# Dynamic Programming for Linear-Time Incremental Parsing

**Liang Huang**
USC Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292
`lhuang@isi.edu`

**Kenji Sagae**
USC Institute for Creative Technologies
13274 Fiji Way
Marina del Rey, CA 90292
`sagae@ict.usc.edu`

## Abstract

Incremental parsing techniques such as shift-reduce have gained popularity thanks to their efficiency, but there remains a major problem: the search is *greedy* and only explores a tiny fraction of the whole space (even with beam search) as opposed to dynamic programming. We show that, surprisingly, dynamic programming is in fact possible for many shift-reduce parsers, by merging "equivalent" stacks based on feature values. Empirically, our algorithm yields up to a five-fold speedup over a state-of-the-art shift-reduce dependency parser with no loss in accuracy. Better search also leads to better learning, and our final parser outperforms all previously reported dependency parsers for English and Chinese, yet is much faster.

## 1 Introduction

In terms of search strategy, most parsing algorithms in current use for data-driven parsing can be divided into two broad categories: *dynamic programming* which includes the dominant CKY algorithm, and *greedy search* which includes most incremental parsing methods such as shift-reduce.[1] Both have pros and cons: the former performs an *exact* search (in cubic time) over an exponentially large space, while the latter is much faster (in linear-time) and is psycholinguistically motivated (Frazier and Rayner, 1982), but its greedy nature may suffer from severe search errors, as it only explores a tiny fraction of the whole space even with a beam.

Can we combine the advantages of both approaches, that is, construct an incremental parser that runs in (almost) linear-time, yet searches over a huge space with dynamic programming?

Theoretically, the answer is negative, as Lee (2002) shows that context-free parsing can be used to compute matrix multiplication, where sub-cubic algorithms are largely impractical.

We instead propose a dynamic programming algorithm for shift-reduce parsing which runs in polynomial time in theory, but linear-time (with beam search) in practice. The key idea is to merge equivalent stacks according to feature functions, inspired by Earley parsing (Earley, 1970; Stolcke, 1995) and generalized LR parsing (Tomita, 1991). However, our formalism is more flexible and our algorithm more practical. Specifically, we make the following contributions:

- *theoretically*, we show that for a large class of modern shift-reduce parsers, dynamic programming is in fact possible and runs in polynomial time as long as the feature functions are *bounded* and *monotonic* (which almost always holds in practice);
- *practically*, dynamic programming is up to five times faster (with the same accuracy) as conventional beam-search on top of a state-of-the-art shift-reduce dependency parser;
- as a by-product, dynamic programming can output a *forest* encoding exponentially many trees, out of which we can draw better and longer $k$-best lists than beam search can;
- finally, better and faster search also leads to better and faster learning. Our final parser achieves the best (unlabeled) accuracies that we are aware of in both English and Chinese among dependency parsers trained on the Penn Treebanks. Being linear-time, it is also much faster than most other parsers, even with a pure Python implementation.

[1] McDonald et al. (2005b) is a notable exception: the MST algorithm is exact search but not dynamic programming.

$$
\begin{array}{ll}
\text{input:} & w_0 \ldots w_{n-1} \\[6pt]
\text{axiom} & 0 : \langle 0,\ \epsilon \rangle : 0 \\[6pt]
\mathsf{sh} & \dfrac{\ell : \langle j,\ S \rangle : c}{\ell + 1 : \langle j+1,\ S | w_j \rangle : c + \xi} \quad j < n \\[12pt]
\mathsf{re}_{\curvearrowleft} & \dfrac{\ell : \langle j,\ S | s_1 | s_0 \rangle : c}{\ell + 1 : \langle j,\ S | s_1 {}^{\curvearrowleft} s_0 \rangle : c + \lambda} \\[12pt]
\mathsf{re}_{\curvearrowright} & \dfrac{\ell : \langle j,\ S | s_1 | s_0 \rangle : c}{\ell + 1 : \langle j,\ S | s_1 {}^{\curvearrowright} s_0 \rangle : c + \rho} \\[12pt]
\text{goal} & 2n - 1 : \langle n,\ s_0 \rangle : c
\end{array}
$$

where $\ell$ is the step, $c$ is the cost, and the shift cost $\xi$ and reduce costs $\lambda$ and $\rho$ are:

$$\xi = \mathbf{w} \cdot \mathbf{f}_{\mathsf{sh}}(j, S) \qquad (1)$$
$$\lambda = \mathbf{w} \cdot \mathbf{f}_{\mathsf{re}_{\curvearrowleft}}(j, S|s_1|s_0) \qquad (2)$$
$$\rho = \mathbf{w} \cdot \mathbf{f}_{\mathsf{re}_{\curvearrowright}}(j, S|s_1|s_0) \qquad (3)$$

Figure 1: Deductive system of vanilla shift-reduce.

For convenience of presentation and experimentation, we will focus on shift-reduce parsing for dependency structures in the remainder of this paper, though our formalism and algorithm can also be applied to phrase-structure parsing.

## 2 Shift-Reduce Parsing

### 2.1 Vanilla Shift-Reduce

Shift-reduce parsing performs a left-to-right scan of the input sentence, and at each step, choose one of the two actions: either *shift* the current word onto the stack, or *reduce* the top two (or more) items at the end of the stack (Aho and Ullman, 1972). To adapt it to dependency parsing, we split the reduce action into two cases, $\mathsf{re}_{\curvearrowleft}$ and $\mathsf{re}_{\curvearrowright}$, depending on which one of the two items becomes the head after reduction. This procedure is known as "arc-standard" (Nivre, 2004), and has been engineered to achieve state-of-the-art parsing accuracy in Huang et al. (2009), which is also the reference parser in our experiments.[2]

More formally, we describe a parser configuration by a *state* $\langle j, S \rangle$ where $S$ is a stack of trees $s_0, s_1, ...$ where $s_0$ is the top tree, and $j$ is the

[2]There is another popular variant, "arc-eager" (Nivre, 2004; Zhang and Clark, 2008), which is more complicated and less similar to the classical shift-reduce algorithm.

input: "I saw Al with Joe"

| *step* | *action* | *stack* | *queue* |
|---|---|---|---|
| 0 | - | | I ... |
| 1 | sh | I | saw ... |
| 2 | sh | I saw | Al ... |
| 3 | $\mathsf{re}_{\curvearrowleft}$ | $\text{I}^{\curvearrowleft}\text{saw}$ | Al ... |
| 4 | sh | $\text{I}^{\curvearrowleft}\text{saw}$ Al | with ... |
| 5a | $\mathsf{re}_{\curvearrowright}$ | $\text{I}^{\curvearrowleft}\text{saw}^{\curvearrowright}\text{Al}$ | with ... |
| 5b | sh | $\text{I}^{\curvearrowleft}\text{saw}$ Al with | Joe |

Figure 2: A trace of vanilla shift-reduce. After step (4), the parser branches off into (5a) or (5b).

queue head position (current word $q_0$ is $w_j$). At each step, we choose one of the three actions:

1. $\mathsf{sh}$: move the head of queue, $w_j$, onto stack $S$ as a singleton tree;
2. $\mathsf{re}_{\curvearrowleft}$: combine the top two trees on the stack, $s_0$ and $s_1$, and replace them with tree $s_1 {}^{\curvearrowleft} s_0$.
3. $\mathsf{re}_{\curvearrowright}$: combine the top two trees on the stack, $s_0$ and $s_1$, and replace them with tree $s_1 {}^{\curvearrowright} s_0$.

Note that the shorthand notation $t^{\curvearrowleft} t'$ denotes a new tree by "attaching tree $t'$ as the leftmost child of the root of tree $t$". This procedure can be summarized as a deductive system in Figure 1. States are organized according to step $\ell$, which denotes the number of actions accumulated. The parser runs in linear-time as there are exactly $2n-1$ steps for a sentence of $n$ words.

As an example, consider the sentence *"I saw Al with Joe"* in Figure 2. At step (4), we face a shift-reduce conflict: either combine "saw" and "Al" in a $\mathsf{re}_{\curvearrowright}$ action (5a), or shift "with" (5b). To resolve this conflict, there is a **cost** $c$ associated with each state so that we can pick the best one (or few, with a beam) at each step. Costs are accumulated in each step: as shown in Figure 1, actions $\mathsf{sh}$, $\mathsf{re}_{\curvearrowleft}$, and $\mathsf{re}_{\curvearrowright}$ have their respective costs $\xi, \lambda$, and $\rho$, which are dot-products of the weights $\mathbf{w}$ and *features* extracted from the state and the action.

### 2.2 Features

We view features as "abstractions" or (partial) observations of the current state, which is an important intuition for the development of dynamic programming in Section 3. **Feature templates** are functions that draw information from the *feature window* (see Tab. 1(b)), consisting of the top few trees on the stack and the first few words on the queue. For example, one such feature template $f_{100} = s_0.\mathsf{w} \circ q_0.\mathsf{t}$ is a conjunction of two **atomic features** $s_0$.w and $q_0$.t, capturing the root word of the top tree $s_0$ on the stack, and the part-of-speech tag of the current head word $q_0$ on the queue. See Tab. 1(a) for the list of feature templates used in the full model. Feature templates are instantiated for a specific state. For example, at step (4) in Fig. 2, the above template $f_{100}$ will generate a feature instance

$$(s_0.\mathsf{w} = \text{Al}) \circ (q_0.\mathsf{t} = \text{IN}).$$

More formally, we denote $\mathbf{f}$ to be the **feature function**, such that $\mathbf{f}(j, S)$ returns a vector of feature instances for state $\langle j, S\rangle$. To decide which action is the best for the current state, we perform a three-way classification based on $\mathbf{f}(j, S)$, and to do so, we further conjoin these feature instances with the action, producing action-conjoined instances like

$$(s_0.\mathsf{w} = \text{Al}) \circ (q_0.\mathsf{t} = \text{IN}) \circ (action = \mathsf{sh}).$$

We denote $\mathbf{f}_{\mathsf{sh}}(j, S)$, $\mathbf{f}_{\mathsf{re}_{\curvearrowleft}}(j, S)$, and $\mathbf{f}_{\mathsf{re}_{\curvearrowright}}(j, S)$ to be the conjoined feature instances, whose dot-products with the weight vector decide the best action (see Eqs. (1-3) in Fig. 1).

### 2.3 Beam Search and Early Update

To improve on strictly greedy search, shift-reduce parsing is often enhanced with beam search (Zhang and Clark, 2008), where $b$ states develop in parallel. At each step we extend the states in the current beam by applying one of the three actions, and then choose the best $b$ resulting states for the next step. Our dynamic programming algorithm also runs on top of beam search in practice.

To train the model, we use the averaged perceptron algorithm (Collins, 2002). Following Collins and Roark (2004) we also use the "early-update" strategy, where an update happens whenever the gold-standard action-sequence falls off the beam, with the rest of the sequence neglected.[3] The intuition behind this strategy is that later mistakes are often caused by previous ones, and are irrelevant when the parser is on the wrong track. Dynamic programming turns out to be a great fit for early updating (see Section 4.3 for details).

## 3 Dynamic Programming (DP)

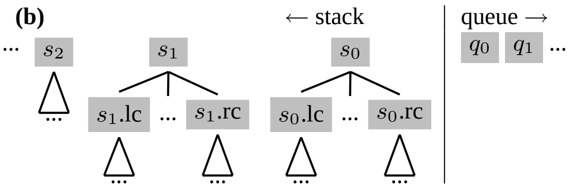
### 3.1 Merging Equivalent States

The key observation for dynamic programming is to merge "equivalent states" in the same beam

[3]As a special case, for the deterministic mode ($b$=1), updates always co-occur with the first mistake made.

| (a) | **Features Templates** $\mathbf{f}(j, S)$ | | $q_i = w_{j+i}$ |
|---|---|---|---|
| (1) | $s_0$.w | $s_0$.t | $s_0$.w ∘ $s_0$.t |
| | $s_1$.w | $s_1$.t | $s_1$.w ∘ $s_1$.t |
| | $q_0$.w | $q_0$.t | $q_0$.w ∘ $q_0$.t |
| (2) | $s_0$.w ∘ $s_1$.w | | $s_0$.t ∘ $s_1$.t |
| | $s_0$.t ∘ $q_0$.t | | $s_0$.w ∘ $s_0$.t ∘ $s_1$.t |
| | $s_0$.t ∘ $s_1$.w ∘ $s_1$.t | | $s_0$.w ∘ $s_1$.w ∘ $s_1$.t |
| | $s_0$.w ∘ $s_0$.t ∘ $s_1$.w | | $s_0$.w ∘ $s_0$.t ∘ $s_1$ ∘ $s_1$.t |
| (3) | $s_0$.t ∘ $q_0$.t ∘ $q_1$.t | | $s_1$.t ∘ $s_0$.t ∘ $q_0$.t |
| | $s_0$.w ∘ $q_0$.t ∘ $q_1$.t | | $s_1$.t ∘ $s_0$.w ∘ $q_0$.t |
| (4) | $s_1$.t ∘ $s_1$.lc.t ∘ $s_0$.t | | $s_1$.t ∘ $s_1$.rc.t ∘ $s_0$.t |
| | $s_1$.t ∘ $s_0$.t ∘ $s_0$.rc.t | | $s_1$.t ∘ $s_1$.lc.t ∘ $s_0$ |
| | $s_1$.t ∘ $s_1$.rc.t ∘ $s_0$.w | | $s_1$.t ∘ $s_0$.w ∘ $s_0$.lc.t |
| (5) | $s_2$.t ∘ $s_1$.t ∘ $s_0$.t | | |

**(b)**



| (c) | **Kernel features for DP** $\widetilde{\mathbf{f}}(j, S) = (j, \mathbf{f}_2(s_2), \mathbf{f}_1(s_1), \mathbf{f}_0(s_0))$ | | | |
|---|---|---|---|---|
| $\mathbf{f}_2(s_2)$ | $s_2$.t | | | |
| $\mathbf{f}_1(s_1)$ | $s_1$.w | $s_1$.t | $s_1$.lc.t | $s_1$.rc.t |
| $\mathbf{f}_0(s_0)$ | $s_0$.w | $s_0$.t | $s_0$.lc.t | $s_0$.rc.t |
| $j$ | $q_0$.w | $q_0$.t | $q_1$.t | |

Table 1: **(a)** feature templates used in this work, adapted from Huang et al. (2009). $x$.w and $x$.t denotes the root word and POS tag of tree (or word) $x$. and $x$.lc and $x$.rc denote $x$'s left- and rightmost child. **(b)** feature window. **(c)** kernel features.

(i.e., same step) if they have the same feature values, because they will have the same costs as shown in the deductive system in Figure 1. Thus we can define two states $\langle j, S\rangle$ and $\langle j', S'\rangle$ to be equivalent, notated $\langle j, S\rangle \sim \langle j', S'\rangle$, iff.

$$j = j' \quad \text{and} \quad \mathbf{f}(j, S) = \mathbf{f}(j', S'). \qquad (4)$$

Note that $j = j'$ is also needed because the queue head position $j$ determines which word to shift next. In practice, however, a small subset of atomic features will be enough to determine the whole feature vector, which we call **kernel features** $\widetilde{\mathbf{f}}(j, S)$, defined as the *smallest set* of atomic templates such that

$$\widetilde{\mathbf{f}}(j, S) = \widetilde{\mathbf{f}}(j', S') \Rightarrow \langle j, S\rangle \sim \langle j', S'\rangle.$$

For example, the full list of 28 feature templates in Table 1(a) can be determined by just 12 atomic features in Table 1(c), which just look at the root words and tags of the top two trees on stack, as well as the tags of their left- and rightmost children, plus the root tag of the third tree $s_2$, and finally the word and tag of the queue head $q_0$ and the

$$\begin{array}{cl}
\text{state form} & \ell : \langle i,\ j,\ s_d...s_0\rangle \colon (c, v, \pi) \qquad \ell\text{: step; } c,\ v\text{: prefix and inside costs; } \pi\text{: predictor states} \\
\text{equivalence} & \ell : \langle i,\ j,\ s_d...s_0\rangle \sim \ell : \langle i,\ j,\ s'_d...s'_0\rangle \quad \text{iff.} \quad \widetilde{\mathbf{f}}(j, s_d...s_0) = \widetilde{\mathbf{f}}(j, s'_d...s'_0) \\
\text{ordering} & \ell : \_ :\ (c, v, \_) \prec \ell : \_ :\ (c', v', \_) \quad \text{iff. } c < c' \text{ or } (c = c' \text{ and } v < v'). \\
\hline
\text{axiom } (p_0) & 0 : \langle 0,\ 0,\ \epsilon\rangle \colon\ (0, 0, \emptyset) \\
\mathsf{sh} & \dfrac{\overset{\text{state } p:}{\ell : \langle \_,\ j,\ s_d...s_0\rangle \colon\ (c, \_, \_)}}{\ell + 1 : \langle j,\ j+1,\ s_{d-1}...s_0, w_j\rangle\ :\ (c + \xi,\ 0,\ \{p\})}\ j < n \\
\mathsf{re}_{\curvearrowleft} & \dfrac{\overset{\text{state } p:}{\_ : \langle k,\ i,\ s'_d...s'_0\rangle \colon\ (c', v', \pi')} \qquad \overset{\text{state } q:}{\ell : \langle i,\ j,\ s_d...s_0\rangle \colon\ (\_, v, \pi)}}{\ell + 1 : \langle k,\ j,\ s'_d...s'_1, s'^{\curvearrowleft}_0 s_0\rangle\ :\ (c' + v + \delta,\ v' + v + \delta,\ \pi')}\ \ p \in \pi \\
\text{goal} & 2n - 1 : \langle 0,\ n,\ s_d...s_0\rangle \colon\ (c,\ c, \{p_0\})
\end{array}$$

where $\xi = \mathbf{w} \cdot \mathbf{f}_{\mathsf{sh}}(j, s_d...s_0)$, and $\delta = \xi' + \lambda$, with $\xi' = \mathbf{w} \cdot \mathbf{f}_{\mathsf{sh}}(i, s'_d...s'_0)$ and $\lambda = \mathbf{w} \cdot \mathbf{f}_{\mathsf{re}_{\curvearrowleft}}(j, s_d...s_0)$.

Figure 3: Deductive system for shift-reduce parsing with dynamic programming. The predictor state set $\pi$ is an implicit graph-structured stack (Tomita, 1988) while the prefix cost $c$ is inspired by Stolcke (1995). The $\mathsf{re}_{\curvearrowright}$ case is similar, replacing $s'^{\curvearrowleft}_0 s_0$ with $s'^{\curvearrowright}_0 s_0$, and $\lambda$ with $\rho = \mathbf{w} \cdot \mathbf{f}_{\mathsf{re}_{\curvearrowright}}(j, s_d...s_0)$. Irrelevant information in a deduction step is marked as an underscore (_) which means "can match anything".

tag of the next word $q_1$. Since the queue is *static information* to the parser (unlike the stack, which changes dynamically), we can use $j$ to replace features from the queue. So in general we write

$$\widetilde{\mathbf{f}}(j, S) = (j, \mathbf{f}_d(s_d), \ldots, \mathbf{f}_0(s_0))$$

if the feature window looks at top $d + 1$ trees on stack, and where $\mathbf{f}_i(s_i)$ extracts kernel features from tree $s_i$ ($0 \leq i \leq d$). For example, for the full model in Table 1(a) we have

$$\widetilde{\mathbf{f}}(j, S) = (j, \mathbf{f}_2(s_2), \mathbf{f}_1(s_1), \mathbf{f}_0(s_0)), \qquad (5)$$

where $d = 2$, $\mathbf{f}_2(x) = x.\mathsf{t}$, and $\mathbf{f}_1(x) = \mathbf{f}_0(x) = (x.\mathsf{w}, x.\mathsf{t}, x.\mathsf{lc.t}, x.\mathsf{rc.t})$ (see Table 1(c)).

## 3.2 Graph-Structured Stack and Deduction

Now that we have the kernel feature functions, it is intuitive that we might only need to remember the relevant bits of information from only the *last* $(d + 1)$ *trees* on stack instead of the whole stack, because they provide all the relevant information for the features, and thus determine the costs. For shift, this suffices as the stack grows on the right; but for reduce actions the stack shrinks, and in order still to maintain $d + 1$ trees, we have to know something about the history. This is exactly why we needed the full stack for vanilla shift-reduce parsing in the first place, and why dynamic programming seems hard here.

To solve this problem we borrow the idea of "graph-structured stack" (GSS) from Tomita (1991). Basically, each state $p$ carries with it a set $\pi(p)$ of **predictor states**, each of which can be combined with $p$ in a reduction step. In a shift step, if state $p$ generates state $q$ (we say "$p$ predicts $q$" in Earley (1970) terms), then $p$ is added onto $\pi(q)$. When two equivalent shifted states get merged, their predictor states get combined. In a reduction step, state $q$ tries to combine with every predictor state $p \in \pi(q)$, and the resulting state $r$ inherits the predictor states set from $p$, i.e., $\pi(r) = \pi(p)$. Interestingly, when two equivalent reduced states get merged, we can prove (by induction) that their predictor states are identical (proof omitted).

Figure 3 shows the new deductive system with dynamic programming and GSS. A new state has the form

$$\ell : \langle i,\ j,\ s_d...s_0\rangle$$

where $[i..j]$ is the span of the top tree $s_0$, and $s_d..s_1$ are merely "left-contexts". It can be combined with some predictor state $p$ spanning $[k..i]$

$$\ell' : \langle k,\ i,\ s'_d...s'_0\rangle$$

to form a larger state spanning $[k..j]$, with the resulting top tree being either $s_1^{\curvearrowleft} s_0$ or $s_1^{\curvearrowright} s_0$.

This style resembles CKY and Earley parsers. In fact, the *chart* in Earley and other agenda-based parsers is indeed a GSS when viewed left-to-right. In these parsers, when a state is popped up from the agenda, it looks for possible sibling states that can combine with it; GSS, however, *explicitly* maintains these predictor states so that the newly-popped state does not need to look them up.[4]

## 3.3 Correctness and Polynomial Complexity

We state the main theoretical result with the proof omitted due to space constraints:

**Theorem 1.** *The deductive system is optimal and runs in worst-case polynomial time as long as the kernel feature function satisfies two properties:*

- **bounded**: $\widetilde{\mathbf{f}}(j, S) = (j, \mathbf{f}_d(s_d), \ldots, \mathbf{f}_0(s_0))$ *for some constant* $d$, *and each* $|\mathbf{f}_t(x)|$ *also bounded by a constant for all possible tree* $x$.
- **monotonic**: $\mathbf{f}_t(x) = \mathbf{f}_t(y) \Rightarrow \mathbf{f}_{t+1}(x) = \mathbf{f}_{t+1}(y)$, *for all* $t$ *and all possible trees* $x, y$.

Intuitively, boundedness means features can only look at a local window and can only extract bounded information on each tree, which is always the case in practice since we can not have infinite models. Monotonicity, on the other hand, says that features drawn from trees farther away from the top should *not* be more refined than from those closer to the top. This is also natural, since the information most relevant to the current decision is always around the stack top. For example, the kernel feature function in Eq. 5 is bounded and monotonic, since $\mathbf{f}_2$ is less refined than $\mathbf{f}_1$ and $\mathbf{f}_0$.

These two requirements are related to grammar refinement by annotation (Johnson, 1998), where annotations must be bounded and monotonic: for example, one cannot refine a grammar by only remembering the grandparent but not the parent symbol. The difference here is that the annotations are not vertical ((grand-)parent), but rather *horizontal* (left context). For instance, a context-free rule $A \rightarrow B\ C$ would become ${}^{D}A \rightarrow {}^{D}B\ {}^{B}C$ for some $D$ if there exists a rule $E \rightarrow \alpha D A \beta$. This resembles the reduce step in Fig. 3.

The very high-level idea of the proof is that boundedness is crucial for polynomial-time, while monotonicity is used for the optimal substructure property required by the correctness of DP.

[4]In this sense, GSS (Tomita, 1988) is really not a new invention: an efficient implementation of Earley (1970) should already have it implicitly, similar to what we have in Fig. 3.

## 3.4 Beam Search based on Prefix Cost

Though the DP algorithm runs in polynomial-time, in practice the complexity is still too high, esp. with a rich feature set like the one in Table 1. So we apply the same beam search idea from Sec. 2.3, where each step can accommodate only the best $b$ states. To decide the ordering of states in each beam we borrow the concept of **prefix cost** from Stolcke (1995), originally developed for weighted Earley parsing. As shown in Fig. 3, the prefix cost $c$ is the total cost of the best action sequence from the initial state to the end of state $p$, i.e., it includes both the **inside cost** $v$ (for Viterbi inside derivation), and the cost of the (best) path leading towards the beginning of state $p$. We say that a state $p$ with prefix cost $c$ is better than a state $p'$ with prefix cost $c'$, notated $p \prec p'$ in Fig. 3, if $c < c'$. We can also prove (by contradiction) that optimizing for prefix cost implies optimal inside cost (Nederhof, 2003, Sec. 4). [5]

As shown in Fig. 3, when a state $q$ with costs $(c, v)$ is combined with a predictor state $p$ with costs $(c', v')$, the resulting state $r$ will have costs

$$(c' + v + \delta,\ v' + v + \delta),$$

where the inside cost is intuitively the combined inside costs plus an additional combo cost $\delta$ from the combination, while the resulting prefix cost $c' + v + \delta$ is the sum of the prefix cost of the predictor state $q$, the inside cost of the current state $p$, and the combo cost. Note the prefix cost of $q$ is irrelevant. The combo cost $\delta = \xi' + \lambda$ consists of shift cost $\xi'$ of $p$ and reduction cost $\lambda$ of $q$.

The cost in the non-DP shift-reduce algorithm (Fig. 1) is indeed a prefix cost, and the DP algorithm subsumes the non-DP one as a special case where no two states are equivalent.

## 3.5 Example: Edge-Factored Model

As a concrete example, Figure 4 simulates an edge-factored model (Eisner, 1996; McDonald et al., 2005a) using shift-reduce with dynamic programming, which is similar to bilexical PCFG parsing using CKY (Eisner and Satta, 1999). Here the kernel feature function is

$$\widetilde{\mathbf{f}}(j, S) = (j, h(s_1), h(s_0))$$

[5]Note that using inside cost $v$ for ordering would be a bad idea, as it will always prefer shorter derivations like in best-first parsing. As in A* search, we need some estimate of "outside cost" to predict which states are more promising, and the prefix cost includes an exact cost for the left outside context, but no right outside context.

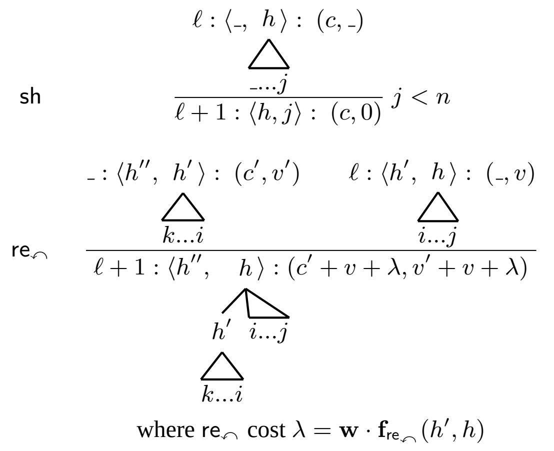

sh
re↶
where re↶ cost $\lambda = \mathbf{w} \cdot \mathbf{f}_{\text{re}_\curvearrowleft}(h', h)$

Figure 4: Example of shift-reduce with dynamic programming: simulating an edge-factored model. GSS is implicit here, and re↷ case omitted.

where $h(x)$ returns the head word index of tree $x$, because all features in this model are based on the head and modifier indices in a dependency link. This function is obviously bounded and monotonic in our definitions. The theoretical complexity of this algorithm is $O(n^7)$ because in a reduction step we have three span indices and three head indices, plus a step index $\ell$. By contrast, the naïve CKY algorithm for this model is $O(n^5)$ which can be improved to $O(n^3)$ (Eisner, 1996).[6] The higher complexity of our algorithm is due to two factors: first, we have to maintain both $h$ and $h'$ in one state, because the current shift-reduce model can not draw features *across* different states (unlike CKY); and more importantly, we group states by step $\ell$ in order to achieve incrementality and linear runtime with beam search that is not (easily) possible with CKY or MST.

## 4 Experiments

We first reimplemented the reference shift-reduce parser of Huang et al. (2009) in Python (henceforth "non-DP"), and then extended it to do dynamic programing (henceforth "DP"). We evaluate their performances on the standard Penn Treebank (PTB) English dependency parsing task[7] using the standard split: secs 02-21 for training, 22 for development, and 23 for testing. Both DP and non-DP parsers use the same feature templates in Table 1. For Secs. 4.1-4.2, we use a baseline model trained with non-DP for both DP and non-DP, so that we can do a side-by-side comparison of search quality; in Sec. 4.3 we will retrain the model with DP and compare it against training with non-DP.

### 4.1 Speed Comparisons

To compare parsing speed between DP and non-DP, we run each parser on the development set, varying the beam width $b$ from 2 to 16 (DP) or 64 (non-DP). Fig. 5a shows the relationship between search quality (as measured by the average model score per sentence, higher the better) and speed (average parsing time per sentence), where DP with a beam width of $b$=16 achieves the same search quality with non-DP at $b$=64, while being 5 times faster. Fig. 5b shows a similar comparison for dependency accuracy. We also test with an edge-factored model (Sec. 3.5) using feature templates (1)-(3) in Tab. 1, which is a subset of those in McDonald et al. (2005b). As expected, this difference becomes more pronounced (8 times faster in Fig. 5c), since the less expressive feature set makes more states "equivalent" and mergeable in DP. Fig. 5d shows the (almost linear) correlation between dependency accuracy and search quality, confirming that better search yields better parsing.

### 4.2 Search Space, Forest, and Oracles

DP achieves better search quality because it expores an exponentially large search space rather than only $b$ trees allowed by the beam (see Fig. 6a). As a by-product, DP can output a *forest* encoding these exponentially many trees, out of which we can draw longer and better (in terms of oracle) $k$-best lists than those in the beam (see Fig. 6b). The forest itself has an oracle of 98.15 (as if $k \to \infty$), computed à la Huang (2008, Sec. 4.1). These candidate sets may be used for reranking (Charniak and Johnson, 2005; Huang, 2008).[8]

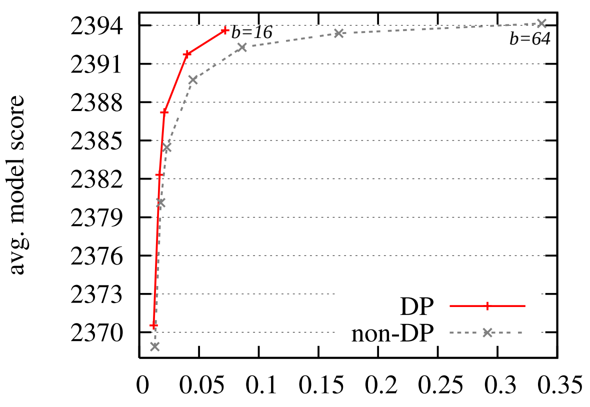
### 4.3 Perceptron Training and Early Updates

Another interesting advantage of DP over non-DP is the faster training with perceptron, even when both parsers use the same beam width. This is due to the use of early updates (see Sec. 2.3), which happen much more often with DP, because a gold-standard state $p$ is often merged with an equivalent (but incorrect) state that has a higher model score, which triggers update immediately. By contrast, in non-DP beam search, states such as $p$ might still

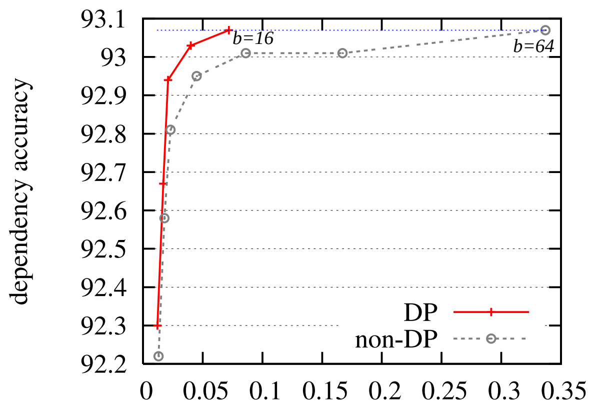
[6]Or $O(n^2)$ with MST, but including non-projective trees.

[7]Using the head rules of Yamada and Matsumoto (2003).

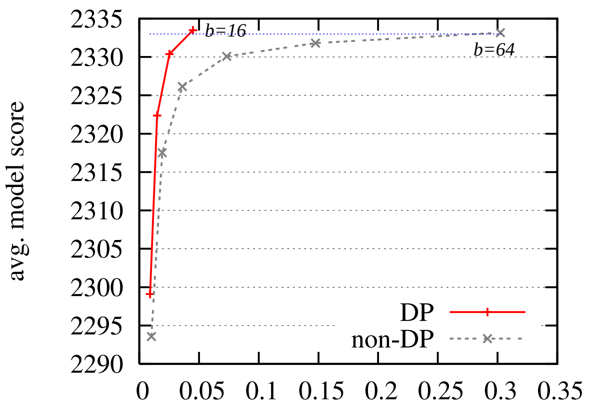
[8]DP's $k$-best lists are extracted from the forest using the algorithm of Huang and Chiang (2005), rather than those in the final beam as in the non-DP case, because many derivations have been merged during dynamic programming.



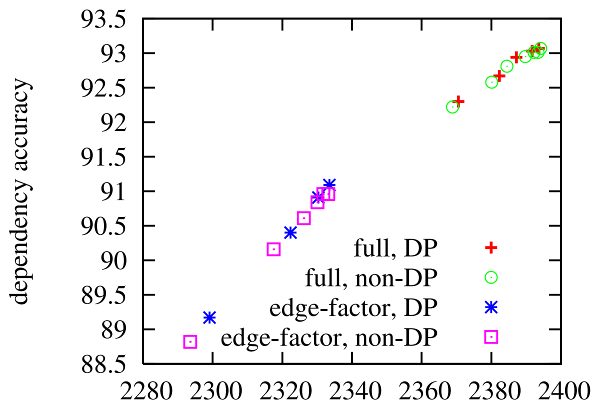
(a) search quality vs. time (full model)



(b) parsing accuracy vs. time (full model)

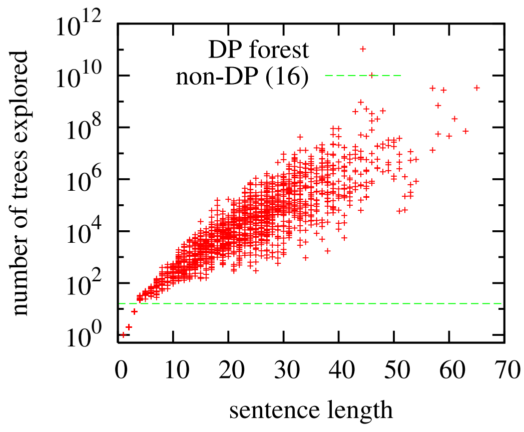


(c) search quality vs. time (edge-factored model)

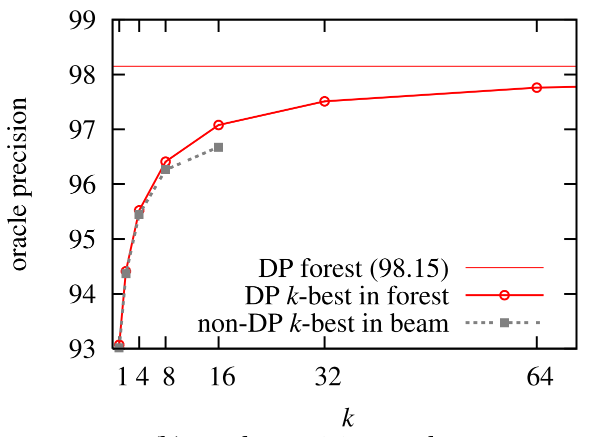


(d) correlation b/w parsing ($y$) and search ($x$)

Figure 5: Speed comparisons between DP and non-DP, with beam size $b$ ranging 2∼16 for DP and 2∼64 for non-DP. Speed is measured by avg. parsing time (secs) per sentence on $x$ axis. With the same level of search quality or parsing accuracy, DP (at $b$=16) is ∼4.8 times faster than non-DP (at $b$=64) with the full model in plots (a)-(b), or ∼8 times faster with the simplified edge-factored model in plot (c). Plot (d) shows the (roughly linear) correlation between parsing accuracy and search quality (avg. model score).



(a) sizes of search spaces



(b) oracle precision on dev

Figure 6: DP searches over a forest of *exponentially many* trees, which also produces better and longer $k$-best lists with higher oracles, while non-DP only explores $b$ trees allowed in the beam ($b = 16$ here).

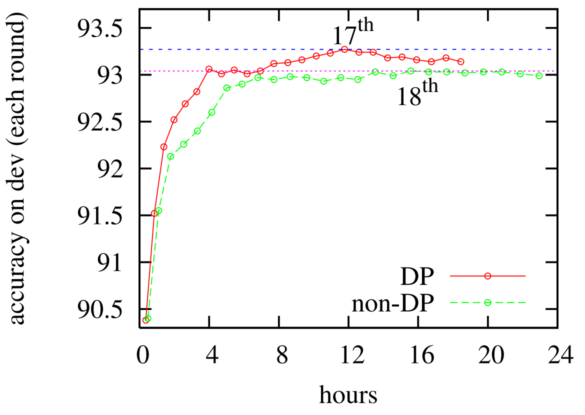


Figure 7: Learning curves (showing precision on dev) of perceptron training for 25 iterations ($b$=8). DP takes 18 hours, peaking at the 17th iteration (93.27%) with 12 hours, while non-DP takes 23 hours, peaking at the 18th (93.04%) with 16 hours.

survive in the beam throughout, even though it is no longer possible to rank the best in the beam.

The higher frequency of early updates results in faster iterations of perceptron training. Table 2 shows the percentage of early updates and the time per iteration during training. While the number of updates is roughly comparable between DP and non-DP, the rate of early updates is much higher with DP, and the time per iteration is consequently shorter. Figure 7 shows that training with DP is about 1.2 times faster than non-DP, and achieves +0.2% higher accuracy on the dev set (93.27%).

Besides training with gold POS tags, we also trained on noisy tags, since they are closer to the test setting (automatic tags on sec 23). In that case, we tag the dev and test sets using an automatic POS tagger (at 97.2% accuracy), and tag the training set using four-way jackknifing similar to Collins (2000), which contributes another +0.1% improvement in accuracy on the test set. Faster training also enables us to incorporate more features, where we found more lookahead features ($q_2$) results in another +0.3% improvement.

### 4.4 Final Results on English and Chinese

Table 3 presents the final test results of our DP parser on the Penn English Treebank, compared with other state-of-the-art parsers. Our parser achieves the highest (unlabeled) dependency accuracy among dependency parsers trained on the Treebank, and is also much faster than most other parsers even with a pure Python implementation

| *it* | update | early% | time | update | early% | time |
|---|---|---|---|---|---|---|
| 1 | 31943 | 98.9 | 22 | 31189 | 87.7 | 29 |
| 5 | 20236 | 98.3 | 38 | 19027 | 70.3 | 47 |
| 17 | 8683 | 97.1 | 48 | 7434 | 49.5 | 60 |
| 25 | 5715 | 97.2 | 51 | 4676 | 41.2 | 65 |

Table 2: Perceptron iterations with DP (left) and non-DP (right). Early updates happen much more often with DP due to equivalent state merging, which leads to faster training (time in minutes).

| | word | L | time | comp. |
|---|---|---|---|---|
| McDonald 05b | 90.2 | Ja | 0.12 | $O(n^2)$ |
| McDonald 05a | 90.9 | Ja | 0.15 | $O(n^3)$ |
| Koo 08 base | 92.0 | – | – | $O(n^4)$ |
| Zhang 08 single | 91.4 | C | 0.11 | $O(n)^‡$ |
| **this work** | **92.1** | **Py** | **0.04** | $O(n)$ |
| †Charniak 00 | 92.5 | C | 0.49 | $O(n^5)$ |
| †Petrov 07 | 92.4 | Ja | 0.21 | $O(n^3)$ |
| Zhang 08 combo | 92.1 | C | – | $O(n^2)^‡$ |
| Koo 08 semisup | 93.2 | – | – | $O(n^4)$ |

Table 3: Final test results on English (PTB). Our parser (in pure Python) has the highest accuracy among dependency parsers trained on the Treebank, and is also much faster than major parsers. †converted from constituency trees. C=C/C++, Py=Python, Ja=Java. Time is in seconds per sentence. Search spaces: ‡linear; others exponential.

(on a 3.2GHz Xeon CPU). Best-performing constituency parsers like Charniak (2000) and Berkeley (Petrov and Klein, 2007) do outperform our parser, since they consider more information during parsing, but they are at least 5 times slower. Figure 8 shows the parse time in seconds for each test sentence. The observed time complexity of our DP parser is in fact linear compared to the superlinear complexity of Charniak, MST (McDonald et al., 2005b), and Berkeley parsers. Additional techniques such as semi-supervised learning (Koo et al., 2008) and parser combination (Zhang and Clark, 2008) do achieve accuracies equal to or higher than ours, but their results are not directly comparable to ours since they have access to extra information like unlabeled data. Our technique is orthogonal to theirs, and combining these techniques could potentially lead to even better results.

We also test our final parser on the Penn Chinese Treebank (CTB5). Following the set-up of Duan et al. (2007) and Zhang and Clark (2008), we split CTB5 into training (secs 001-815 and 1001-

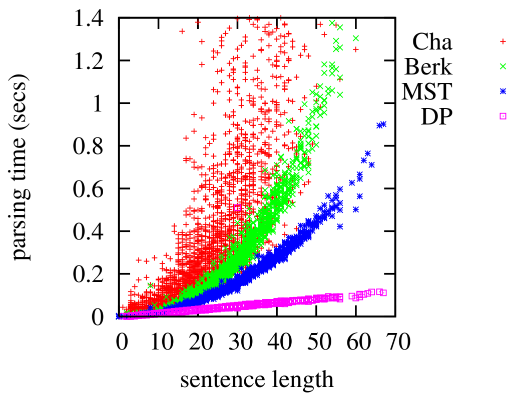


Figure 8: Scatter plot of parsing time against sentence length, comparing with Charniak, Berkeley, and the $O(n^2)$ MST parsers.

| | word | non-root | root | compl. |
|---|---|---|---|---|
| Duan 07 | 83.88 | 84.36 | 73.70 | 32.70 |
| Zhang 08$^\dagger$ | 84.33 | 84.69 | 76.73 | 32.79 |
| **this work** | 85.20 | 85.52 | 78.32 | 33.72 |

Table 4: Final test results on Chinese (CTB5). $^\dagger$The transition parser in Zhang and Clark (2008).

1136), development (secs 886-931 and 1148-1151), and test (secs 816-885 and 1137-1147) sets, assume gold-standard POS-tags for the input, and use the head rules of Zhang and Clark (2008). Table 4 summarizes the final test results, where our work performs the best in all four types of (unlabeled) accuracies: word, non-root, root, and complete match (all excluding punctuations). [9,10]

## 5 Related Work

This work was inspired in part by Generalized LR parsing (Tomita, 1991) and the graph-structured stack (GSS). Tomita uses GSS for exhaustive LR parsing, where the GSS is equivalent to a dynamic programming chart in chart parsing (see Footnote 4). In fact, Tomita's GLR is an instance of techniques for tabular simulation of nondeterministic pushdown automata based on deductive systems (Lang, 1974), which allow for cubic-time exhaustive shift-reduce parsing with context-free grammars (Billot and Lang, 1989).

Our work advances this line of research in two aspects. First, ours is more general than GLR in that it is not restricted to LR (a special case of shift-reduce), and thus does not require building an LR table, which is impractical for modern grammars with a large number of rules or features. In contrast, we employ the ideas behind GSS more flexibly to merge states based on features values, which can be viewed as constructing an implicit LR table *on-the-fly*. Second, unlike previous theoretical results about cubic-time complexity, we achieved linear-time performance by smart beam search with prefix cost inspired by Stolcke (1995), allowing for state-of-the-art data-driven parsing.

To the best of our knowledge, our work is the first linear-time incremental parser that performs dynamic programming. The parser of Roark and Hollingshead (2009) is also almost linear time, but they achieved this by discarding parts of the CKY chart, and thus do achieve incrementality.

## 6 Conclusion

We have presented a dynamic programming algorithm for shift-reduce parsing, which runs in linear-time in practice with beam search. This framework is general and applicable to a large-class of shift-reduce parsers, as long as the feature functions satisfy boundedness and monotonicity. Empirical results on a state-the-art dependency parser confirm the advantage of DP in many aspects: faster speed, larger search space, higher oracles, and better and faster learning. Our final parser outperforms all previously reported dependency parsers trained on the Penn Treebanks for both English and Chinese, and is much faster in speed (even with a Python implementation). For future work we plan to extend it to constituency parsing.

## Acknowledgments

We thank David Chiang, Yoav Goldberg, Jonathan Graehl, Kevin Knight, and Roger Levy for helpful discussions and the three anonymous reviewers for comments. Mark-Jan Nederhof inspired the use of prefix cost. Yue Zhang helped with Chinese datasets, and Wenbin Jiang with feature sets. This work is supported in part by DARPA GALE Contract No. HR0011-06-C-0022 under subcontract to BBN Technologies, and by the U.S. Army Research, Development, and Engineering Command (RDECOM). Statements and opinions expressed do not necessarily reflect the position or the policy of the United States Government, and no official endorsement should be inferred.

[9]Duan et al. (2007) and Zhang and Clark (2008) did not report word accuracies, but those can be recovered given non-root and root ones, and the number of non-punctuation words.

[10]Parser combination in Zhang and Clark (2008) achieves a higher word accuracy of 85.77%, but again, it is not directly comparable to our work.

## References

Alfred V. Aho and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling*, volume I: Parsing of *Series in Automatic Computation*. Prentice Hall, Englewood Cliffs, New Jersey.

S. Billot and B. Lang. 1989. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th ACL*, pages 143–151.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine-grained $n$-best parsing and discriminative reranking. In *Proceedings of the 43rd ACL*, Ann Arbor, MI.

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

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proceedings of ACL*.

Michael Collins. 2000. Discriminative reranking for natural language parsing. In *Proceedings of ICML*, pages 175–182.

Michael Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of EMNLP*.

Xiangyu Duan, Jun Zhao, and Bo Xu. 2007. Probabilistic models for action-based chinese dependency parsing. In *Proceedings of ECML/PKDD*.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Jason Eisner and Giorgio Satta. 1999. Efficient parsing for bilexical context-free grammars and head-automaton grammars. In *Proceedings of ACL*.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of COLING*.

Lyn Frazier and Keith Rayner. 1982. Making and correcting errors during sentence comprehension: Eye movements in the analysis of structurally ambiguous sentences. *Cognitive Psychology*, 14(2):178 – 210.

Liang Huang and David Chiang. 2005. Better $k$-best Parsing. In *Proceedings of the Ninth International Workshop on Parsing Technologies (IWPT-2005)*.

Liang Huang, Wenbin Jiang, and Qun Liu. 2009. Bilingually-constrained (monolingual) shift-reduce parsing. In *Proceedings of EMNLP*.

Liang Huang. 2008. Forest reranking: Discriminative parsing with non-local features. In *Proceedings of the ACL: HLT*, Columbus, OH, June.

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

Terry Koo, Xavier Carreras, and Michael Collins. 2008. Simple semi-supervised dependency parsing. In *Proceedings of ACL*.

B. Lang. 1974. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Languages and Programming, 2nd Colloquium*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269, Saarbrücken. Springer-Verlag.

Lillian Lee. 2002. Fast context-free grammar parsing requires fast Boolean matrix multiplication. *Journal of the ACM*, 49(1):1–15.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005a. Online large-margin training of dependency parsers. In *Proceedings of the 43rd ACL*.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005b. Non-projective dependency parsing using spanning tree algorithms. In *Proc. of HLT-EMNLP*.

Mark-Jan Nederhof. 2003. Weighted deductive parsing and Knuth's algorithm. *Computational Linguistics*, pages 135–143.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Incremental Parsing: Bringing Engineering and Cognition Together. Workshop at ACL-2004*, Barcelona.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *Proceedings of HLT-NAACL*.

Brian Roark and Kristy Hollingshead. 2009. Linear complexity context-free parsing pipelines via chart constraints. In *Proceedings of HLT-NAACL*.

Andreas Stolcke. 1995. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201.

Masaru Tomita. 1988. Graph-structured stack and natural language parsing. In *Proceedings of the 26th annual meeting on Association for Computational Linguistics*, pages 249–257, Morristown, NJ, USA. Association for Computational Linguistics.

Masaru Tomita, editor. 1991. *Generalized LR Parsing*. Kluwer Academic Publishers.

H. Yamada and Y. Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of EMNLP*.