle$ which are not yet present in K_e.

K_e) in which the priority of the top-most element has changed due to the addition of t to $T_{\mathrm{head}(e)}$ or due to the insertion of new tuples in line 24.

Algorithm 1 solves the N-best runs problem instead of the N-best trees problem if we replace $\mathrm{lab}(e)$ by e on line 15 and make lines 17 and 21 unconditional.

3.2 The algorithm BEST RUNS

We now recall the algorithm by Huang and Chiang (2005), which we henceforth will refer to as BEST RUNS. The type of hypergraph used as input to BEST RUNS differs slightly from the one used in the previous section in that the admissible weight structures are not restricted to the tropical semiring. Instead, each hyperedge e is equipped with a weight function $\mathrm{wt}_e \colon \mathbb{R}_+^{\mathrm{rank}(e)} \to \mathbb{R}_+$. The definition of the weight of a run $\rho = e[\rho_1, \ldots, \rho_k]$ is then changed to $\mathrm{wt}(\rho) = \mathrm{wt}_e(\mathrm{wt}(\rho_1), \ldots, \mathrm{wt}(\rho_k))$.

For the algorithm to work, these weight functions wt_e are required to be monotonic: $\mathrm{wt}_e(w_1, \ldots, w_k) \geq \mathrm{wt}_e(w'_1, \ldots, w'_k)$ 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 $\mathrm{wt}_e(w_1, \ldots, w_k) = w + \sum_{i \in [k]} w_i$ 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 3.3.

The input to BEST RUNS is a triple (G, v_{f}, N), where the components are as for BEST TREES. The algorithm is outlined in Algorithm 2. Line 2 is a preprocessing step that can be performed in $O(|E|)$ time using the Viterbi algorithm, given that the rank of the alphabet used is considered a constant. A list $res[v]$ is used to store, for each node $v \in V$, the at most N discovered best runs arriving at v. The search space is represented by an array of heaps, here denoted $cands$. For each node v, $cands[v]$ holds a heap storing the (at most N) best, so far unexploited, runs arriving at v. That is, if we have already picked the M 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 M-th best run ρ is used as follows: if $\mathrm{tail}(e) = v_1 \cdots v_k$, we obtain a new candidate by replacing the i-th direct subtree of ρ with the next (and thereby minimally worse) run in res arriving at v_i.

3.3 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 16 and 20 of Algorithm 1, duplicate trees are discarded. If these conditions are removed, BEST TREES solves the best runs problem. (Strictly speaking, a further change required for this is that the tree t constructed on line 15 must be defined as $e[t_1, \ldots, t_{\mathrm{rank}(e)}]$ rather than as $\mathrm{lab}(e)[t_1, \ldots, t_{\mathrm{rank}(e)}]$.)

Following Huang and Chiang (2005), let us assume that the ranked alphabet Σ is fixed. The running time of BEST TREES is $O(N\,|E|\,(\log|E| + \log N))$ (Björklund, Drewes, and Jonsson 2019). Since $|E| \in O(|V|^{r+1})$, where r is the maximal rank of symbols in Σ, it follows that $\log|E| \in O(\log|V|)$, so the second expression simplifies to $N\,|E|\,(\log|V| + \log N) \leq |E| \log|V|\, N \log N$. As shown by Huang and Chiang (2005), the worst case running time of BEST RUNS is $O(|E| \log|V| + mN \log N)$ where m is the size of the largest run among the

Algorithm 2 Compute $N \in \mathbb{N}$ runs arriving at $v_\mathrm{f} \in V = [|V|]$ of minimal weight according to a hypergraph $G = (V, E, \mathrm{wt})$ (Huang and Chiang 2005).

```
 1: procedure BEST RUNS(G, v_f, N)
 2:     Compute 1-best run ρ_v^best arriving at v, for every v ∈ V
 3:     res[v] ← LIST-EMPTY() for all v ∈ V
 4:     LIST-ADD(res[v], ρ_v^best) for all v ∈ V
 5:     BEST RUNS_0(v_f, N)
 6:     for ρ ∈ res[v_f] do
 7:         output(ρ)
 8:     end for
 9: end procedure
10:
11: procedure BEST RUNS_0(v, N)
12:     if |res[v]| ≥ N then
13:         return
14:     end if
15:     if cands[v] is undefined then
16:         tmp ← SORT-BY-WEIGHT({e[ρ_{v_1}^best, ..., ρ_{v_k}^best] | e ∈ E and v_1, ..., v_k ∈ V where
                                                                  head(e) = v and v_1 ··· v_k = tail(e)})
17:         cands[v] ← HEAP-IFY(tmp[1] ... tmp[N])
18:         LIST-ADD(res[v], HEAP-EXTRACT-MIN(cands[v]))
19:     end if
20:     while LIST-SIZE(res[v]) < N and HEAP-SIZE(cands[v]) > 0 do
21:         s ← LIST-SIZE(res[v])
22:         ρ ← LIST-GET(res[v], s)                          ▷ the s-best run arriving at v
23:         NEXT RUNS(cands[v], ρ)                           ▷ build new candidates from ρ
24:         LIST-ADD(res[v], HEAP-EXTRACT-MIN(cands[v]))
25:     end while
26: end procedure
27:
28: procedure NEXT RUNS(vcands, ρ = (e[ρ_1, ..., ρ_rank(e)]))
29:     for i ← 1, ..., rank(e) do
30:         list ← res[root(ρ_i)]
31:         N' ← index of ρ in res[root(ρ)]
32:         BEST RUNS_0(root(ρ_i), N')                       ▷ make sure list has N' ≤ N elements
33:         if N' ≤ LIST-SIZE(list) then
34:             ρ'_i ← LIST-GET(list, N')
35:             ρ' ← e[ρ_1, ..., ρ_{i-1}, ρ'_i, ρ_{i+1}, ..., ρ_rank(e)]
36:             if ρ' ∉ vcands then
37:                 HEAP-INSERT(vcands, ρ')
38:             end if
39:         end if
40:     end for
41: end procedure
```

N results.[7] Assuming that N is large enough to make the second term the dominating one, the difference between both running times is thus a factor of $(|E| \log |V|)/m$ (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 m depends on both N and the structure of G, and the algorithms are specialized on different problems.

7 In fact, the running time obtained by Huang and Chiang (2005) is $O(|E| + mN \log N)$, but this assumes G to be acyclic. When G is cyclic, line 2 must be implemented by using Knuth's algorithm, resulting in an additional factor $\log |V|$ in the first term.

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 $O(N\,|E|)$ candidates whereas BEST RUNS needs only $O(N\,|V|)$. 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 16 and 20). To see this, consider a given node $v \in V$. Each time a queue element $(i_1, \ldots, i_{\mathrm{rank}(e)})$ with $\mathrm{head}(e) = v$ is dequeued from a queue K_e, the corresponding run is appended to the list T_v on line 17, and $\mathrm{rank}(e)$ queue elements are inserted on line 24. As this can only happen at most N times per node v, in total only $O(N\,|V|)$ 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_v for every node v in order to explore the search space in a more goal-oriented fashion.[8] (Recall the discussion in the second paragraph of Section 3.1.) Let ℓ be the maximum of the distances of nodes in V to v_f, i.e.,

$$\ell = \max_{v \in V} \mathrm{dist}(v, v_\mathrm{f}),$$

where $dist(v, v_\mathrm{f})$ denotes the length of the shortest sequence of hyperedges $e_0 \cdots e_n$ such that $v \in [\mathrm{tail}(e_0)]$, $\mathrm{head}(e_{i-1}) \in [\mathrm{tail}(e_i)]$ for all $i \in [n]$, and $v_\mathrm{f} = \mathrm{head}(e_n)$. Then, 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 $\rho = e[\cdots]$ arriving at a node v is constructed. The priority of the corresponding queue element on line 12 is given by $(\mathrm{wt}(\rho) + w, d)$, where w and d are the weight and depth of c_v. If $v \neq v_\mathrm{f}$, then $c_v \neq \Box$, and thus it is of the form $c_u[\![e'[\rho_1, \ldots, \rho_{i-1}, \Box, \rho_{i+1}, \ldots, \rho_k]]\!]$, where the ρ_j are the 1-best runs arriving at their respective nodes (see Figure 1).

It follows that line 24 inserts the element into K_e that represents the run $\rho' = e'[\rho_1, \ldots, \rho_{i-1}, \rho, \rho_{i+1}, \ldots, \rho_k]$. Since each of the runs ρ_i, arriving at some node u_i, is already in the respective list T_{u_i}, the run ρ' immediately becomes a current candidate arriving at u. The corresponding pair $(w_u, d_u) = (\mathrm{wt}(c_u), \mathrm{depth}(c_u))$ satisfies $w_u + \mathrm{wt}(\rho') = w + \mathrm{wt}(\rho)$ and $d_u = d - 1$. 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 v_f, i.e., the run will be outputted. This also shows that queues whose elements are not needed for generating output trees, will never become filled beyond their initial element.

The reader may have noticed that the discussion above also explains why the best trees t_v^{best}, $v \in V$, are inserted into each T_v prior to the start of the main loop, rather than letting T_v be empty and initializing K_e to $1^{\mathrm{rank}(e)}$ for all $v \in V$ and $e \in E$: 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 $T_{\mathrm{head}(e)}$. If this happens, edges of weight 0 may repeatedly be picked because e' in the reasoning above is not yet enabled. However, with a trick the initialization can nevertheless be simplified in the way mentioned. The idea is to delay the execution of line 25 for every queue K_e until e has actually appeared in an output tree. Thus, until this has happened, K_e is

8 The possibility to use this optimization depends on the use of the tropical semiring as weight structure. That does not seem to be possible in the more general setting of Büchse et al. (2010).

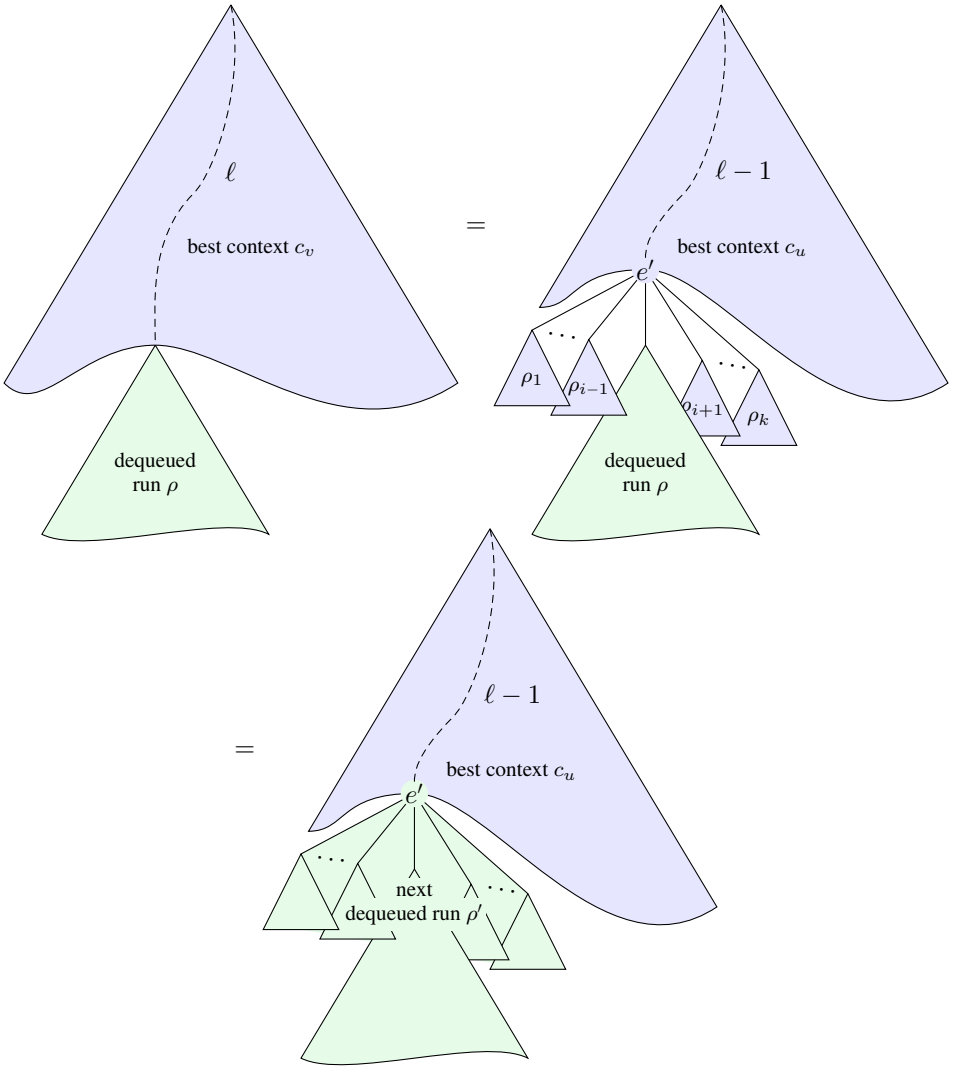


Figure 1
When a run ρ reaching node v is dequeued with $\ell > 0$ (top left), then its best context c_v is of the form $c_u[\![e'[\rho_1, \ldots, \rho_{i-1}, \square, \rho_{i+1}, \ldots, \rho_k]]\!]$, where c_u is the best context for $u = \mathrm{head}(e')$ and the ρ_i are 1-best trees reaching the remaining tail nodes of e' (top right). Hence, $\rho' = e'[\rho_1, \ldots, \rho_{i-1}, \rho, \rho_{i+1}, \ldots, \rho_k]$ (or a run which likewise is less than ℓ steps away from an output run) will be the next run dequeued (bottom).

disabled from contributing another tree to $T_{\mathrm{head}(e)}$. This variant turned out to have efficiency advantages in practice and has therefore been implemented in BETTY. Another slight advantage of this variant is that BETTY can handle a *set* of final nodes rather than only a single one. This is not conveniently possible when using the initialization of Algorithm 1. The reason is that line 9 would have to be generalized to outputting the best trees of all final nodes, which cannot be done while maintaining the correctness of the algorithm, because the second best tree for a node v may have a lesser weight than the best tree for another node v'.

4. 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, e.g., garbage collection, we only measure the time used by the actual application by timing its thread. Moreover, 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 variance 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 shown to be sufficient to fulfill the requirement for practically all instances seen.

All test scripts are written in Python. We run the experiments on a computer with a 3.60 Hz Intel Core i7-4790 processor.

4.1 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` 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 practically 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). 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. 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 grammars' designer, and do not reflect inherent complexities of the languages.)

`PolyNonDet` A family of grammars of increasing size indexed by i. The rules for member i follow (the final state is q_i).

- $a \xrightarrow{0} q_j$ 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` A family of grammars of increasing size, also indexed by i. The final state is q_f for all members. 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 $\Omega((i+1)^n)$, and we say that these are *exponentially nondeterministic*.

The variables in all result-displaying plots in this paper are either N (the number of trees or runs to output), m (the number of hyperedges or rules 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 exchangeable, but for the latter, m grows quadratically with increasing i.

4.2 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 task. The `ExpNonDet` data set was chosen since it has the largest degree of nondeterminism achievable, which should challenge both of the algorithms maximally. The results are presented in Figures 2 and 3 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 for both increasing N and increasing i.

More interestingly, in Figure 2, BEST TREES display 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 we must 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.

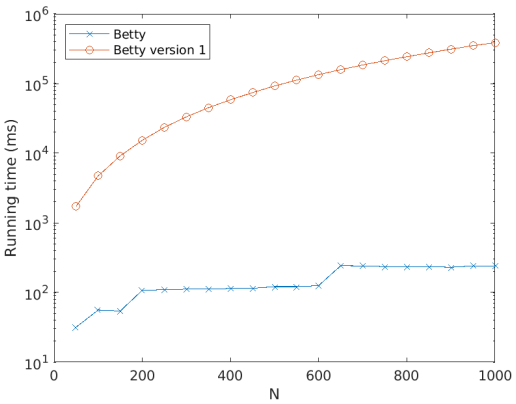


Figure 2
Comparison on the `ExpNonDet` file with $i = 7$.

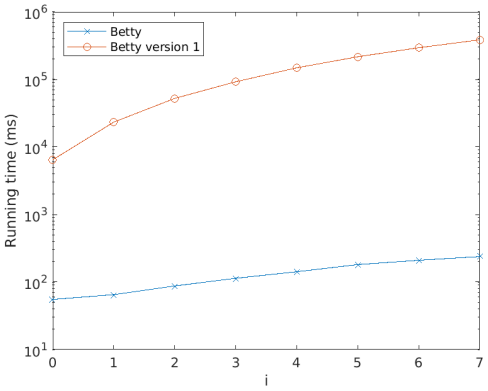


Figure 3
Comparison for $N = 1000$ on the `ExpNonDet` corpus.

4.3 Comparison with TIBURON

Now we compare BETTY with TIBURON for all data sets, starting with the `MT-data` corpus. Figure 4 shows the running times for finding the N best runs using TIBURON for every file of the `MT-data` corpus and $N = 1\,000, 2\,000, \ldots, 20\,000$. The corresponding results for BETTY can be seen in Figure 5. When we instead let BETTY solve the best trees problem for the same

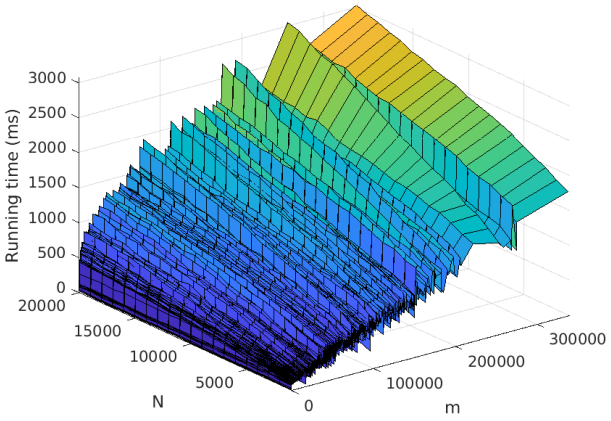


Figure 4
TIBURON solves the best runs problem for the `MT-data` corpus.

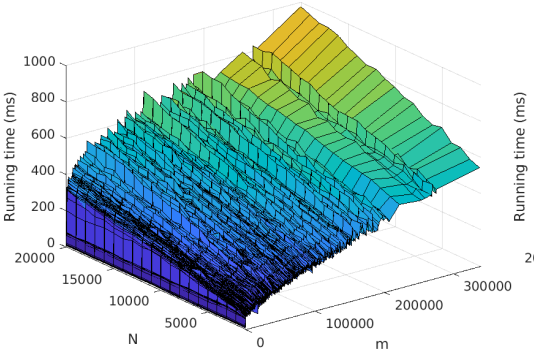


Figure 5
BETTY solves the best runs problem for the `MT-data` corpus.

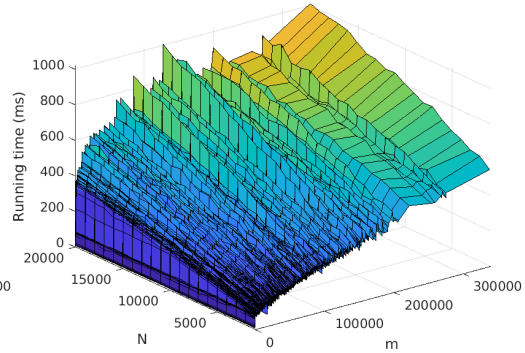


Figure 6
BETTY solves the best trees problem for the `MT-data` corpus.

input, we achieve the result in Figure 6. It seems that TIBURON is slower than BETTY for both tasks, but to be certain, we perform a statistical test. Since 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. 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. Table 1 shows the result of a Wilcoxon signed rank test on our data. We only report on the outcome of the test comparing BETTY on the best trees task with TIBURON, since the best runs task yields nearly identical results. Each table value corresponds to one test for two paired vectors v_{BETTY} and v_{TIBURON}. For our statistical test, we use a significance level of 0.01 and can therefore conclude that H_0 is rejected for all partitions of our data (note that each table value is of order 10^{-18}). 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 the statistical tests.

Table 1
Probabilities of, under the null hypothesis H_0 that there is no difference between BETTY and TIBURON, observing values as or more extreme than the data, with the alternative hypothesis H_1 that BETTY on the best trees task is faster than TIBURON on the best runs task for `MT-data`.

N	Increasing file sizes, groups of 103 files $[\cdot 10^{-18}]$								
1 000	0.6514	0.6327	0.9530	0.7767	0.6514	0.6326	0.6327	0.6327	0.6326
2 000	0.6326	0.6326	0.6326	0.7218	0.6326	0.6327	0.6327	0.6327	0.6326
3 000	0.6326	0.6326	0.6327	0.6708	0.6327	0.6326	0.6326	0.6326	0.6327
4 000	0.6326	0.6327	0.6327	0.6709	0.6326	0.6326	0.6326	0.6327	0.6327
5 000	0.6327	0.6326	0.6325	0.6709	0.6326	0.6326	0.6327	0.6327	0.6327
6 000	0.6326	0.6326	0.6326	0.6708	0.6327	0.6327	0.6327	0.6326	0.6327
7 000	0.6327	0.6324	0.6326	0.6709	0.6326	0.6326	0.6326	0.6326	0.6327
8 000	0.6327	0.6327	0.6327	0.6709	0.6326	0.6326	0.6327	0.6326	0.6327
9 000	0.6327	0.6327	0.6326	0.6709	0.6327	0.6327	0.6326	0.6326	0.6327
10 000	0.6327	0.6327	0.6326	0.6709	0.6326	0.6326	0.6327	0.6326	0.6326
11 000	0.6327	0.6327	0.6327	0.6709	0.6327	0.6327	0.6326	0.6327	0.6327
12 000	0.6326	0.6327	0.6327	0.6707	0.6326	0.6327	0.6327	0.6326	0.6327
13 000	0.6327	0.6326	0.6327	0.6709	0.6326	0.6326	0.6327	0.6326	0.6327
14 000	0.6327	0.6327	0.6327	0.6709	0.6326	0.6326	0.6327	0.6327	0.6327
15 000	0.6327	0.6327	0.6327	0.6709	0.6326	0.6325	0.6327	0.6327	0.6327
16 000	0.6326	0.6326	0.6326	0.6708	0.6326	0.6327	0.6327	0.6326	0.6327
17 000	0.6327	0.6327	0.6326	0.6709	0.6326	0.6327	0.6515	0.6326	0.6327
18 000	0.6327	0.6327	0.6327	0.6709	0.6515	0.6326	0.6327	0.6326	0.6327
19 000	0.6326	0.6327	0.6327	0.6709	0.6327	0.6327	0.6327	0.6327	0.6327
20 000	0.6327	0.6327	0.6327	0.6515	0.6327	0.6327	0.6327	0.6327	0.6327

Next, we apply the three algorithms to the `GF-data` corpus. We run tests for 42 different languages, and present a selection of typical and atypical results. The typical results are shown in Figures 7, 8 and 9, and concern the languages Persian, Thai, and Somali. In all of these plots, TIBURON displays a slower performance than BETTY, regardless of what task is solved. The single atypical result is found in Figure 10, 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. We conjecture that this 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 very small and many represent 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 though this is a combination that we rarely see in practice, it is an interesting example of when computing the best contexts could be considered superfluous.

Now we have arrived at the artificial corpora which were created to investigate the effect of nondeterminism on the algorithms. Running TIBURON on the `PolyNonDet` corpus yields the running times in Figure 11, and using BETTY to solve the best runs and the best trees problems results in the numbers in Figures 12 and 13, respectively. We observe that TIBURON's running times are significantly larger than those of BETTY. None of the algorithms seems to be affected by increasing i, which means that they can all handle increasing degrees of polynomial nondeterminism well. By inspection of Figure 14 that compares the algorithms with respect to N, it is possible to conclude that they are all in $O(N)$ on this particular example.

When we instead use the `ExpNonDet` corpus to expose the algorithms to exponential nondeterminism, the best runs algorithms still do the job well, with BETTY (Figure 16) being

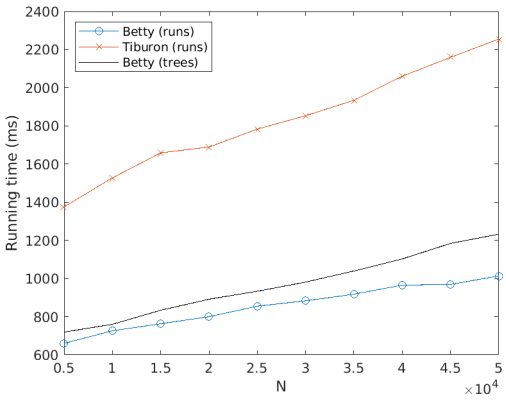


Figure 7
Comparison on the Persian file from `GF-data`.

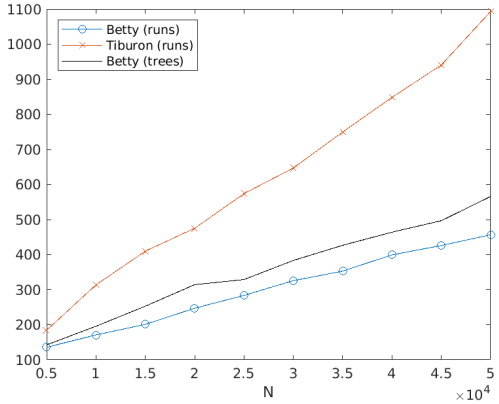


Figure 8
Comparison on the Thai file from `GF-data`.

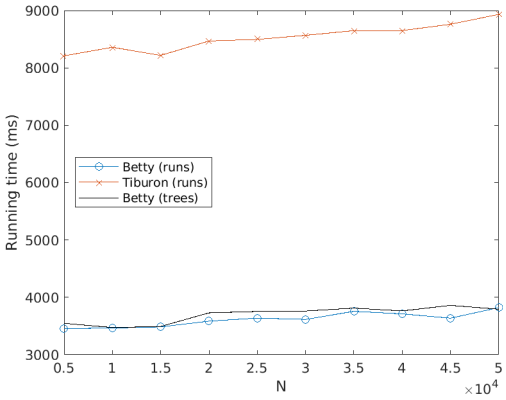


Figure 9
Comparison on the Somali file from `GF-data`.

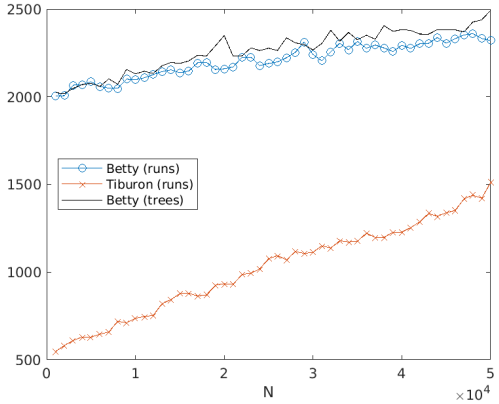


Figure 10
Comparison on the Latin file from `GF-data`.

slightly faster than TIBURON (Figure 15). However, the best trees problem is no longer that easily solvable. As can be seen in Figure 17, we had to restrict the intervals for both i and N to make the task feasible. 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 17 indicate where we go from including trees of sizes smaller than s to including trees of size s. Asymptotic 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 18 and 19 make a closer comparison of TIBURON and BETTY for the best runs task with respect to the variables N and i, respectively. The effect of increasing N is linear as for the `PolyNonDet` data, but when considering i, the previously constant behavior is now linear.

5. Conclusion

We have complemented previous theoretical work on the BEST TREES algorithm (Björklund, Drewes, and Jonsson 2019) with an experimental evaluation. Since 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 consists of machine translation

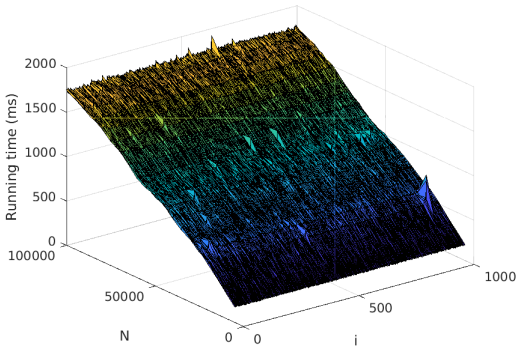


Figure 11
TIBURON solves the best runs problem for the `PolyNonDet` corpus.

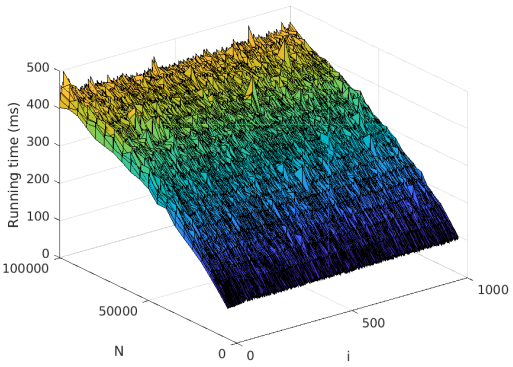


Figure 12
BETTY solves the best runs problem for the `PolyNonDet` corpus.

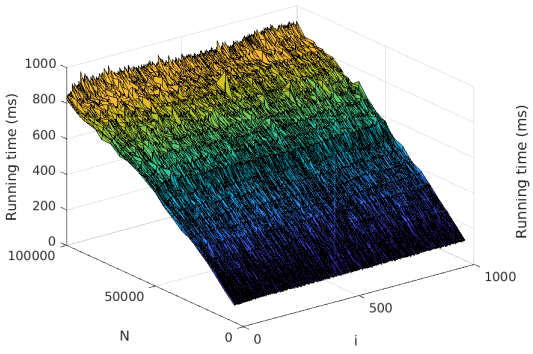


Figure 13
BETTY solves the best trees problem for the `PolyNonDet` corpus.

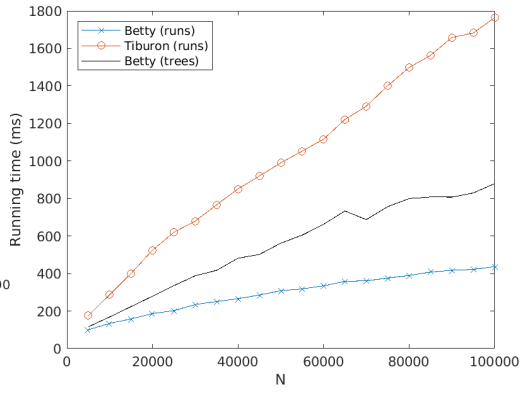


Figure 14
Comparison on the `PolyNonDet` file with $i = 999$.

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, 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.

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 hyperedges instead of

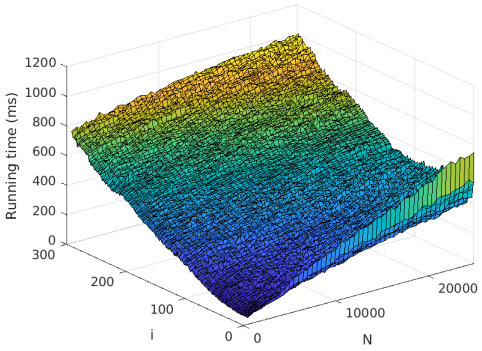


Figure 15
TIBURON solves the best runs problem for the `ExpNonDet` corpus.

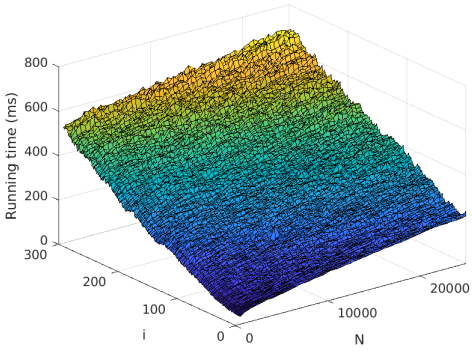


Figure 16
BETTY solves the best runs problem for the `ExpNonDet` corpus.

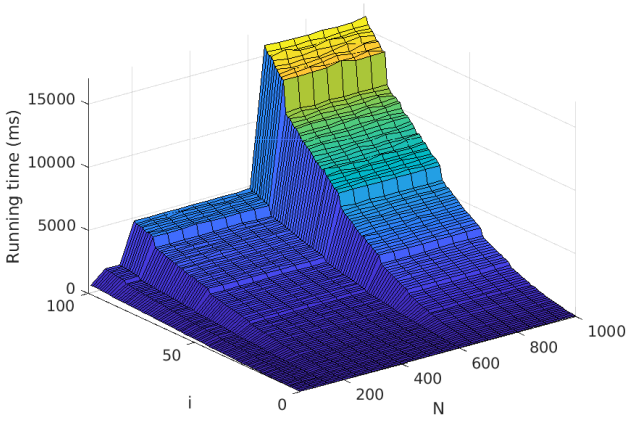


Figure 17
BETTY solves the best trees problem for the `ExpNonDet` corpus.

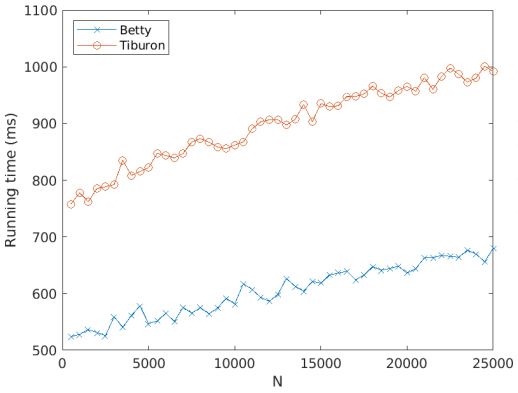


Figure 18
Comparison between TIBURON and BETTY for the best runs problem on the `ExpNonDet` corpus for $i = 299$.

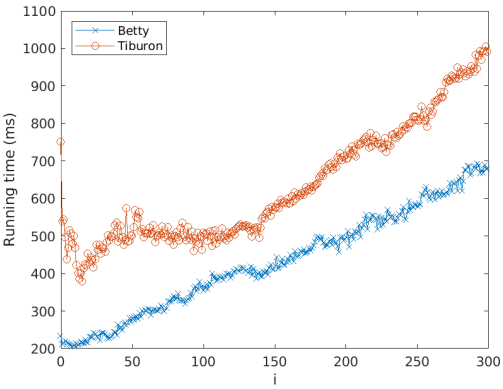


Figure 19
Comparison between TIBURON and BETTY for the best runs problem on the `ExpNonDet` corpus for $N = 25000$.

nodes 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 between a node and a 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 what 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. However, since the tropical semiring and the Viterbi semiring are the most prominent ones used to rank hypotheses in NLP, the computational advantages seen in this paper seem to justify the restriction to these semirings, even when looking only at the N-best runs problem.

Acknowledgements

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; and to Aarne Ranta, Peter Ljunglöf, and Krasimir Angelov for introducing us to Grammatical Framework.

References

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*.

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.

Björklund, Johanna, Frank Drewes, and Anna Jonsson. 2019. Faster computation of N-best lists for weighted tree automata. Under review.

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.

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

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

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, Association for Computational Linguistics, Sydney, Australia.

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

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.

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.

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, Springer.

Knuth, Donald E. 1974. Computer Programming as an Art. *Commun. ACM*, 17(12):667–673.

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

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.

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

Steedman, Mark. 1987. Combinatory grammars and parasitic gaps. *Natural Language and Linguistic Theory*, 5:403–439.