Exploring the Potential of Intractable Parsers

Mark Hopkins
Dept. of Computational Linguistics
Saarland University
Saarbrücken, Germany
mhopkins@coli.uni-sb.de

Jonas Kuhn
Dept. of Computational Linguistics
Saarland University
Saarbrücken, Germany
jonask@coli.uni-sb.de

Abstract

We revisit the idea of history-based parsing, and present a history-based parsing framework that strives to be simple, general, and flexible. We also provide a decoder for this probability model that is linear-space, optimal, and anytime. A parser based on this framework, when evaluated on Section 23 of the Penn Treebank, compares favorably with other state-of-the-art approaches, in terms of both accuracy and speed.

1 Introduction

Much of the current research into probabilistic parsing is founded on probabilistic context-free grammars (PCFGs) (Collins, 1996; Charniak, 1997; Collins, 1999; Charniak, 2000; Charniak, 2001; Klein and Manning, 2003). For instance, consider the parse tree in Figure 1. One way to decompose this parse tree is to view it as a sequence of applications of CFG rules. For this particular tree, we could view it as the application of rule "NP → NP PP," followed by rule "NP → DT NN," followed by rule "DT → that," and so forth. Hence instead of analyzing $P(tree)$, we deal with the more modular:

P(NP → NP PP, NP → DT NN, DT → that, NN → money, PP → IN NP, IN → in, NP → DT NN, DT → the, NN → market)

Obviously this joint distribution is just as difficult to assess and compute with as $P(tree)$. However there exist cubic-time dynamic programming algorithms to find the most likely parse if we assume that all CFG rule applications are marginally independent of one another. The problem, of course, with this simplification is that although it is computationally attractive, it is usually too strong of an independence assumption. To mitigate this loss of context, without sacrificing algorithmic tractability, typically researchers annotate the nodes of the parse tree with contextual information. A simple example is the annotation of nodes with their parent labels (Johnson, 1998).

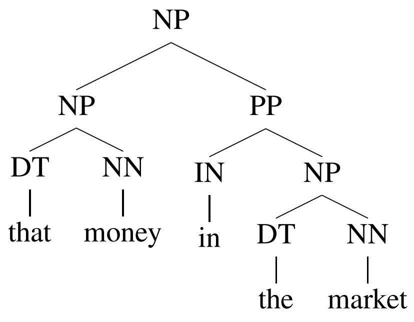

Figure 1: Example parse tree.

The choice of which annotations to use is one of the main features that distinguish parsers based on this approach. Generally, this approach has proven quite effective in producing English phrase-structure grammar parsers that perform well on the Penn Treebank.

One drawback of this approach is its inflexibility. Because we are adding probabilistic context by changing the data itself, we make our data increasingly sparse as we add features. Thus we are constrained from adding too many features, because at some point we will not have enough data to sustain them. We must strike a delicate balance between how much context we want to include versus how much we dare to partition our data set.

The major alternative to PCFG-based approaches are so-called *history-based* parsers (Black et al., 1993). These parsers differ from PCFG parsers in that they incorporate context by using a more complex probability model, rather than by modifying the data itself. The tradeoff to using a more powerful probabilistic model is that one can no longer employ dynamic programming to find the most probable parse. Thus one trades assurances of polynomial running time for greater modeling flexibility.

There are two canonical parsers that fall into this category: the decision-tree parser of (Magerman, 1995), and the maximum-entropy parser of (Ratnaparkhi, 1997). Both showed decent results on parsing the Penn Treebank, but in the decade since these papers were published, history-based parsers have been largely ignored by the research community in favor of PCFG-based approaches. There are several reasons why this may be. First is naturally the matter of time efficiency. Magerman reports decent parsing times, but for the purposes of efficiency, must restrict his results to sentences of length 40 or less. Furthermore, his two-phase stack decoder is a bit complicated and is acknowledged to require too much memory to handle certain sentences. Ratnaparkhi is vague about the running time performance of his parser, stating that it is "observed linear-time," but in any event, provides only a heuristic, not a complete algorithm.

Next is the matter of flexibility. The main advantage of abandoning PCFGs is the opportunity to have a more flexible and adaptable probabilistic parsing model. Unfortunately, both Magerman and Ratnaparkhi's models are rather specific and complicated. Ratnaparkhi's, for instance, consists of the interleaved sequence of four different types of tree construction operations. Furthermore, both are inextricably tied to the learning procedure that they employ (decision trees for Magerman, maximum entropy for Ratnaparkhi).

In this work, our goal is to revisit history-based parsers, and provide a general-purpose framework that is (a) simple, (b) fast, (c) space-efficient and (d) easily adaptable to new domains. As a method of evaluation, we use this framework with a very simple set of features to see how well it performs (both in terms of accuracy and running time) on the Penn Treebank. The overarching goal is to develop a history-based hierarchical labeling framework that is viable not only for parsing, but for other application areas that current rely on dynamic programming, like phrase-based machine translation.

2 Preliminaries

For the following discussion, it will be useful to establish some terminology and notational conventions. Typically we will represent variables with capital letters (e.g. X, Y) and sets of variables with bold-faced capital letters (e.g. $\mathbf{X}$, $\mathbf{Y}$). The domain of a variable X will be denoted $dom(X)$, and typically we will use the lower-case correspondent (in this case, x) to denote a value in the domain of X. A *partial assignment* (or simply *assignment*) of a set $\mathbf{X}$ of variables is a function $\mathbf{w}$ that maps a subset $\mathbf{W}$ of the variables of $\mathbf{X}$ to values in their respective domains. We define $dom(\mathbf{w}) = \mathbf{W}$. When $\mathbf{W} = \mathbf{X}$, then we say that $\mathbf{w}$ is a *full assignment* of $\mathbf{X}$. The *trivial assignment* of $\mathbf{X}$ makes no variable assignments.

Let $\mathbf{w}(X)$ denote the value that partial assignment $\mathbf{w}$ assigns to variable X. For value $x \in dom(X)$, let $\mathbf{w}[X = x]$ denote the assignment identical to $\mathbf{w}$ except that $\mathbf{w}[X = x](X) = x$. For a set $\mathbf{Y}$ of variables, let $\mathbf{w}|_{\mathbf{Y}}$ denote the restriction of partial assignment $\mathbf{w}$ to the variables in $dom(\mathbf{w}) \cap \mathbf{Y}$.

3 The Generative Model

The goal of this section is to develop a probabilistic process that generates labeled trees in a manner considerably different from PCFGs. We will use the tree in Figure 2 to motivate our model. In this example, nodes of the tree are labeled with either an A or a B. We can represent this tree using two charts. One chart labels each span with a boolean value, such that a span is labeled $true$ iff it is a constituent in the tree. The other chart labels each span with a label from our labeling scheme (A or B) or with the value $null$ (to represent that the span is unlabeled). We show these charts in Figure 3. Notice that we may want to have more than one labeling scheme. For instance, in the parse tree of Figure 1, there are three different types of labels: word labels, preterminal labels, and nonterminal labels. Thus we would use four 5x5 charts instead of two 3x3 charts to represent that tree.

We will pause here and generalize these concepts. Define a *labeling scheme* as a set of symbols including a special symbol $null$ (this will desig-

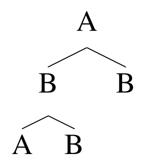

Figure 2: Example labeled tree.

	1	2	3
1	true	true	true
2	-	true	false
3	-	-	true

	1	2	3
1	A	B	A
2	-	B	null
3	-	-	B

Figure 3: Chart representation of the example tree: the left chart tells us which spans are tree constituents, and the right chart tells us the labels of the spans ($null$ means unlabeled).

nate that a given span is unlabeled). For instance, we can define $L^1 = \{null, A, B\}$ to be a labeling scheme for the example tree.

Let $\mathcal{L} = \{L^1, L^2, ...L^m\}$ be a set of labeling schemes. Define a *model variable* of $\mathcal{L}$ as a symbol of the form S_{ij} or L^k_{ij}, for positive integers i, j, k, such that $i \leq j$ and $k \leq m$. Model variables of the form S_{ij} indicate whether span (i, j) is a tree constituent, hence the *domain* of S_{ij} is $\{true, false\}$. Such variables correspond to entries in the left chart of Figure 3. Model variables of the form L^k_{ij} indicate which label from scheme L^k is assigned to span (i, j), hence the *domain* of model variable L^k_{ij} is L^k. Such variables correspond to entries in the right chart of Figure 3. Here we have only one labeling scheme.

Let $\mathbf{V}_{\mathcal{L}}$ be the (countably infinite) set of model variables of $\mathcal{L}$. Usually we are interested in trees over a given sentence of finite length n. Let $\mathbf{V}^n_{\mathcal{L}}$ denote the finite subset of $\mathbf{V}_{\mathcal{L}}$ that includes precisely the model variables of the form S_{ij} or L^k_{ij}, where $j \leq n$.

Basically then, our model consists of two types of decisions: (1) whether a span should be labeled, and (2) if so, what label(s) the span should have. Let us proceed with our example. To generate the tree of Figure 2, the first decision we need to make is how many leaves it will have (or equivalently, how large our tables will be). We assume that we have a probability distribution P_N over the set of positive integers. For our example tree, we draw the value 3, with probability $P_N(3)$.

Now that we know our tree will have three leaves, we can now decide which spans will be constituents and what labels they will have. In other words, we assign values to the variables in $\mathbf{V}^3_{\mathcal{L}}$. First we need to choose the order in which we will make these assignments. For our example, we will assign model variables in the following order: S_{11}, L^1_{11}, S_{22}, L^1_{22}, S_{33}, L^1_{33}, S_{12}, L^1_{12}, S_{23}, L^1_{23}, S_{13}, L^1_{13}. A detailed look at this assignment process should help clarify the details of the model.

Assigning S_{11}: The first model variable in our order is S_{11}. In other words, we need to decide whether the span $(1, 1)$ should be a constituent. We could let this decision be probabilistically determined, but recall that we are trying to generate a well-formed tree, thus the leaves and the root should always be considered constituents. To handle situations when we would like to make deterministic variable assignments, we supply an auxilliary function $\mathcal{A}$ that tells us (given a model variable X and the history of decisions made so far) whether X should be automatically determined, and if so, what value it should be assigned. In our running example, we ask $\mathcal{A}$ whether S_{11} should be automatically determined, given the previous assignments made (so far only the value chosen for n, which was 3). The so-called *auto-assignment function* $\mathcal{A}$ responds (since S_{11} is a leaf span) that S_{11} should be automatically assigned the value $true$, making span $(1, 1)$ a constituent.

Assigning L^1_{11}: Next we want to assign a label to the first leaf of our tree. There is no compelling reason to deterministically assign this label. Therefore, the auto-assignment function $\mathcal{A}$ declines to assign a value to L^1_{11}, and we proceed to assign its value probabilistically. For this task, we would like a probability distribution over the labels of labeling scheme $L^1 = \{null, A, B\}$, conditioned on the decision history so far. The difficulty is that it is clearly impractical to learn conditional distributions over every conceivable history of variable assignments. So first we distill the important features from an assignment history. For instance, one such feature (though possibly not a good one) could be whether an odd or an even number of nodes have so far been labeled with an A. Our conditional probability distribution is conditioned on the values of these features, instead of the entire assignment history. Consider specifically model variable L^1_{11}. We compute its features (an even number of nodes – zero – have so far been labeled with an A), and then we use these feature values to access the relevant probability distribution over $\{null, A, B\}$. Drawing from this conditional distribution, we probabilistically assign the value A to variable L^1_{11}.

Assigning S_{22}, L^1_{22}, S_{33}, L^1_{33}: We proceed in this way to assign values to S_{22}, L^1_{22}, S_{33}, L^1_{33} (the S-variables deterministically, and the L^1-variables probabilistically).

Assigning S_{12}: Next comes model variable S_{12}. Here, there is no reason to deterministically dictate whether span $(1, 2)$ is a constituent or not. Both should be considered options. Hence we treat this situation the same as for the L^1 variables. First we extract the relevant features from the assignment history. We then use these features to access the correct probability distribution over the domain of S_{12} (namely $\{true, false\}$). Drawing from this conditional distribution, we probabilistically assign the value $true$ to S_{12}, making span $(1, 2)$ a constituent in our tree.

Assigning L^1_{12}: We proceed to probabilistically assign the value B to L^1_{12}, in the same manner as we did with the other L^1 model variables.

Assigning S_{23}: Now we must determine whether span $(2, 3)$ is a constituent. We could again probabilistically assign a value to S_{23} as we did for S_{12}, but this could result in a hierarchical structure in which both spans $(1, 2)$ and $(2, 3)$ are constituents, which is not a tree. For trees, we cannot allow two model variables S_{ij} and S_{kl} to both be assigned $true$ if they *properly overlap*, i.e. their spans overlap and one is not a subspan of the other. Fortunately we have already established auto-assignment function $\mathcal{A}$, and so we simply need to ensure that it automatically assigns the value $false$ to model variable S_{kl} if a properly overlapping model variable S_{ij} has previously been assigned the value $true$.

Assigning L^1_{23}, S_{13}, L^1_{13}: In this manner, we can complete our variable assignments: L^1_{23} is automatically determined (since span $(2, 3)$ is not a constituent, it should not get a label), as is S_{13} (to ensure a rooted tree), while the label of the root is probabilistically assigned.

We can summarize this generative process as a general modeling tool. Define a *hierarchical labeling process* (HLP) as a 5-tuple $\langle \mathcal{L}, <, \mathcal{A}, \mathcal{F}, \mathcal{P}\rangle$ where:

- $\mathcal{L} = \{L^1, L^2, ..., L^m\}$ is a finite set of *labeling schemes*.
- $<$ is a *model order*, defined as a total ordering of the model variables $\mathbf{V}_{\mathcal{L}}$ such that for all i, j, k: $S_{ij} < L^k_{ij}$ (i.e. we decide whether a span is a constituent before attempting to label it).
- $\mathcal{A}$ is an *auto-assignment function*. Specifically $\mathcal{A}$ takes three arguments: a model variable Y of $\mathbf{V}_{\mathcal{L}}$, a partial assignment $\mathbf{x}$ of $\mathbf{V}_{\mathcal{L}}$, and integer n. The function $\mathcal{A}$ maps this 3-tuple to $false$ if the variable Y should not be automatically assigned a value based on the current history, or the pair $\langle true, y\rangle$, where y is the value in the domain of Y that should be automatically assigned to Y.
- $\mathcal{F} = \{\mathcal{F}^S, \mathcal{F}^1, \mathcal{F}^2, ..., \mathcal{F}^m\}$ is a set of *feature functions*. Specifically, $\mathcal{F}^k$ (resp., $\mathcal{F}^S$) takes four arguments: a partial assignment $\mathbf{x}$ of $\mathbf{V}_{\mathcal{L}}$, and integers i , j , n such that $1 \leq i \leq j \leq n$. It maps this 4-tuple to a full assignment $\mathbf{f}^k$ (resp., $\mathbf{f}^S$) of some finite set $\mathbf{F}^k$ (resp., $\mathbf{F}^S$) of feature variables.
- $\mathcal{P} = \{P_N, P_S, P_1, P_2, ..., P_m\}$ is a set of probability distributions. P_N is a marginal probability distribution over the set of positive integers, whereas $\{P_S, P_1, P_2, ..., P_m\}$ are conditional probability distributions. Specifically, P_k (respectively, P_S) is a function that takes as its argument a full assignment $\mathbf{f}^k$ (resp., $\mathbf{f}^S$) of feature set $\mathbf{F}^k$ (resp.,

```
HLPGEN(HLP $\mathcal{H} = \langle \mathcal{L}, <, \mathcal{A}, \mathcal{F}, \mathcal{P}\rangle$):
  1. Choose a positive integer $n$ from distribution $P_N$. Let $\mathbf{x}$ be the trivial assignment of $\mathbf{V}_{\mathcal{L}}$.
  2. In the order defined by $<$, compute step 3 for each model variable $Y$ of $\mathbf{V}^n_{\mathcal{L}}$.
  3. If $\mathcal{A}(Y, \mathbf{x}, n) = \langle true, y\rangle$ for some $y$ in the domain of model variable $Y$, then let $\mathbf{x} = \mathbf{x}[Y = y]$. Otherwise assign a value to $Y$ from its domain:
     (a) If $Y = S_{ij}$, then let $\mathbf{x} = \mathbf{x}[S_{ij} = s_{ij}]$, where $s_{ij}$ is a value drawn from distribution $P_S(s|\mathcal{F}^S(\mathbf{x}, i, j, n))$.
     (b) If $Y = L^k_{ij}$, then let $\mathbf{x} = \mathbf{x}[L^k_{ij} = l^k_{ij}]$, where $l^k_{ij}$ is a value drawn from distribution $P_k(l^k|\mathcal{F}^k(\mathbf{x}, i, j, n))$.
  4. Return $\langle n, \mathbf{x}\rangle$.
```

Figure 4: Pseudocode for the generative process.

$\mathcal{A}$(variable Y, assignment $\mathbf{x}$, int n):

1. If $Y = S_{ij}$, and there exists a properly overlapping model variable S_{kl} such that $\mathbf{x}(S_{kl}) = true$, then return $\langle true, false \rangle$.
2. If $Y = S_{ii}$ or $Y = S_{1n}$, then return $\langle true, true \rangle$.
3. If $Y = L^k_{ij}$, and $\mathbf{x}(S_{ij}) = false$, then return $\langle true, null \rangle$.
4. Else return $false$.

Figure 5: An example auto-assignment function.

$\mathbf{F}^S$). It maps this to a probability distribution over $dom(L^k)$ (resp., $\{true, false\}$).

An HLP probabilistically generates an assignment of its model variables using the generative process shown in Figure 4. Taking an HLP $\mathcal{H} = \langle \mathcal{L}, <, \mathcal{A}, \mathcal{F}, \mathcal{P} \rangle$ as input, HLPGen outputs an integer n, and an *$\mathcal{H}$-labeling* $\mathbf{x}$ of length n, defined as a full assignment of $\mathbf{V}^n_{\mathcal{L}}$.

Given the auto-assignment function in Figure 5, every $\mathcal{H}$-labeling generated by HLPGen can be viewed as a labeled tree using the interpretation: span (i,j) is a constituent iff $S_{ij} = true$; span (i,j) has label $l^k \in dom(L^k)$ iff $L^k_{ij} = l^k$.

4 Learning

The generative story from the previous section allows us to express the probability of a labeled tree as $P(n, \mathbf{x})$, where $\mathbf{x}$ is an $\mathcal{H}$-labeling of length n. For model variable X, define $\mathbf{V}^<_{\mathcal{L}}(X)$ as the subset of $\mathbf{V}_{\mathcal{L}}$ appearing before X in model order $<$. With the help of this terminology, we can decompose $P(n, \mathbf{x})$ into the following product:

$$P_0(n) \cdot \prod_{S_{ij} \in \mathbf{Y}} P_S(\mathbf{x}(S_{ij}) | \mathbf{f}^S_{ij}) \cdot \prod_{L^k_{ij} \in \mathbf{Y}} P_k(\mathbf{x}(L^k_{ij}) | \mathbf{f}^k_{ij})$$

where $\mathbf{f}^S_{ij} = \mathcal{F}^S(\mathbf{x}|_{\mathbf{V}^<_{\mathcal{L}}(S_{ij})}, i, j, n)$ and $\mathbf{f}^k_{ij} = \mathcal{F}^k(\mathbf{x}|_{\mathbf{V}^<_{\mathcal{L}}(L^k_{ij})}, i, j, n)$ and $\mathbf{Y}$ is the subset of $\mathbf{V}^n_{\mathcal{L}}$ that was not automatically assigned by HLPGen.

Usually in parsing, we are interested in computing the most likely tree given a specific sentence. In our framework, this generalizes to computing: $argmax_{\mathbf{x}} P(\mathbf{x}|n, \mathbf{w})$, where $\mathbf{w}$ is a subassignment of an $\mathcal{H}$-labeling $\mathbf{x}$ of length n. In natural language parsing, $\mathbf{w}$ could specify the constituency and word labels of the leaf-level spans. This would be equivalent to asking: given a sentence, what is its most likely parse?

Let $\mathbf{W} = dom(\mathbf{w})$ and suppose that we choose a model order $<$ such that for every pair of model variables $W \in \mathbf{W}, X \in V_{\mathcal{L}} \backslash \mathbf{W}$, either $W < X$ or W is always auto-assigned. Then $P(\mathbf{x}|n, \mathbf{w})$ can be expressed as:

$$\prod_{S_{ij} \in \mathbf{Y} \backslash \mathbf{W}} P_S(\mathbf{x}(S_{ij}) | \mathbf{f}^S_{ij}) \cdot \prod_{L^k_{ij} \in \mathbf{Y} \backslash \mathbf{W}} P_k(\mathbf{x}(L^k_{ij}) | \mathbf{f}^k_{ij})$$

Hence the distributions we need to learn are probability distributions $P_S(s_{ij}|\mathbf{f}_S)$ and $P_k(l^k_{ij}|\mathbf{f}_k)$. This is fairly straightforward. Given a data bank consisting of labeled trees (such as the Penn Treebank), we simply convert each tree into its $\mathcal{H}$-labeling and use the probabilistically determined variable assignments to compile our training instances. In this way, we compile $k+1$ sets of training instances that we can use to induce P_S, and the P_k distributions. The choice of which learning technique to use is up to the personal preference of the user. The only requirement is that it must return a conditional probability distribution, and not a hard classification. Techniques that allow this include relative frequency, maximum entropy models, and decision trees. For our experiments, we used maximum entropy learning. Specifics are deferred to Section 6.

5 Decoding

For the PCFG parsing model, we can find $argmax_{tree} P(tree|sentence)$ using a cubic-time dynamic programming-based algorithm. By adopting a more flexible probabilistic model, we sacrifice polynomial-time guarantees. The central question driving this paper is whether we can jettison these guarantees and still obtain good performance in practice. For the decoding of the probabilistic model of the previous section, we choose a depth-first branch-and-bound approach, specifically because of two advantages. First, this approach takes linear space. Second, it is anytime,

```
HLPDECODE(HLP $\mathcal{H}$, int $n$, assignment $\mathbf{w}$):
    1. Initialize stack $S$ with the pair $\langle \mathbf{x}_\emptyset, 1\rangle$, where
       $\mathbf{x}_\emptyset$ is the trivial assignment of $\mathbf{V}_\mathcal{L}$. Let
       $\mathbf{x}_{best} = \mathbf{x}_\emptyset$; let $p_{best} = 0$. Until stack $S$ is
       empty, repeat steps 2 to 4.
    2. Pop topmost pair $\langle \mathbf{x}, p\rangle$ from stack $S$.
    3. If $p > p_{best}$ and $\mathbf{x}$ is an $\mathcal{H}$-labeling of length
       $n$, then: let $\mathbf{x}_{best} = \mathbf{x}$; let $p_{best} = p$.
    4. If $p > p_{best}$ and $\mathbf{x}$ is not yet a $\mathcal{H}$-labeling of
       length $n$, then:
        (a) Let $Y$ be the earliest variable in $\mathbf{V}^n_\mathcal{L}$ (according
            to model order $<$) unassigned by $\mathbf{x}$.
        (b) If $Y \in dom(\mathbf{w})$, then push pair $\langle \mathbf{x}[Y = \mathbf{w}(Y)], p\rangle$
            onto stack $S$.
        (c) Else if $\mathcal{A}(Y, \mathbf{x}, n) = \langle true, y\rangle$ for some
            value $y \in dom(Y)$, then push pair $\langle \mathbf{x}[Y = y], p\rangle$
            onto stack $S$.
        (d) Otherwise for every value $y \in dom(Y)$,
            push pair $\langle \mathbf{x}[Y = y], p \cdot q(y)\rangle$ onto stack
            $S$ in ascending order of the value of $q(y)$, where:
            $q(y) = \begin{cases} P_S(y|\mathcal{F}^S(\mathbf{x}, i, j, n)) & \text{if } Y = S_{ij} \\ P_k(y|\mathcal{F}^k(\mathbf{x}, i, j, n)) & \text{if } Y = L^k_{ij} \end{cases}$
    5. Return $\mathbf{x}_{best}$.
```

Figure 6: Pseudocode for the decoder.

i.e. it finds a (typically good) solution early and improves this solution as the search progresses. Thus if one does not wish the spend the time to run the search to completion (and ensure optimality), one can use this algorithm easily as a heuristic by halting prematurely and taking the best solution found thus far.

The search space is simple to define. Given an HLP $\mathcal{H}$, the search algorithm simply makes assignments to the model variables (depth-first) in the order defined by $<$.

This search space can clearly grow to be quite large, however in practice the search speed is improved drastically by using branch-and-bound backtracking. Namely, at any choice point in the search space, we first choose the least cost child to expand (i.e. we make the most probable assignment). In this way, we quickly obtain a greedy solution (in linear time). After that point, we can continue to keep track of the best solution we have found so far, and if at any point we reach an internal node of our search tree with partial cost greater than the total cost of our best solution, we can discard this node and discontinue exploration of that subtree. This technique can result in a significant aggregrate savings of computation time, depending on the nature of the cost function.

Figure 6 shows the pseudocode for the depth-first branch-and-bound decoder. For an HLP $\mathcal{H} = \langle \mathcal{L}, <, \mathcal{A}, \mathcal{F}, \mathcal{P}\rangle$, a positive integer n, and a partial assignment $\mathbf{w}$ of $\mathbf{V}^n_\mathcal{L}$, the call HLPDECODE($\mathcal{H}$, n, $\mathbf{w}$) returns the $\mathcal{H}$-labeling $\mathbf{x}$ of length n such that $P(\mathbf{x}|n, \mathbf{w})$ is maximized.

6 Experiments

We employed a familiar experimental set-up. For training, we used sections 2–21 of the WSJ section of the Penn treebank. As a development set, we used the first 20 files of section 22, and then saved section 23 for testing the final model. One unconventional preprocessing step was taken. Namely, for the entire treebank, we compressed all unary chains into a single node, labeled with the label of the node furthest from the root. We did so in order to simplify our experiments, since the framework outlined in this paper allows only one label per labeling scheme per span. Thus by avoiding unary chains, we avoid the need for many labeling schemes or more complicated compound labels (labels like "NP-NN"). Since our goal here was not to create a parsing tool but rather to explore the viability of this approach, this seemed a fair concession. It should be noted that it is indeed possible to create a fully general parser using our framework (for instance, by using the above idea of compound labels for unary chains).

The main difficulty with this compromise is that it renders the familiar metrics of labeled precision and labeled recall incomparable with previous work (i.e. the LP of a set of candidate parses with respect to the unmodified test set differs from the LP with respect to the preprocessed test set). This would be a major problem, were it not for the existence of other metrics which measure only the quality of a parser's recursive decomposition of a sentence. Fortunately, such metrics do exist, thus we used *cross-bracketing statistics* as the basic measure of quality for our parser. The cross-bracketing score of a set of candidate parses with

word(i+k) = w	word(j+k) = w
preterminal(i+k) = p	**preterminal(j+k) = p**
label(i+k) = l	**label(j+k) = l**
category(i+k) = c	category(j+k) = c
signature(i,i+k) = s	

Figure 7: Basic feature templates used to determine constituency and labeling of span (i, j). k is an arbitrary integer.

	≤ 40		≤ 100	
	CB	0CB	CB	0CB
Magerman (1995)	1.26	56.6		
Collins (1996)	1.14	59.9		
Klein/Manning (2003)	1.10	60.3	1.31	57.2
this paper	**1.09**	**58.2**	**1.25**	**55.2**
Charniak (1997)	1.00	62.1		
Collins (1999)	0.90	67.1		

Figure 8: Cross-bracketing results for Section 23 of the Penn Treebank.

respect to the unmodified test set is identical to the cross-bracketing score with respect to the preprocessed test set, hence our preprocessing causes no comparability problems as viewed by this metric.

For our parsing model, we used an HLP $\mathcal{H} = \langle \mathcal{L}, <, \mathcal{A}, \mathcal{F}, \mathcal{P} \rangle$ with the following parameters. $\mathcal{L}$ consisted of three labeling schemes: the set L^{wd} of word labels, the set L^{pt} of preterminal labels, and the set L^{nt} of nonterminal labels. The order $<$ of the model variables was the unique order such that for all suitable integers i, j, k, l: (1) $S_{ij} < L^{wd}_{ij} < L^{pt}_{ij} < L^{nt}_{ij}$, (2) $L^{nt}_{ij} < S_{kl}$ iff span (i, j) is strictly shorter than span (k, l) or they have the same length and integer i is less than integer k. For auto-assignment function $\mathcal{A}$, we essentially used the function in Figure 5, modified so that it automatically assigned $null$ to model variables L^{wd}_{ij} and L^{pt}_{ij} for $i \neq j$ (i.e. no preterminal or word tagging of internal nodes), and to model variables L^{nt}_{ii} (i.e. no nonterminal tagging of leaves, rendered unnecessary by our preprocessing step).

Rather than incorporate part-of-speech tagging into the search process, we opted to pretag the sentences of our development and test sets with an off-the-shelf tagger, namely the Brill tagger (Brill, 1994). Thus the object of our computation was HLPDECODE($\mathcal{H}$, n, $\mathbf{w}$), where n was the length of the sentence, and partial assignment $\mathbf{w}$ specified the word and PT labels of the leaves. Given this partial assignment, the job of HLPDECODE was to find the most probable assignment of model variables S_{ij} and L^{nt}_{ij} for $1 \leq i < j \leq n$.

The two probability models, P^S and P^{nt}, were trained in the manner described in Section 4. Two decisions needed to be made: which features to use and which learning technique to employ. As for the learning technique, we used maximum entropy models, specifically the implementation called MegaM provided by Hal Daume (Daumé III, 2004). For P^S, we needed features that would be relevant to deciding whether a given span (i, j) should be considered a constituent. The basic building blocks we used are depicted in Figure 7. A few words of explanation are in order. By $label(k)$, we mean the highest nonterminal label so far assigned that covers word k, or if such a label does not yet exist, then the preterminal label of k (recall that our model order was bottom-up). By $category(k)$, we mean the category of the preterminal label of word k (given a coarser, hand-made categorization of preterminal labels that grouped all noun tags into one category, all verb tags into another, etc.). By $signature(k, m)$, where $k \leq m$, we mean the sequence $\langle label(k), label(k+1), ..., label(m) \rangle$, from which all consecutive sequences of identical labels are compressed into a single label. For instance, $\langle IN, NP, NP, VP, VP \rangle$ would become $\langle IN, NP, VP \rangle$. Ad-hoc conjunctions of these basic binary features were used as features for our probability model P^S. In total, approximately 800,000 such conjunctions were used.

For P^{nt}, we needed features that would be relevant to deciding which nonterminal label to give to a given constituent span. For this somewhat simpler task, we used a subset of the basic features used for P^S, shown in bold in Figure 7. Ad-hoc conjunctions of these boldface binary features were used as features for our probability model P^{nt}. In total, approximately 100,000 such conjunctions were used.

As mentioned earlier, we used cross-bracketing statistics as our basis of comparision. These results as shown in Figure 8. CB denotes the average cross-bracketing, i.e. the overall percentage of candidate constituents that properly overlap with a constituent in the gold parse. 0CB denotes the percentage of sentences in the test set that exhibit no cross-bracketing. With a simple feature set, we manage to obtain performance comparable to the unlexicalized PCFG parser of (Klein and Manning, 2003) on the set of sentences of length

40 or less. On the subset of Section 23 consisting of sentences of length 100 or less, our parser slightly outperforms their results in terms of average cross-bracketing. Interestingly, our parser has a lower percentage of sentences exhibiting no cross bracketing. To reconcile this result with the superior overall cross-bracketing score, it would appear that when our parser does make bracketing errors, the errors tend to be less severe.

The surprise was how quickly the parser performed. Despite its exponential worst-case time bounds, the search space turned out to be quite conducive to depth-first branch-and-bound pruning. Using an unoptimized Java implementation on a 4x Opteron 848 with 16GB of RAM, the parser required (on average) less than 0.26 seconds per sentence to optimally parse the subset of Section 23 comprised of sentences of 40 words or less. It required an average of 0.48 seconds per sentence to optimally parse the sentences of 100 words or less (an average of less than 3.5 seconds per sentence for those sentences of length 41-100). As noted earlier, the parser requires space linear in the size of the sentence.

7 Discussion

This project began with a question: can we develop a history-based parsing framework that is simple, general, and effective? We sought to provide a versatile probabilistic framework that would be free from the constraints that dynamic programming places on PCFG-based approaches. The work presented in this paper gives favorable evidence that more flexible (and worst-case intractable) probabilistic approaches can indeed perform well in practice, both in terms of running time and parsing quality.

We can extend this research in multiple directions. First, the set of features we selected were chosen with simplicity in mind, to see how well a simple and unadorned set of features would work, given our probabilistic model. A next step would be a more carefully considered feature set. For instance, although lexical information was used, it was employed in only a most basic sense. There was no attempt to use head information, which has been so successful in PCFG parsing methods.

Another parameter to experiment with is the model order, i.e. the order in which the model variables are assigned. In this work, we explored only one specific order (the left-to-right, leaves-to-head assignment) but in principle there are many other feasible orders. For instance, one could try a top-down approach, or a bottom-up approach in which internal nodes are assigned immediately after all of their descendants' values have been determined.

Throughout this paper, we strove to present the model in a very general manner. There is no reason why this framework cannot be tried in other application areas that rely on dynamic programming techniques to perform hierarchical labeling, such as phrase-based machine translation. Applying this framework to such application areas, as well as developing a general-purpose parser based on HLPs, are the subject of our continuing work.

References

Ezra Black, Fred Jelinek, John Lafferty, David M. Magerman, Robert Mercer, and Salim Roukos. 1993. Towards history-based grammars: using richer models for probabilistic parsing. In *Proc. ACL*.

Eric Brill. 1994. Some advances in rule-based part of speech tagging. In *Proc. AAAI*.

Eugene Charniak. 1997. Statistical parsing with a context-free grammar and word statistics. In *Proc. AAAI*.

Eugene Charniak. 2000. A maximum entropy-inspired parser. In *Proc. NAACL*.

Eugene Charniak. 2001. Immediate-head parsing for language models. In *Proc. ACL*.

Michael Collins. 1996. A new statistical parser based on bigram lexical dependencies. In *Proc. ACL*.

Michael Collins. 1999. *Head-driven statistical models for natural language parsing*. Ph.D. thesis, University of Pennsylvania.

Hal Daumé III. 2004. Notes on CG and LM-BFGS optimization of logistic regression. Paper available at http://www.isi.edu/ hdaume/docs/daume04cg-bfgs.ps, implementation available at http://www.isi.edu/ hdaume/megam/, August.

Mark Johnson. 1998. Pcfg models of linguistic tree representations. *Computational Linguistics*, 24:613–632.

Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *Proc. ACL*.

David M. Magerman. 1995. Statistical decision-tree models for parsing. In *Proc. ACL*.

Adwait Ratnaparkhi. 1997. A linear observed time statistical parser based on maximum entropy models. In *Proc. EMNLP*.