Better k-best Parsing

Liang Huang
Dept. of Computer & Information Science
University of Pennsylvania
3330 Walnut Street
Philadelphia, PA 19104
lhuang3@cis.upenn.edu

David Chiang
Inst. for Advanced Computer Studies
University of Maryland
3161 AV Williams
College Park, MD 20742
dchiang@umiacs.umd.edu

Abstract

We discuss the relevance of *k-best parsing* to recent applications in natural language processing, and develop efficient algorithms for k-best trees in the framework of hypergraph parsing. To demonstrate the efficiency, scalability and accuracy of these algorithms, we present experiments on Bikel's implementation of Collins' lexicalized PCFG model, and on Chiang's CFG-based decoder for hierarchical phrase-based translation. We show in particular how the improved output of our algorithms has the potential to improve results from parse reranking systems and other applications.

1 Introduction

Many problems in natural language processing (NLP) involve optimizing some objective function over a set of possible analyses of an input string. This set is often exponential-sized but can be compactly represented by merging equivalent subanalyses. If the objective function is compatible with a packed representation, then it can be optimized efficiently by dynamic programming. For example, the distribution of parse trees for a given sentence under a PCFG can be represented as a packed forest from which the highest-probability tree can be easily extracted.

However, when the objective function f has no compatible packed representation, exact inference would be intractable. To alleviate this problem, one common approach from machine learning is loopy belief propagation (Pearl, 1988). Another solution (which is popular in NLP) is to split the computation into two phases: in the first phase, use some compatible objective function f' to produce a k-best list (the top k candidates under f'), which serves as an approximation to the full set. Then, in the second phase, optimize f over all the analyses in the k-best list. A typical example is discriminative reranking on k-best lists from a generative module, such as (Collins, 2000) for parsing and (Shen *et al.*, 2004) for translation, where the reranking model has nonlocal features that cannot be computed during parsing proper. Another example is minimum-Bayes-risk decoding (Kumar and Byrne, 2004; Goodman, 1998),where, assuming f' defines a probability distribution over all candidates, one seeks the candidate with the highest expected score according to an arbitrary metric (e.g., PARSEVAL or BLEU); since in general the metric will not be compatible with the parsing algorithm, the k-best lists can be used to approximate the full distribution f'. A similar situation occurs when the parser can produce multiple derivations that are regarded as equivalent (e.g., multiple lexicalized parse trees corresponding to the same unlexicalized parse tree); if we want the maximum *a posteriori* parse, we have to sum over equivalent derivations. Again, the equivalence relation will in general not be compatible with the parsing algorithm, so the k-best lists can be used to approximate f', as in Data Oriented Parsing (Bod, 2000) and in speech recognition (Mohri and Riley, 2002).

Another instance of this k-best approach is *cascaded optimization*. NLP systems are often cascades of modules, where we want to optimize the modules' objective functions jointly. However, often a module is incompatible with the packed representation of the previous module due to factors like non-local dependencies. So we might want to postpone some disambiguation by propagating k-best lists to subsequent phases, as in joint parsing and semantic role labeling (Gildea and Jurafsky, 2002; Sutton and McCallum, 2005), information extraction and coreference resolution (Wellner *et al.*, 2004), and formal semantics of TAG (Joshi and Vijay-Shanker, 1999).

Moreover, much recent work on *discriminative training* uses k-best lists; they are sometimes used to approximate the normalization constant or partition function (which would otherwise be intractable), or to train a model by optimizing some metric incompatible with the packed representation. For example, Och (2003) shows how to train a log-linear translation model not by maximizing the likelihood of training data, but maximizing the BLEU score (among other metrics) of the model on

Proceedings of the Ninth International Workshop on Parsing Technologies (IWPT), pages 53–64,
Vancouver, October 2005.

the data. Similarly, Chiang (2005) uses the k-best parsing algorithm described below in a CFG-based log-linear translation model in order to learn feature weights which maximize BLEU.

For algorithms whose packed representations are graphs, such as Hidden Markov Models and other finite-state methods, Ratnaparkhi's MXPARSE parser (Ratnaparkhi, 1997), and many stack-based machine translation decoders (Brown *et al.*, 1995; Och and Ney, 2004), the k-best paths problem is well-studied in both pure algorithmic context (see (Eppstein, 2001) and (Brander and Sinclair, 1995) for surveys) and NLP/Speech community (Mohri, 2002; Mohri and Riley, 2002). This paper, however, aims at the k-best tree algorithms whose packed representations are hypergraphs (Gallo *et al.*, 1993; Klein and Manning, 2001) (equivalently, and/or graphs or packed forests), which includes most parsers and parsing-based MT decoders. Any algorithm expressible as a weighted deductive system (Shieber *et al.*, 1995; Goodman, 1999; Nederhof, 2003) falls into this class. In our experiments, we apply the algorithms to the lexicalized PCFG parser of Bikel (2004), which is very similar to Collins' Model 2 (Collins, 2003), and to a synchronous CFG based machine translation system (Chiang, 2005).

2 Previous Work

As pointed out by Charniak and Johnson (2005), the major difficulty in k-best parsing is dynamic programming. The simplest method is to abandon dynamic programming and rely on aggressive pruning to maintain tractability, as is used in (Collins, 2000; Bikel, 2004). But this approach is prohibitively slow, and produces rather low-quality k-best lists (see Sec. 5.1.2). Gildea and Jurafsky (2002) described an $O(k^2)$-overhead extension for the CKY algorithm and reimplemented Collins' Model 1 to obtain k-best parses with an average of 14.9 parses per sentence. Their algorithm turns out to be a special case of our Algorithm 0 (Sec. 4.1), and is reported to also be prohibitively slow.

Since the original design of the algorithm described below, we have become aware of two efforts that are very closely related to ours, one by Jiménez and Marzal (2000) and another done in parallel to ours by Charniak and Johnson (2005). Jiménez and Marzal present an algorithm very similar to our Algorithm 3 (Sec. 4.4) while Charniak and Johnson propose using an algorithm similar to our Algorithm 0, but with multiple passes to improve efficiency. They apply this method to the Charniak (2000) parser to get 50-best lists for reranking, yielding an improvement in parsing accuracy.

Our work differs from Jiménez and Marzal's in the following three respects. First, we formulate the parsing problem in the more general framework of hypergraphs (Klein and Manning, 2001), making it applicable to a very wide variety of parsing algorithms, whereas Jiménez and Marzal define their algorithm as an extension of CKY, for CFGs in Chomsky Normal Form (CNF) only. This generalization is not only of theoretical importance, but also critical in the application to state-of-the-art parsers such as (Collins, 2003) and (Charniak, 2000). In Collins' parsing model, for instance, the rules are dynamically generated and include unary productions, making it very hard to convert to CNF by preprocessing, whereas our algorithms can be applied directly to these parsers. Second, our Algorithm 3 has an improvement over Jiménez and Marzal which leads to a slight theoretical and empirical speedup. Third, we have implemented our algorithms on top of state-of-the-art, large-scale statistical parser/decoders and report extensive experimental results while Jiménez and Marzal's was tested on relatively small grammars.

On the other hand, our algorithms are more scalable and much more general than the coarse-to-fine approach of Charniak and Johnson. In our experiments, we can obtain 10000-best lists nearly as fast as 1-best parsing, with very modest use of memory. Indeed, Charniak (p.c.) has adopted our Algorithm 3 into his own parser implementation and confirmed our findings.

In the literature of k shortest-path problems, Minieka (1974) generalized the Floyd algorithm in a way very similar to our Algorithm 0 and Lawler (1977) improved it using an idea similar to but a little slower than the binary branching case of our Algorithm 1. For hypergraphs, Gallo *et al.* (1993) study the shortest hyperpath problem and Nielsen *et al.* (2005) extend it to k shortest hyperpath. Our work differes from (Nielsen *et al.*, 2005) in two aspects. First, we solve the problem of k-best derivations (i.e., trees), not the k-best hyperpaths, although in many cases they coincide (see Sec. 3 for further discussions). Second, their work assumes non-negative costs (or probabilities ≤ 1) so that they can apply Dijkstra-like algorithms. Although generative models, being probability-based, do not suffer from this problem, more general models (e.g., log-linear models) may require negative edge costs (McDonald *et al.*, 2005; Taskar *et al.*, 2004). Our work, based on the Viterbi algorithm, is still applicable as long as the hypergraph is acyclic, and is used by McDonald *et al.* (2005) to get the k-best parses.

3 Formulation

Following Klein and Manning (2001), we use weighted directed hypergraphs (Gallo *et al.*, 1993) as an abstraction of the probabilistic parsing problem.

Definition 1. An *ordered hypergraph* (henceforth *hypergraph*) H is a tuple $\langle V, E, t, \mathbf{R} \rangle$, where V is a finite set of *vertices*, E is a finite set of *hyperarcs*, and $\mathbf{R}$ is the set of *weights*. Each hyperarc $e \in E$ is a triple

$e = \langle T(e), h(e), f(e)\rangle$, where $h(e) \in V$ is its *head* and $T(e) \in V^*$ is a vector of *tail* nodes. $f(e)$ is a *weight function* from $\mathbf{R}^{|T(e)|}$ to $\mathbf{R}$. $t \in V$ is a distinguished vertex called *target vertex*.

Note that our definition is different from those in previous work in the sense that the tails are now *vectors* rather than sets, so that we can allow multiple occurrences of the same vertex in a tail and there is an ordering among the components of a tail.

Definition 2. A hypergraph H is said to be *monotonic* if there is a total ordering $\preceq$ on $\mathbf{R}$ such that every weight function f in H is monotonic in each of its arguments according to $\preceq$, i.e., if $f : \mathbf{R}^m \mapsto \mathbf{R}$, then $\forall 1 \leq i \leq m$, if $a_i \preceq a_i'$, then $f(a_1, \cdots, a_i, \cdots, a_m) \preceq f(a_1, \cdots, a_i', \cdots, a_m)$. We also define the comparison function $\min_{\preceq}(a, b)$ to output a if $a \preceq b$, or b if otherwise.

In this paper we will assume this monotonicity, which corresponds to the optimal substructure property in dynamic programming (Cormen *et al.*, 2001).

Definition 3. We denote $|e| = |T(e)|$ to be the *arity* of the hyperarc. If $|e| = 0$, then $f(e) \in \mathbf{R}$ is a constant and we call $h(e)$ a *source vertex*. We define the *arity* of a hypergraph to be the maximum arity of its hyperarcs.

Definition 4. The *backward-star* $BS(v)$ of a vertex v is the set of incoming hyperarcs $\{e \in E \mid h(e) = v\}$. The *in-degree* of v is $|BS(v)|$.

Definition 5. A *derivation* D of a vertex v in a hypergraph H, its size $|D|$ and its weight $w(D)$ are recursively defined as follows:

- If $e \in BS(v)$ with $|e| = 0$, then $D = \langle e, \epsilon\rangle$ is a derivation of v, its size $|D| = 1$, and its weight $w(D) = f(e)()$.
- If $e \in BS(v)$ where $|e| > 0$ and D_i is a derivation of $T_i(e)$ for $1 \leq i \leq |e|$, then $D = \langle e, D_1 \cdots D_{|e|}\rangle$ is a derivation of v, its size $|D| = 1 + \sum_{i=1}^{|e|} |D_i|$ and its weight $w(D) = f(e)(w(D_1), \ldots, w(D_{|e|}))$.

The ordering on weights in $\mathbf{R}$ induces an ordering on derivations: $D \preceq D'$ iff $w(D) \preceq w(D')$.

Definition 6. Define $D_i(v)$ to be the i^{th}-best derivation of v. We can think of $D_1(v), \ldots, D_k(v)$ as the components of a vector we shall denote by $\mathbf{D}(v)$. The *k-best derivations problem* for hypergraphs, then, is to find $\mathbf{D}(t)$ given a hypergraph $\langle V, E, t, \mathbf{R}\rangle$.

With the derivations thus ranked, we can introduce a nonrecursive representation for derivations that is analogous to the use of *back-pointers* in parser implementation.

Definition 7. A *derivation with back-pointers (dbp)* $\hat{D}$ of v is a tuple $\langle e, \mathbf{j}\rangle$ such that $e \in BS(v)$, and $\mathbf{j} \in \{1, 2, \ldots, k\}^{|e|}$. There is a one-to-one correspondence $\sim$ between dbps of v and derivations of v:

$$\langle e, (j_1 \cdots j_{|e|})\rangle \sim \langle e, D_{j_1}(T_1(e)) \cdots D_{j_{|e|}}(T_{|e|}(e))\rangle$$

Accordingly, we extend the weight function w to dbps: $w(\hat{D}) = w(D)$ if $\hat{D} \sim D$. This in turn induces an ordering on dbps: $\hat{D} \preceq \hat{D}'$ iff $w(\hat{D}) \preceq w(\hat{D}')$. Let $\hat{D}_i(v)$ denote the i^{th}-best dbp of v.

Where no confusion will arise, we use the terms 'derivation' and 'dbp' interchangeably.

Computationally, then, the k-best problem can be stated as follows: given a hypergraph H with arity a, compute $\hat{D}_1(t), \ldots, \hat{D}_k(t)$.[1]

As shown by Klein and Manning (2001), hypergraphs can be used to represent the search space of most parsers (just as graphs, also known as trellises or lattices, can represent the search space of finite-state automata or HMMs). More generally, hypergraphs can be used to represent the search space of most *weighted deductive system* (Nederhof, 2003). For example, the weighted CKY algorithm given a context-free grammar $G = \langle N, T, P, S\rangle$ in Chomsky Normal Form (CNF) and an input string w can be represented as a hypergraph of arity 2 as follows. Each *item* $[X, i, j]$ is represented as a vertex v, corresponding to the recognition of nonterminal X spanning w from positions $i+1$ through j. For each production rule $X \to YZ$ in P and three free indices $i < j < k$, we have a hyperarc $\langle((Y, i, k), (Z, k, j)), (X, i, k), f\rangle$ corresponding to the *instantiation* of the inference rule COMPLETE in the deductive system of (Shieber *et al.*, 1995), and the weight function f is defined as $f(a, b) = ab \cdot \Pr(X \to YZ)$, which is the same as in (Nederhof, 2003). In this sense, hypergraphs can be thought of as *compiled* or *instantiated* versions of weighted deductive systems.

A parser does nothing more than traverse this hypergraph. In order that derivation values be computed correctly, however, we need to traverse the hypergraph in a particular order:

Definition 8. The *graph projection* of a hypergraph $H = \langle V, E, t, \mathbf{R}\rangle$ is a directed graph $G = \langle V, E'\rangle$ where $E' = \{(u, v) \mid \exists e \in BS(v), u \in T(e)\}$. A hypergraph H is said to be *acyclic* if its graph projection G is a directed acyclic graph; then a *topological ordering* of H is an ordering of V that is a topological ordering in G (from sources to target).

We assume the input hypergraph is acyclic so that we can use its topological ordering to traverse it. In practice the hypergraph is typically not known in advance, but the

[1] Note that although we have defined the weight of a derivation as a function on derivations, in practice one would store a derivation's weight inside the dbp itself, to avoid recomputing it over and over.

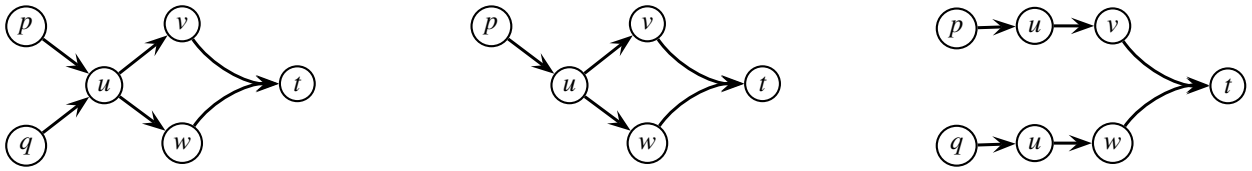


Figure 1: Examples of hypergraph, hyperpath, and derivation: (a) a hypergraph H, with t as the target vertex and p, q as source vertices, (b) a hyperpath π_t in H, and (c) a derivation of t in H, where vertex u appears twice with two different (sub-)derivations. This would be impossible in a hyperpath.

topological ordering often is, so that the (dynamic) hypergraph can be generated in that order. For example, for CKY it is sufficient to generate all items $[X, i, j]$ before all items $[Y, i', j']$ when $j' - i' > j - i$ (X and Y are arbitrary nonterminals).

Excursus: Derivations and Hyperpaths

The work of Klein and Manning (2001) introduces a correspondence between hyperpaths and derivations. When extended to the k-best case, however, that correspondence no longer holds.

Definition 9. (Nielsen *et al.*, 2005) Given a hypergraph $H = \langle V, E, t, \mathbf{R} \rangle$, a *hyperpath* π_v of destination $v \in V$ is an *acyclic minimal* hypergraph $H_\pi = \langle V_\pi, E_\pi, v, \mathbf{R} \rangle$ such that

1. $E_\pi \subseteq E$
2. $v \in V_\pi = \bigcup_{e \in E_\pi} (T(e) \cup \{h(e)\})$
3. $\forall u \in V_\pi$, u is either a source vertex or connected to a source vertex in H_π.

As illustrated by Figure 1, derivations (as trees) are different from hyperpaths (as minimal hypergraphs) in the sense that in a derivation the same vertex can appear more than once with possibly different sub-derivations while it is represented at most once in a hyperpath. Thus, the k-best derivations problem we solve in this paper is very different in nature from the k-shortest hyperpaths problem in (Nielsen *et al.*, 2005).

However, the two problems do coincide when $k = 1$ (since all the sub-derivations must be optimal) and for this reason the 1-best hyperpath algorithm in (Klein and Manning, 2001) is very similar to the 1-best tree algorithm in (Knuth, 1977). For k-best case ($k > 1$), they also coincide when the hypergraph is isomorphic to a Case-Factor Diagram (CFD) (McAllester *et al.*, 2004) (proof omitted). The derivation forest of CFG parsing under the CKY algorithm, for instance, can be represented as a CFD while the forest of Earley algorithm can not. An item (or equivalently, a vertex in hypergraph) can appear twice in an Earley derivation because of the prediction rule (see Figure 2 for an example).

$$\dfrac{\dfrac{(A \to \alpha.B\beta, i, j)}{(B \to .\gamma, j, j)}}{\dfrac{\cdots}{(B \to \gamma., j, k)}} \quad \cdots \qquad \dfrac{(A \to \alpha.B\beta, i, j) \quad (B \to \gamma., j, k)}{(A \to \alpha B.\beta, i, k)}$$

Figure 2: An Earley derivation. Note that item $(A \to \alpha.B\beta, i, j)$ appears twice (predict and complete).

```
1: procedure VITERBI(k)
2:     for v ∈ V in topological order do
3:         for e ∈ BS(v) do ▷ for all incoming hyperarcs
4:             D̂_1(v) ← min_≤(D̂_1(v), ⟨e, 1⟩)        ▷ update
```

Figure 3: The generic 1-best Viterbi algorithm

The k-best derivations problem has potentially more applications in tree generation (Knight and Graehl, 2005), which can not be modeled by hyperpaths. But detailed discussions along this line are out of the scope of this paper.

4 Algorithms

The traditional 1-best Viterbi algorithm traverses the hypergraph in topological order and for each vertex v, calculates its 1-best derivation $D_1(v)$ using all incoming hyperarcs $e \in BS(v)$ (see Figure 3). If we take the arity of the hypergraph to be constant, then the overall time complexity of this algorithm is $O(|E|)$.

4.1 Algorithm 0: naïve

Following (Goodman, 1999; Mohri, 2002), we isolate two basic operations in line 4 of the 1-best algorithm that

can be generalized in order to extend the algorithm: first, the formation of the derivation $\langle e, \mathbf{1} \rangle$ out of $|e|$ best subderivations (this is a generalization of the binary operator $\otimes$ in a semiring); second, $\min_{\preceq}$, which chooses the better of two derivations (same as the $\oplus$ operator in an *idempotent* semiring (Mohri, 2002)). We now generalize these two operations to operate on k-best lists.

Let $r = |e|$. The new multiplication operation, $\mathbf{mult}_{\preceq k}(e)$, is performed in three steps:

1. enumerate the k^r derivations $\{\langle e, j_1 \cdots j_r \rangle \mid \forall i, 1 \leq j_i \leq k\}$. Time: $O(k^r)$.
2. sort these k^r derivations (according to weight). Time: $O(k^r \log(k^r)) = O(rk^r \log k)$.
3. select the first k elements from the sorted list of k^r elements. Time: $O(k)$.

So the overall time complexity of $\mathbf{mult}_{\preceq k}$ is $O(rk^r \log k)$.

We also have to extend $\min_{\preceq}$ to $\mathbf{merge}_{\preceq k}$, which takes two vectors of length k (or fewer) as input and outputs the top k (in sorted order) of the $2k$ elements. This is similar to merge-sort (Cormen *et al.*, 2001) and can be done in linear time $O(k)$. Then, we only need to rewrite line 4 of the Viterbi algorithm (Figure 3) to extend it to the k-best case:

$$4: \quad \hat{\mathbf{D}}(v) \leftarrow \mathbf{merge}_{\preceq k}(\hat{\mathbf{D}}(v), \mathbf{mult}_{\preceq k}(e))$$

and the time complexity for this line is $O(|e|k^{|e|} \log k)$, making the overall complexity $O(|E|k^a \log k)$ if we consider the arity a of the hypergraph to be constant.[2] The overall space complexity is $O(|V|k)$ since for each vertex we need to store a vector of length k.

In the context of CKY parsing for CFG, the 1-best Viterbi algorithm has complexity $O(n^3|P|)$ while the k-best version is $O(n^3|P|k^2 \log k)$, which is slower by a factor of $O(k^2 \log k)$.

4.2 Algorithm 1: speed up $\mathbf{mult}_{\preceq k}$

First we seek to exploit the fact that input vectors are all sorted and the function f is monotonic; moreover, we are only interested in the top k elements of the $k^{|e|}$ possibilities.

Define $\mathbf{1}$ to be the vector whose elements are all 1; define $\mathbf{b}^i$ to be the vector whose elements are all 0 except $b^i_i = 1$.

As we compute $\mathbf{p}_e = \mathbf{mult}_{\preceq k}(e)$, we maintain a *candidate set* C of derivations that have the potential to be the next best derivation in the list. If we picture the input as an $|e|$-dimensional space, C contains those derivations that have not yet been included in $\mathbf{p}_e$, but are on the boundary with those which have. It is initialized to $\{\langle e, \mathbf{1} \rangle\}$. At each step, we extract the best derivation from C—call it $\langle e, \mathbf{j} \rangle$—and append it to $\mathbf{p}_e$. Then $\langle e, \mathbf{j} \rangle$ must be replaced in C by its neighbors,

$$\{\langle e, \mathbf{j} + \mathbf{b}^l \rangle \mid 1 \leq l \leq |e|\}$$

(see Figure 4.2 for an illustration). We implement C as a priority queue (Cormen *et al.*, 2001) to make the extraction of its best derivation efficient. At each iteration, there are one Extract-Min and $|e|$ Insert operations. If we use a binary-heap implementation for priority queues, we get $O(|e| \log k|e|)$ time complexity for each iteration.[3] Since we are only interested in the top k elements, there are k iterations and the time complexity for a single $\mathbf{mult}_{\preceq k}$ is $O(k|e| \log k|e|)$, yielding an overall time complexity of $O(|E|k \log k)$ and reducing the multiplicative overhead by a factor of $O(k^{a-1})$ (again, assuming a is constant). In the context of CKY parsing, this reduces the overhead to $O(k \log k)$. Figure 5 shows the additional pseudocode needed for this algorithm. It is integrated into the Viterbi algorithm (Figure 3) simply by rewriting line 4 of to invoke the function $\textsc{Mult}(e, k)$:

$$4: \quad \hat{\mathbf{D}}(v) \leftarrow \mathbf{merge}_{\preceq k}(\hat{\mathbf{D}}(v), \textsc{Mult}(e, k))$$

4.3 Algorithm 2: combine $\mathbf{merge}_{\preceq k}$ into $\mathbf{mult}_{\preceq k}$

We can further speed up both $\mathbf{merge}_{\preceq k}$ and $\mathbf{mult}_{\preceq k}$ by a similar idea. Instead of letting each $\mathbf{mult}_{\preceq k}$ generate a full k derivations for each hyperarc e and only then applying $\mathbf{merge}_{\preceq k}$ to the results, we can combine the candidate sets for all the hyperarcs into a single candidate set. That is, we initialize C to $\{\langle e, \mathbf{1} \rangle \mid e \in BS(v)\}$, the set of all the top parses from each incoming hyperarc (cf. Algorithm 1). Indeed, it suffices to keep only the top k out of the $|BS(v)|$ candidates in C, which would lead to a significant speedup in the case where $|BS(v)| \gg k$. [4] Now the top derivation in C is the top derivation for v. Then, whenever we remove an element $\langle e, \mathbf{j} \rangle$ from C, we replace it with the $|e|$ elements $\{\langle e, \mathbf{j} + \mathbf{b}^l \rangle \mid 1 \leq l \leq |e|\}$ (again, as in Algorithm 1). The full pseudocode for this algorithm is shown in Figure 6.

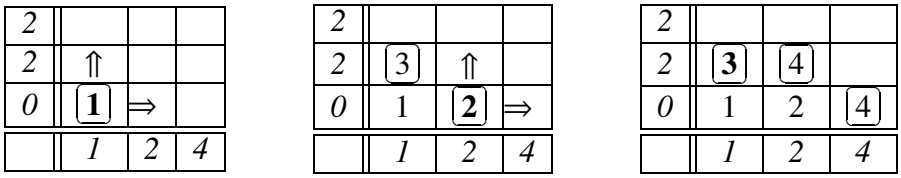
4.4 Algorithm 3: compute $\mathbf{mult}_{\preceq k}$ lazily

Algorithm 2 exploited the idea of lazy computation: performing $\mathbf{mult}_{\preceq k}$ only as many times as necessary. But this algorithm still calculates a full k-best list for every vertex in the hypergraph, whereas we are only interested in

[2] Actually, we do not need to sort all $k^{|e|}$ elements in order to extract the top k among them; there is an efficient algorithm (Cormen *et al.*, 2001) that can select the kth best element from the $k^{|e|}$ elements in time $O(k^{|e|})$. So we can improve the overhead to $O(k^a)$.

[3] If we maintain a Min-Heap along with the Min-Heap, we can reduce the per-iteration cost to $O(|e| \log k)$, and with Fibonacci heap we can further improve it to be $O(|e| + \log k)$. But these techniques do not change the overall complexity when a is constant, as we will see.

[4] This can be implemented by a linear-time randomized-selection algorithm (a.k.a. quick-select) (Cormen *et al.*, 2001).



(a) (b) (c)

Figure 4: An illustration of Algorithm 1 in $|e| = 2$ dimensions. Here $k = 3$, $\preceq$ is the numerical $\leq$, and the monotonic function f is defined as $f(a, b) = a + b$. Italic numbers on the x and y axes are a_i's and b_j's, respectively. We want to compute the top 3 results from $f(a_i, b_j)$ with $1 \leq i, j \leq 3$. In each iteration the current *frontier* is shown in oval boxes, with the bold-face denoting the best element among them. That element will be extracted and replaced by its two neighbors (⇑ and ⇒) in the next iteration.

```
1: function Mult(e, k)
2:     cand ← {⟨e, 1⟩}                          ▷ initialize the heap
3:     p ← empty list                           ▷ the result of mult_{≤k}
4:     while |p| < k and |cand| > 0 do
5:         AppendNext(cand, p, k)
6:     return p
7:
8: procedure AppendNext(cand, p)
9:     ⟨e, j⟩ ← Extract-Min(cand)
10:    append ⟨e, j⟩ to p
11:    for i ← 1 . . . |e| do                   ▷ add the |e| neighbors
12:        j′ ← j + b^i
13:        if j′_i ≤ |D̂(T_i(e))| and ⟨e, j′⟩ ∉ cand then
14:            Insert(cand, ⟨e, j′⟩)            ▷ add to heap
```

Figure 5: Part of Algorithm 1.

```
1: procedure FindAllKBest(k)
2:     for v ∈ V in topological order do
3:         FindKBest(v, k)
4:
5: procedure FindKBest(v, k)
6:     GetCandidates(v, k)                      ▷ initialize the heap
7:     while |D̂(v)| < k and |cand[v]| > 0 do
8:         AppendNext(cand[v], D̂(v))
9:
10: procedure GetCandidates(v, k)
11:    temp ← {⟨e, 1⟩ | e ∈ BS(v)}
12:    cand[v] ← the top k elements in temp     ▷ prune
           away useless candidates
13:    Heapify(cand[v])
```

Figure 6: Algorithm 2

```
1: procedure LazyKthBest(v, k, k′)                          ▷ k′ is the global k
2:     if |D̂(v)| ≥ k then                                  ▷ k^th derivation already computed?
3:         return
4:     if cand[v] is not defined then                       ▷ first visit of vertex v?
5:         GetCandidates(v, k′)                             ▷ initialize the heap
6:         append Extract-Min(cand[v]) to D̂(v)              ▷ 1-best
7:     while |D̂(v)| < k and |cand[v]| > 0 do
8:         ⟨e, j⟩ ← D̂_{|D̂(v)|}(v)                          ▷ last derivation
9:         LazyNext(cand[v], e, j, k′)                      ▷ update the heap, adding the successors of last derivation
10:        append Extract-Min(cand[v]) to D̂(v)              ▷ get the next best derivation and delete it from the heap
11:
12: procedure LazyNext(cand, e, j, k′)
13:    for i ← 1 . . . |e| do                               ▷ add the |e| neighbors
14:        j′ ← j + b^i
15:        LazyKthBest(T_i(e), j′_i, k′)                    ▷ recursively solve a sub-problem
16:        if j′_i ≤ |D̂(T_i(e))| and ⟨e, j′⟩ ∉ cand then    ▷ if it exists and is not in heap yet
17:            Insert(cand, ⟨e, j′⟩)                        ▷ add to heap
```

Figure 7: Algorithm 3

Algorithm	Time Complexity
1-best Viterbi	$O(E)$
Algorithm 0	$O(Ek^a \log k)$
Algorithm 1	$O(Ek \log k)$
Algorithm 2	$O(E + Vk \log k)$
Algorithm 3	$O(E + \lvert D_{max}\rvert k \log k)$
generalized J&M	$O(E + \lvert D_{max}\rvert k \log(d + k))$

Table 1: Summary of Algorithms.

the k-best derivations of the target vertex (goal item). We can therefore take laziness to an extreme by delaying the whole k-best calculation until after parsing. Algorithm 3 assumes an initial parsing phase that generates the hypergraph and finds the 1-best derivation of each item; then in the second phase, it proceeds as in Algorithm 2, but starts at the goal item and calls itself recursively only as necessary. The pseudocode for this algorithm is shown in Figure 7. As a side note, this second phase should be applicable also to a cyclic hypergraph as long as its derivation weights are bounded.

Algorithm 2 has an overall complexity of $O(|E| + |V|k \log k)$ and Algorithm 3 is $O(|E| + |D_{\max}|k \log k)$ where $|D_{\max}|$ is the size of the longest among all top k derivations (for CFG in CNF, $|D| = 2n - 1$ for all D, so $|D_{\max}|$ is $O(n)$). These are significant improvements against Algorithms 0 and 1 since it turns the multiplicative overhead into an additive overhead. In practice, $|E|$ usually dominates, as in CKY parsing of CFG. So theoretically the running times grow very slowly as k increases, which is exactly demonstrated by our experiments below.

4.5 Summary and Discussion of Algorithms

The four algorithms, along with the 1-best Viterbi algorithm and the generalized Jiménez and Marzal algorithm, are compared in Table 1.

The key difference between our Algorithm 3 and Jiménez and Marzal's algorithm is the restriction of top k candidates before making heaps (line 11 in Figure 6, see also Sec. 4.3). Without this line Algorithm 3 could be considered as a generalization of the Jiménez and Marzal algorithm to the case of acyclic monotonic hypergraphs. This line is also responsible for improving the time complexity from $O(|E| + |D_{\max}|k \log(d + k))$ (generalized Jiménez and Marzal algorithm) to $O(|E| + |D_{\max}|k \log k)$, where $d = \max_v |BS(v)|$ is the maximum in-degree among all vertices. So in case $k < d$, our algorithm outperforms Jiménez and Marzal's.

5 Experiments

We report results from two sets of experiments. For probabilistic parsing, we implemented Algorithms 0, 1, and 3 on top of a widely-used parser (Bikel, 2004) and conducted experiments on parsing efficiency and the quality of the k-best-lists. We also implemented Algorithms 2 and 3 in a parsing-based MT decoder (Chiang, 2005) and report results on decoding speed.

5.1 Experiment 1: Bikel Parser

Bikel's parser (2004) is a state-of-the-art multilingual parser based on lexicalized context-free models (Collins, 2003; Eisner, 2000). It does support k-best parsing, but, following Collins' parse-reranking work (Collins, 2000) (see also Section 5.1.2), it accomplishes this by simply abandoning dynamic programming, i.e., no items are considered equivalent (Charniak and Johnson, 2005). Theoretically, the time complexity is *exponential* in n (the input sentence length) and constant in k, since, without merging of equivalent items, there is no limit on the number of items in the chart. In practice, beam search is used to reduce the observed time.[5] But with the standard beam width of 10^{-4}, this method becomes prohibitively expensive for $n \geq 25$ on Bikel's parser. Collins (2000) used a narrower 10^{-3} beam and further applied a cell limit of 100,[6] but, as we will show below, this has a detrimental effect on the quality of the output. We therefore omit this method from our speed comparisons, and use our implementation of Algorithm 0 (naïve) as the baseline.

We implemented our k-best Algorithms 0, 1, and 3 on top of Bikel's parser and conducted experiments on a 2.4 GHz 64-bit AMD Opteron with 32 GB memory. The program is written in Java 1.5 running on the Sun JVM in server mode with a maximum heap size of 5 GB. For this experiment, we used sections 02–21 of the Penn Treebank (PTB) (Marcus *et al.*, 1993) as the training data and section 23 (2416 sentences) for evaluation, as is now standard. We ran Bikel's parser using its settings to emulate Model 2 of (Collins, 2003).

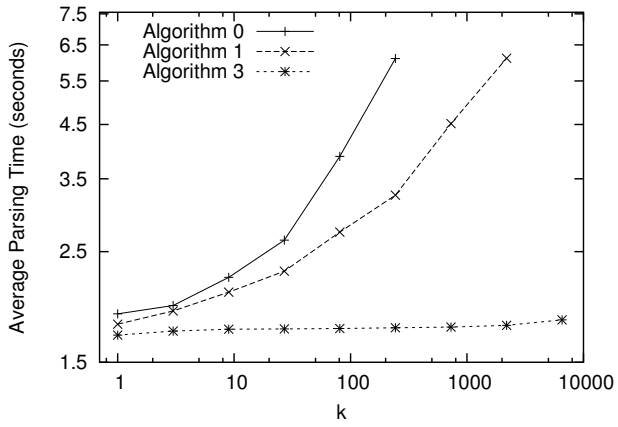
5.1.1 Efficiency

We tested our algorithms under various conditions. We first did a comparison of the average parsing time per sentence of Algorithms 0, 1, and 3 on section 23, with $k \leq 10000$ for the standard beam of width 10^{-4}. Figure 8(a) shows that the parsing speed of Algorithm 3 improved dramatically against the other algorithms and is nearly constant in k, which exactly matches the complexity analysis. Algorithm 1 ($k \log k$) also significantly outperforms the baseline naïve algorithm ($k^2 \log k$).

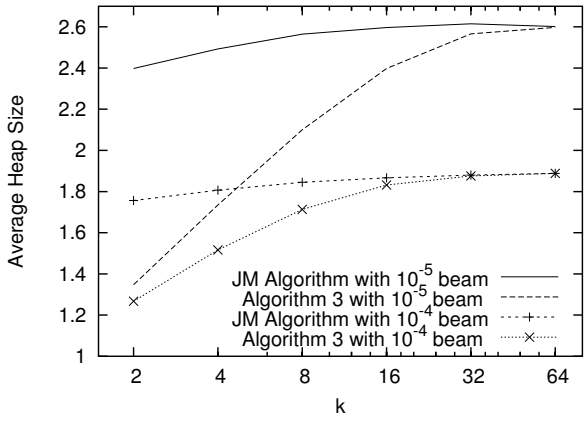
We also did a comparison between our Algorithm 3 and the Jiménez and Marzal algorithm in terms of average

[5] In beam search, or *threshold pruning*, each cell in the chart (typically containing all the items corresponding to a span $[i, j]$) is reduced by discarding all items that are worse than β times the score of the best item in the cell. This β is known as the *beam width*.

[6] In this type of pruning, also known as *histogram pruning*, only the α best items are kept in each cell. This α is called the *cell limit*.



(a) Average parsing speed (Algs. 0 vs. 1 vs. 3, log-log)



(b) Average heap size (Alg. 3 vs. Jiménez and Marzal)

Figure 8: Efficiency results of the k-best Algorithms, compared to Jiménez and Marzal's algorithm

heap size. Figure 8(b) shows that for larger k, the two algorithms have the same average heap size, but for smaller k, our Algorithm 3 has a considerably smaller average heap size. This difference is useful in applications where only short k-best lists are needed. For example, McDonald *et al.* (2005) find that $k = 5$ gives optimal parsing accuracy.

5.1.2 Accuracy

Our efficient k-best algorithms enable us to search over a larger portion of the whole search space (e.g. by less aggressive pruning), thus producing k-best lists with better quality than previous methods. We demonstrate this by comparing our k-best lists to those in (Ratnaparkhi, 1997), (Collins, 2000) and the parallel work by Charniak and Johnson (2005) in several ways, including oracle reranking and average number of found parses.

Ratnaparkhi (1997) introduced the idea of oracle reranking: suppose there exists a perfect reranking scheme that magically picks the best parse that has the highest F-score among the top k parses for each sentence. Then the performance of this oracle reranking scheme is the upper bound of any actual reranking system like (Collins, 2000).As k increases, the F-score is nondecreasing, and there is some k (which might be very large) at which the F-score converges.

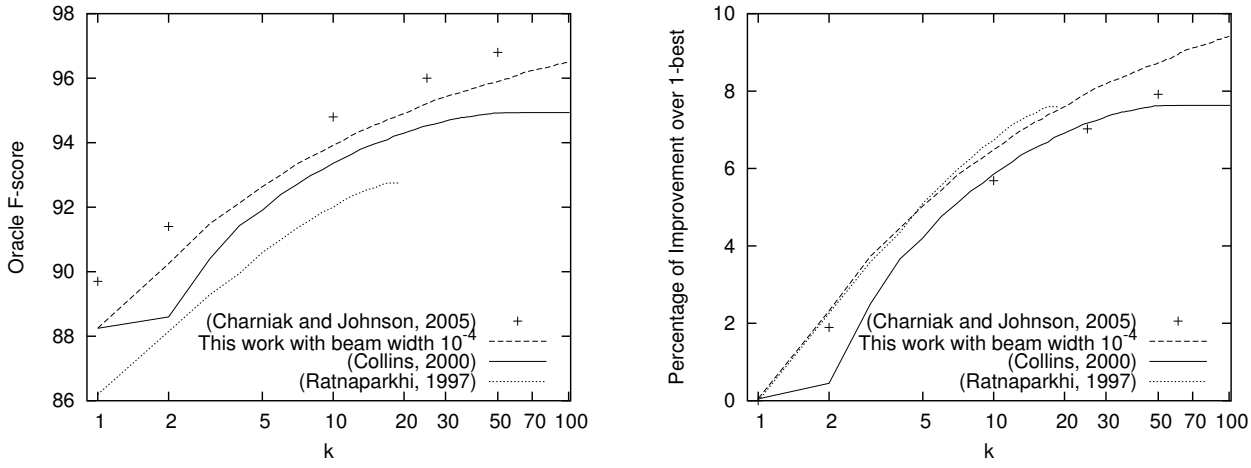
Ratnaparkhi reports experiments using oracle reranking with his statistical parser MXPARSE, which can compute its k-best parses (in his experiments, $k = 20$). Collins (2000), in his parse-reranking experiments, used his Model 2 parser (Collins, 2003) with a beam width of 10^{-3} together with a cell limit of 100 to obtain k-best lists; the average number of parses obtained per sentence was 29.2, the maximum, 101.[7] Charniak and Johnson (2005) use coarse-to-fine parsing on top of the Charniak (2000) parser and get 50-best lists for section 23.

Figure 9(a) compares the results of oracle reranking. Collins' curve converges at around $k = 50$ while ours continues to increase. With a beam width of 10^{-4} and $k = 100$, our parser plus oracle reaches an F-score of 96.4%, compared to Collins' 94.9%. Charniak and Johnson's work, however, is based on a completely different parser whose 1-best F-score is 1.5 points higher than the 1-bests of ours and Collins', making it difficult to compare in absolute numbers. So we instead compared the *relative* improvement over 1-best. Figure 9(b) shows that our work has the largest percentage of improvement in terms of F-score when $k > 20$.

To further explore the impact of Collins' cell limit on the quality of k-best lists, we plotted average number of parses for a given sentence length (Figure 10). Generally speaking, as input sentences get longer, the number of parses grows (exponentially). But we see that the curve for Collins' k-best list goes down for large k (> 40). We suspect this is due to the cell limit of 100 pruning away potentially good parses too early in the chart. As sentences get longer, it is more likely that a lower-probability parse might contribute eventually to the k-best parses. So we infer that Collins' k-best lists have limited quality for large k, and this is demonstrated by the early convergence of its oracle-reranking score. By comparison, our curves of both beam widths continue to grow with $k = 100$.

All these experiments suggest that our k-best parses are of better quality than those from previous k-best parsers,

[7] The reason the maximum is 101 and not 100 is that Collins merged the 100-best list using a beam of 10^{-3} with the 1-best list using a beam of 10^{-4} (Collins, p.c.).



(a) Oracle Reranking

(b) Relative Improvement

Figure 9: Absolutive and Relative F-scores of oracle reranking for the top k ($\leq$ 100) parses for section 23, compared to (Charniak and Johnson, 2005), (Collins, 2000) and (Ratnaparkhi, 1997).

Figure 10: Average number of parses for each sentence length in section 23, using k=100, with beam width 10^{-4} and 10^{-3}, compared to (Collins, 2000).

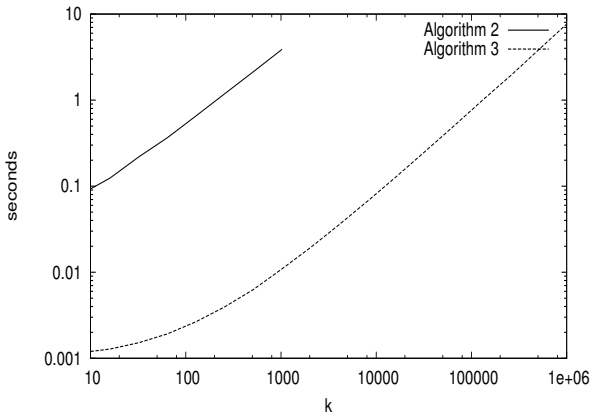


Figure 11: Algorithm 2 compared with Algorithm 3 (offline) on MT decoding task. Average time (both excluding initial 1-best phase) vs. k (log-log).

and similar quality to those from (Charniak and Johnson, 2005) which has so far the highest F-score after reranking, and this might lead to better results in real parse reranking.

5.2 Experiment 2: MT decoder

Our second experiment was on a CKY-based decoder for a machine translation system (Chiang, 2005), implemented in Python 2.4 accelerated with Psyco 1.3 (Rigo, 2004). We implemented Algorithms 2 and 3 to compute k-best English translations of Mandarin sentences. Because the CFG used in this system is large to begin with (millions of rules), and then effectively intersected with a finite-state machine on the English side (the language model), the grammar constant for this system is quite large. The decoder uses a relatively narrow beam search for efficiency.

We ran the decoder on a 2.8 GHz Xeon with 4 GB of memory, on 331 sentences from the 2002 NIST MTEval test set. We tested Algorithm 2 for $k = 2^i, 3 \leq i \leq 10$, and Algorithm 3 (offline algorithm) for $k = 2^i, 3 \leq i \leq 20$. For each sentence, we measured the time to calculate the k-best list, *not* including the initial 1-best parsing phase. We then averaged the times over our test set to produce the graph of Figure 11, which shows that Algorithm 3 runs an average of about 300 times faster than Algorithm 2. Furthermore, we were able to test Algorithm 3 up to $k = 10^6$ in a reasonable amount of time.[8]

[8]The curvature in the plot for Algorithm 3 for $k < 1000$ may be due to lack of resolution in the timing function for short times.

6 Conclusion

The problem of k-best parsing and the effect of k-best list size and quality on applications are subjects of increasing interest for NLP research. We have presented here a general-purpose algorithm for k-best parsing and applied it to two state-of-the-art, large-scale NLP systems: Bikel's implementation of Collins' lexicalized PCFG model (Bikel, 2004; Collins, 2003) and Chiang's synchronous CFG based decoder (Chiang, 2005) for machine translation. We hope that this work will encourage further investigation into whether larger and better k-best lists will improve performance in NLP applications, questions which we ourselves intend to pursue as well.

Acknowledgements

We would like to thank one of the anonymous reviewers of a previous version of this paper for pointing out the work by Jiménez and Marzal, and Eugene Charniak and Mark Johnson for providing an early draft of their paper and very useful comments. We are also extremely grateful to Dan Bikel for the help in experiments, and Michael Collins for providing the data in his paper. Our thanks also go to Dan Gildea, Jonathan Graehl, Julia Hockenmaier, Aravind Joshi, Kevin Knight, Daniel Marcu, Mitch Marcus, Ryan McDonald, Fernando Pereira, Giorgio Satta, Libin Shen, and Hao Zhang.

References

Bikel, D. M. (2004). Intricacies of Collins' parsing model. *Computational Linguistics*, **30**, 479–511.

Bod, R. (2000). Parsing with the shortest derivation. In *Proc. Eighteenth International Conference on Computational Linguistics (COLING)*, pages 69–75.

Brander, A. and Sinclair, M. (1995). A comparative study of k-shortest path algorithms. In *Proc. 11th UK Performance Engineering Workshop for Computer and Telecommunications Systems*.

Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Lai, J. C., and Mercer, R. L. (1995). Method and system for natural language translation. U. S. Patent 5,477,451.

Charniak, E. (2000). A maximum-entropy-inspired parser. In *Proc. First Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 132–139.

Charniak, E. and Johnson, M. (2005). Coarse-to-fine-grained n-best parsing and discriminative reranking. In *Proc. ACL 2005*.

Chiang, D. (2005). A hierarchical phrase-based model for statistical machine translation. In *Proc. ACL 2005*.

Collins, M. (2000). Discriminative reranking for natural language parsing. In *Proc. Seventeenth International Conference on Machine Learning (ICML)*, pages 175–182. Morgan Kaufmann.

Collins, M. (2003). Head-driven statistical models for natural language parsing. *Computational Linguistics*, **29**, 589–637.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, second edition.

Eisner, J. (2000). Bilexical grammars and their cubic-time parsing algorithms. In H. Bunt and A. Nijholt, editors, *Advances in Probabilistic and Other Parsing Technologies*, pages 29–62. Kluwer Academic Publishers.

Eppstein, D. (2001). Bibliography on *k* shortest paths and other "*k* best solutions" problems. http://www.ics.uci.edu/~eppstein/bibs/kpath.bib.

Gallo, G., Longo, G., and Pallottino, S. (1993). Directed hypergraphs and applications. *Discrete Applied Mathematics*, **42**(2), 177–201.

Gildea, D. and Jurafsky, D. (2002). Automatic labeling of semantic roles. *Computational Linguistics*, **28**(3), 245–288.

Goodman, J. (1998). *Parsing Inside-Out*. Ph.D. thesis, Harvard University.

Goodman, J. (1999). Semiring parsing. *Computational Linguistics*, **25**, 573–605.

Jiménez, V. M. and Marzal, A. (2000). Computation of the *n* best parse trees for weighted and stochastic context-free grammars. In *Proc. of the Joint IAPR International Workshops on Advances in Pattern Recognition*.

Joshi, A. K. and Vijay-Shanker, K. (1999). Compositional semantics with lexicalized tree-adjoining grammar (LTAG): How much underspecification is necessary? In H. C. Bunt and E. G. C. Thijsse, editors, *Proc. IWCS-3*, pages 131–145.

Klein, D. and Manning, C. D. (2001). Parsing and hypergraphs. In *Proceedings of the Seventh International Workshop on Parsing Technologies (IWPT-2001), 17-19 October 2001, Beijing, China*. Tsinghua University Press.

Knight, K. and Graehl, J. (2005). An overview of probabilistic tree transducers for natural language processing. In *Proc. of the Sixth International Conference on Intelligent Text Processing and Computational Linguistics (CICLing)*, LNCS.

Knuth, D. (1977). A generalization of Dijkstra's algorithm. *Information Processing Letters*, **6**(1).

Kumar, S. and Byrne, W. (2004). Minimum bayes-risk decoding for statistical machine translation. In *HLT-NAACL*.

Lawler, E. L. (1977). Comment on computing the *k* shortest paths in a graph. *Comm. of the ACM*, **20**(8), 603–604.

Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, **19**, 313–330.

McAllester, D., Collins, M., and Pereira, F. (2004). Case-factor diagrams for structured probabilistic modeling. In *Proc. UAI 2004*.

McDonald, R., Crammer, K., and Pereira, F. (2005). Online large-margin training of dependency parsers. In *Proc. ACL 2005*.

Minieka, E. (1974). On computing sets of shortest paths in a graph. *Comm. of the ACM*, **17**(6), 351–353.

Mohri, M. (2002). Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, **7**(3), 321–350.

Mohri, M. and Riley, M. (2002). An efficient algorithm for the n-best-strings problem. In *Proceedings of the International Conference on Spoken Language Processing 2002 (ICSLP '02)*, Denver, Colorado.

Nederhof, M.-J. (2003). Weighted deductive parsing and Knuth's algorithm. *Computational Linguistics*, pages 135–143.

Nielsen, L. R., Andersen, K. A., and Pretolani, D. (2005). Finding the k shortest hyperpaths. *Computers and Operations Research*.

Och, F. J. (2003). Minimum error rate training in statistical machine translation. In *Proc. ACL 2003*, pages 160–167.

Och, F. J. and Ney, H. (2004). The alignment template approach to statistical machine translation. *Computational Linguistics*, **30**, 417–449.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Ratnaparkhi, A. (1997). A linear observed time statistical parser based on maximum entropy models. In *Proc. EMNLP 1997*, pages 1–10.

Rigo, A. (2004). Representation-based just-in-time specialization and the Psyco prototype for Python. In N. Heintze and P. Sestoft, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 15–26.

Shen, L., Sarkar, A., and Och, F. J. (2004). Discriminative reranking for machine translation. In *Proc. HLT-NAACL 2004*.

Shieber, S., Schabes, Y., and Pereira, F. (1995). Principles and implementation of deductive parsing. *Journal of Logic Programming*, **24**, 3–36.

Sutton, C. and McCallum, A. (2005). Joint parsing and semantic role labeling. In *Proc. CoNLL 2005*.

Taskar, B., Klein, D., Collins, M., Koller, D., and Manning, C. (2004). Max-margin parsing. In *Proc. EMNLP 2004*.

Wellner, B., McCallum, A., Peng, F., and Hay, M. (2004). An integrated, conditional model of information extraction and coreference with application to citation matching. In *Proc. UAI 2004*.