Forest Reranking: Discriminative Parsing with Non-Local Features[*]

Liang Huang

University of Pennsylvania
Philadelphia, PA 19104
lhuang3@cis.upenn.edu

Abstract

Conventional n-best reranking techniques often suffer from the limited scope of the n-best list, which rules out many potentially good alternatives. We instead propose *forest reranking*, a method that reranks a packed forest of exponentially many parses. Since exact inference is intractable with non-local features, we present an approximate algorithm inspired by forest rescoring that makes discriminative training practical over the whole Treebank. Our final result, an F-score of 91.7, outperforms both 50-best and 100-best reranking baselines, and is better than any previously reported systems trained on the Treebank.

1 Introduction

Discriminative reranking has become a popular technique for many NLP problems, in particular, parsing (Collins, 2000) and machine translation (Shen et al., 2005). Typically, this method first generates a list of top-n candidates from a baseline system, and then reranks this n-best list with arbitrary features that are not computable or intractable to compute within the baseline system. But despite its apparent success, there remains a major drawback: this method suffers from the limited scope of the n-best list, which rules out many potentially good alternatives. For example 41% of the correct parses were not in the candidates of $\sim$30-best parses in (Collins, 2000). This situation becomes worse with longer sentences because the number of possible interpretations usually grows exponentially with the

	local	non-local
conventional reranking	only at the root	
DP-based discrim. parsing	exact	N/A
this work: forest-reranking	exact	*on-the-fly*

Table 1: Comparison of various approaches for incorporating local and non-local features.

sentence length. As a result, we often see very few variations among the n-best trees, for example, 50-best trees typically just represent a combination of 5 to 6 binary ambiguities (since $2^5 < 50 < 2^6$).

Alternatively, discriminative parsing is tractable with exact and efficient search based on dynamic programming (DP) if all features are restricted to be *local*, that is, only looking at a local window within the factored search space (Taskar et al., 2004; McDonald et al., 2005). However, we miss the benefits of non-local features that are not representable here.

Ideally, we would wish to combine the merits of both approaches, where an efficient inference algorithm could integrate both local and non-local features. Unfortunately, exact search is intractable (at least in theory) for features with unbounded scope. So we propose *forest reranking*, a technique inspired by forest rescoring (Huang and Chiang, 2007) that approximately reranks the packed forest of exponentially many parses. The key idea is to compute non-local features incrementally from bottom up, so that we can rerank the n-best subtrees at all internal nodes, instead of only at the root node as in conventional reranking (see Table 1). This method can thus be viewed as a step towards the integration of discriminative reranking with traditional chart parsing.

Although previous work on discriminative parsing has mainly focused on short sentences (≤ 15 words) (Taskar et al., 2004; Turian and Melamed, 2007), our work scales to the whole Treebank, where
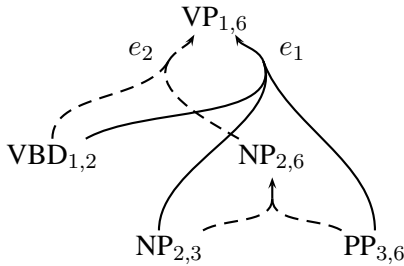
Figure 1: A partial forest of the example sentence.

we achieved an F-score of 91.7, which is a 19% error reduction from the 1-best baseline, and outperforms both 50-best and 100-best reranking. This result is also better than any previously reported systems trained on the Treebank.

2 Packed Forests as Hypergraphs

Informally, a packed parse forest, or *forest* in short, is a compact representation of all the derivations (i.e., parse trees) for a given sentence under a context-free grammar (Billot and Lang, 1989). For example, consider the following sentence

$$_0 \text{ I } _1 \text{ saw } _2 \text{ him } _3 \text{ with } _4 \text{ a } _5 \text{ mirror } _6$$

where the numbers between words denote string positions. Shown in Figure 1, this sentence has (at least) two derivations depending on the attachment of the prep. phrase $PP_{3,6}$ "with a mirror": it can either be attached to the verb "saw",

$$\frac{VBD_{1,2} \quad NP_{2,3} \quad PP_{3,6}}{VP_{1,6}}, \quad (*)$$

or be attached to "him", which will be further combined with the verb to form the same VP as above. These two derivations can be represented as a single forest by sharing common sub-derivations. Such a forest has a structure of a hypergraph (Klein and Manning, 2001; Huang and Chiang, 2005), where items like $PP_{3,6}$ are called *nodes*, and deductive steps like (*) correspond to *hyperedges*.

More formally, a **forest** is a pair $\langle V, E \rangle$, where V is the set of **nodes**, and E the set of **hyperedges**. For a given sentence $w_{1:l} = w_1 \ldots w_l$, each node $v \in V$ is in the form of $X_{i,j}$, which denotes the recognition of nonterminal X spanning the substring from positions i through j (that is, $w_{i+1} \ldots w_j$). Each hyperedge $e \in E$ is a pair $\langle tails(e), head(e) \rangle$, where

$head(e) \in V$ is the consequent node in the deductive step, and $tails(e) \in V^*$ is the list of antecedent nodes. For example, the hyperedge for deduction (*) is notated:

$$e_1 = \langle (VBD_{1,2},\ NP_{2,3},\ PP_{3,6}),\ VP_{1,6} \rangle$$

We also denote $IN(v)$ to be the set of **incoming hyperedges** of node v, which represent the different ways of deriving v. For example, in the forest in Figure 1, $IN(VP_{1,6})$ is $\{e_1, e_2\}$, with $e_2 = \langle (VBD_{1,2},\ NP_{2,6}),\ VP_{1,6} \rangle$. We call $|e|$ the **arity** of hyperedge e, which counts the number of tail nodes in e. The arity of a hypergraph is the maximum arity over all hyperedges. A CKY forest has an arity of 2, since the input grammar is required to be binary branching (cf. Chomsky Normal Form) to ensure cubic time parsing complexity. However, in this work, we use forests from a Treebank parser (Charniak, 2000) whose grammar is often flat in many productions. For example, the arity of the forest in Figure 1 is 3. Such a Treebank-style forest is easier to work with for reranking, since many features can be directly expressed in it. There is also a distinguished **root node** TOP in each forest, denoting the goal item in parsing, which is simply $S_{0,l}$ where S is the start symbol and l is the sentence length.

3 Forest Reranking

3.1 Generic Reranking with the Perceptron

We first establish a unified framework for parse reranking with both n-best lists and packed forests.

For a given sentence s, a generic reranker selects the best parse $\hat{y}$ among the set of candidates $cand(s)$ according to some scoring function:

$$\hat{y} = \underset{y \in cand(s)}{\operatorname{argmax}}\ score(y) \quad (1)$$

In n-best reranking, $cand(s)$ is simply a set of n-best parses from the baseline parser, that is, $cand(s) = \{y_1, y_2, \ldots, y_n\}$. Whereas in forest reranking, $cand(s)$ is a forest implicitly representing the set of exponentially many parses.

As usual, we define the score of a parse y to be the dot product between a high dimensional feature representation and a weight vector $\mathbf{w}$:

$$score(y) = \mathbf{w} \cdot \mathbf{f}(y) \quad (2)$$

where the feature extractor $\mathbf{f}$ is a vector of d functions $\mathbf{f} = (f_1, \ldots, f_d)$, and each feature f_j maps a parse y to a real number $f_j(y)$. Following (Charniak and Johnson, 2005), the first feature $f_1(y) = \log \Pr(y)$ is the log probability of a parse from the baseline generative parser, while the remaining features are all integer valued, and each of them counts the number of times that a particular configuration occurs in parse y. For example, one such feature f_{2000} might be a question

> *"how many times is a* VP *of length 5 surrounded by the word* 'has' *and the period? "*

which is an instance of the **WordEdges** feature (see Figure 2(c) and Section 3.2 for details).

Using a machine learning algorithm, the weight vector $\mathbf{w}$ can be estimated from the training data where each sentence s_i is labelled with its correct ("**gold-standard**") parse y_i^*. As for the learner, Collins (2000) uses the boosting algorithm and Charniak and Johnson (2005) use the maximum entropy estimator. In this work we use the averaged perceptron algorithm (Collins, 2002) since it is an online algorithm much simpler and orders of magnitude faster than Boosting and MaxEnt methods.

Shown in Pseudocode 1, the perceptron algorithm makes several passes over the whole training data, and in each iteration, for each sentence s_i, it tries to predict a best parse $\hat{y}_i$ among the candidates $cand(s_i)$ using the current weight setting. Intuitively, we want the gold parse y_i^* to be picked, but in general it is *not* guaranteed to be within $cand(s_i)$, because the grammar may fail to cover the gold parse, and because the gold parse may be pruned away due to the limited scope of $cand(s_i)$. So we define an **oracle parse** y_i^+ to be the candidate that has the highest Parseval F-score with respect to the gold tree y_i^*:[1]

$$y_i^+ \triangleq \operatorname*{argmax}_{y \in cand(s_i)} F(y, y_i^*) \qquad (3)$$

where function F returns the F-score. Now we train the reranker to pick the oracle parses as often as possible, and in case an error is made (line 6), perform an update on the weight vector (line 7), by adding the difference between two feature representations.

---

[1]If one uses the gold y_i^* for oracle y_i^+, the perceptron will continue to make updates towards something unreachable even when the decoder has picked the best possible candidate.

Pseudocode 1 Perceptron for Generic Reranking

1: **Input**: Training examples $\{cand(s_i), y_i^+\}_{i=1}^N$ $\quad \triangleright y_i^+$ is the oracle tree for s_i among $cand(s_i)$
2: $\mathbf{w} \leftarrow \mathbf{0}$ $\qquad\qquad\qquad\qquad\qquad \triangleright$ initial weights
3: **for** $t \leftarrow 1 \ldots T$ **do** $\qquad\qquad\qquad \triangleright T$ iterations
4: $\quad$ **for** $i \leftarrow 1 \ldots N$ **do**
5: $\qquad \hat{y} = \operatorname{argmax}_{y \in cand(s_i)} \mathbf{w} \cdot \mathbf{f}(y)$
6: $\qquad$ **if** $\hat{y} \neq y_i^+$ **then**
7: $\qquad\qquad \mathbf{w} \leftarrow \mathbf{w} + \mathbf{f}(y_i^+) - \mathbf{f}(\hat{y})$
8: **return** $\mathbf{w}$

In n-best reranking, since all parses are explicitly enumerated, it is trivial to compute the oracle tree.[2] However, it remains widely open how to identify the *forest oracle*. We will present a dynamic programming algorithm for this problem in Sec. 4.1.

We also use a refinement called "averaged parameters" where the final weight vector is the average of weight vectors after each sentence in each iteration over the training data. This averaging effect has been shown to reduce overfitting and produce much more stable results (Collins, 2002).

3.2 Factorizing Local and Non-Local Features

A key difference between n-best and forest reranking is the handling of features. In n-best reranking, all features are treated equivalently by the decoder, which simply computes the value of each one on each candidate parse. However, for forest reranking, since the trees are not explicitly enumerated, many features can not be directly computed. So we first classify features into local and non-local, which the decoder will process in very different fashions.

We define a feature f to be **local** if and only if it can be factored among the local productions in a tree, and **non-local** if otherwise. For example, the **Rule** feature in Fig. 2(a) is local, while the **ParentRule** feature in Fig. 2(b) is non-local. It is worth noting that some features which seem complicated at the first sight are indeed local. For example, the **WordEdges** feature in Fig. 2(c), which classifies a node by its label, span length, and surrounding words, is still local since all these information are encoded either in the node itself or in the input sentence. In contrast, it would become non-local if we replace the surrounding words by surrounding POS

---

[2]In case multiple candidates get the same highest F-score, we choose the parse with the highest log probability from the baseline parser to be the oracle parse (Collins, 2000).

(a) **Rule** (local)
⟨ VP → VBD NP PP ⟩

(b) **ParentRule** (non-local)
⟨ VP → VBD NP PP | S ⟩

(c) **WordEdges** (local)
⟨ NP 5 has . ⟩

(d) **NGramTree** (non-local)
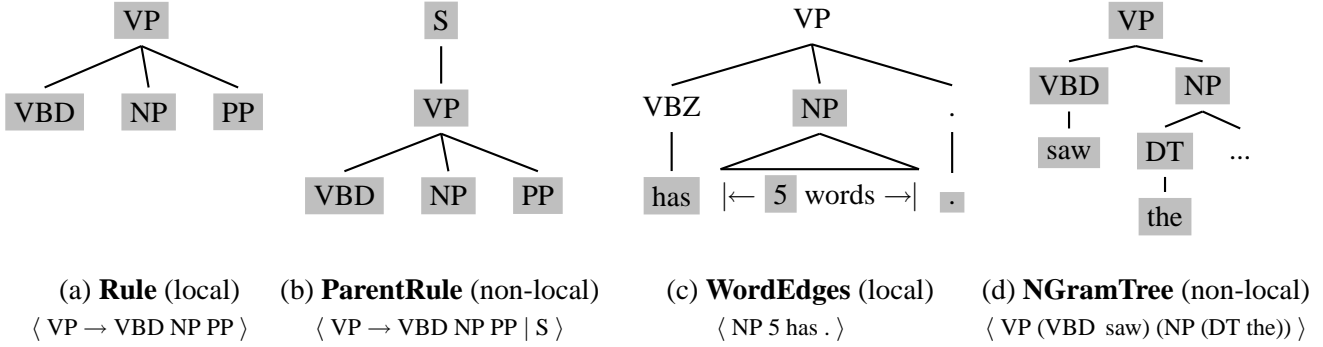⟨ VP (VBD saw) (NP (DT the)) ⟩

Figure 2: Illustration of some example features. Shaded nodes denote information included in the feature.

tags, which are generated dynamically.

More formally, we split the feature extractor $\mathbf{f} = (f_1, \ldots, f_d)$ into $\mathbf{f} = (\mathbf{f}_L; \mathbf{f}_N)$ where $\mathbf{f}_L$ and $\mathbf{f}_N$ are the local and non-local features, respectively. For the former, we extend their domains from parses to hyperedges, where $f(e)$ returns the value of a local feature $f \in \mathbf{f}_L$ on hyperedge e, and its value on a parse y factors across the hyperedges (local productions),

$$\mathbf{f}_L(y) = \sum_{e \in y} \mathbf{f}_L(e) \qquad (4)$$

and we can pre-compute $\mathbf{f}_L(e)$ for each e in a forest.

Non-local features, however, can not be pre-computed, but we still prefer to compute them *as early as possible*, which we call "on-the-fly" computation, so that our decoder can be sensitive to them at internal nodes. For instance, the **NGramTree** feature in Fig. 2 (d) returns the minimum tree fragment spanning a bigram, in this case "saw" and "the", and should thus be computed at the *smallest common ancestor* of the two, which is the VP node in this example. Similarly, the **ParentRule** feature in Fig. 2 (b) can be computed when the S subtree is formed. In doing so, we essentially factor non-local features across *subtrees*, where for each subtree y' in a parse y, we define a **unit feature** $\mathring{f}(y')$ to be the part of $f(y)$ that are computable within y', but not computable in any (proper) subtree of y'. Then we have:

$$\mathbf{f}_N(y) = \sum_{y' \in y} \mathring{\mathbf{f}}_N(y') \qquad (5)$$

Intuitively, we compute the unit non-local features at each subtree from bottom-up. For example, for the binary-branching node $A_{i,k}$ in Fig. 3, the
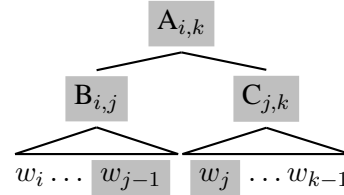

Figure 3: Example of the unit **NGramTree** feature at node $A_{i,k}$: ⟨ A (B $\ldots w_{j-1}$) (C $\ldots w_j$) ⟩.

unit **NGramTree** instance is for the pair $\langle w_{j-1}, w_j \rangle$ on the boundary between the two subtrees, whose smallest common ancestor is the current node. Other unit **NGramTree** instances within this span have already been computed in the subtrees, except those for the boundary words of the whole node, w_i and w_{k-1}, which will be computed when this node is further combined with other nodes in the future.

3.3 Approximate Decoding via Cube Pruning

Before moving on to approximate decoding with non-local features, we first describe the algorithm for exact decoding when only local features are present, where many concepts and notations will be re-used later. We will use $\mathbf{D}(v)$ to denote the top derivations of node v, where $D_1(v)$ is its 1-best derivation. We also use the notation $\langle e, \mathbf{j} \rangle$ to denote the derivation along hyperedge e, using the j_ith sub-derivation for tail u_i, so $\langle e, \mathbf{1} \rangle$ is the best derivation along e. The exact decoding algorithm, shown in Pseudocode 2, is an instance of the bottom-up Viterbi algorithm, which traverses the hypergraph in a topological order, and at each node v, calculates its 1-best derivation using each incoming hyperedge $e \in IN(v)$. The cost of e, $c(e)$, is the score of its

589

Pseudocode 2 Exact Decoding with Local Features

```
1: function VITERBI(⟨V, E⟩)
2:     for v ∈ V in topological order do
3:         for e ∈ IN(v) do
4:             c(e) ← w · f_L(e) + ∑_{u_i ∈ tails(e)} c(D_1(u_i))
5:             if c(e) > c(D_1(v)) then      ▷ better derivation?
6:                 D_1(v) ← ⟨e, 1⟩
7:                 c(D_1(v)) ← c(e)
8:     return D_1(TOP)
```

Pseudocode 3 Cube Pruning for Non-local Features

```
1: function CUBE(⟨V, E⟩)
2:     for v ∈ V in topological order do
3:         KBEST(v)
4:     return D_1(TOP)
5: procedure KBEST(v)
6:     heap ← ∅;  buf ← ∅
7:     for e ∈ IN(v) do
8:         c(⟨e, 1⟩) ← EVAL(e, 1)       ▷ extract unit features
9:         append ⟨e, 1⟩ to heap
10:    HEAPIFY(heap)                    ▷ prioritized frontier
11:    while |heap| > 0 and |buf| < k do
12:        item ← POP-MAX(heap)         ▷ extract next-best
13:        append item to buf
14:        PUSHSUCC(item, heap)
15:    sort buf to D(v)
16: procedure PUSHSUCC(⟨e, j⟩, heap)
17:    e is v → u_1 . . . u_{|e|}
18:    for i in 1 . . . |e| do
19:        j′ ← j + b^i              ▷ b^i is 1 only on the ith dim.
20:        if |D(u_i)| ≥ j′_i then     ▷ enough sub-derivations?
21:            c(⟨e, j′⟩) ← EVAL(e, j′)       ▷ unit features
22:            PUSH(⟨e, j′⟩, heap)
23: function EVAL(e, j)
24:    e is v → u_1 . . . u_{|e|}
25:    return w · f_L(e) + w · f̊_N(⟨e, j⟩) + ∑_i c(D_{j_i}(u_i))
```

(pre-computed) local features $\mathbf{w} \cdot \mathbf{f}_L(e)$. This algorithm has a time complexity of $O(E)$, and is almost identical to traditional chart parsing, except that the forest might be more than binary-branching.

For non-local features, we adapt cube pruning from forest rescoring (Chiang, 2007; Huang and Chiang, 2007), since the situation here is analogous to machine translation decoding with integrated language models: we can view the scores of unit non-local features as the language model cost, computed on-the-fly when combining sub-constituents.

Shown in Pseudocode 3, cube pruning works bottom-up on the forest, keeping a beam of at most k derivations at each node, and uses the k-best parsing Algorithm 2 of Huang and Chiang (2005) to speed up the computation. When combining the sub-

derivations along a hyperedge e to form a new subtree $y' = \langle e, \mathbf{j} \rangle$, we also compute its unit non-local feature values $\mathring{\mathbf{f}}_N(\langle e, \mathbf{j} \rangle)$ (line 25). A priority queue (*heap* in Pseudocode 3) is used to hold the candidates for the next-best derivation, which is initialized to the set of best derivations along each hyperedge (lines 7 to 9). Then at each iteration, we pop the best derivation (lines 12), and push its successors back into the priority queue (line 14). Analogous to the language model cost in forest rescoring, the unit feature cost here is a non-monotonic score in the dynamic programming backbone, and the derivations may thus be extracted *out-of-order*. So a buffer *buf* is used to hold extracted derivations, which is sorted at the end (line 15) to form the list of top-k derivations $\mathbf{D}(v)$ of node v. The complexity of this algorithm is $O(E + Vk \log k\mathcal{N})$ (Huang and Chiang, 2005), where $O(\mathcal{N})$ is the time for on-the-fly feature extraction for each subtree, which becomes the bottleneck in practice.

4 Supporting Forest Algorithms

4.1 Forest Oracle

Recall that the Parseval F-score is the harmonic mean of labelled precision P and labelled recall R:

$$ F(y, y^*) \triangleq \frac{2PR}{P + R} = \frac{2|y \cap y^*|}{|y| + |y^*|} \qquad (6) $$

where $|y|$ and $|y^*|$ are the numbers of brackets in the test parse and gold parse, respectively, and $|y \cap y^*|$ is the number of matched brackets. Since the harmonic mean is a non-linear combination, we can not optimize the F-scores on sub-forests independently with a greedy algorithm. In other words, the optimal F-score tree in a forest is *not* guaranteed to be composed of two optimal F-score subtrees.

We instead propose a dynamic programming algorithm which optimizes the number of matched brackets for a given number of test brackets. For example, our algorithm will ask questions like,

"when a test parse has 5 brackets, what is the maximum number of matched brackets?"

More formally, at each node v, we compute an *oracle function* $ora[v] : \mathbb{N} \mapsto \mathbb{N}$, which maps an integer t to $ora[v](t)$, the max. number of matched brackets

Pseudocode 4 Forest Oracle Algorithm

1: **function** ORACLE($\langle V, E \rangle, y^*$)
2:     **for** $v \in V$ in topological order **do**
3:         **for** $e \in BS(v)$ **do**
4:             e is $v \rightarrow u_1 u_2 \ldots u_{|e|}$
5:             $ora[v] \leftarrow ora[v] \oplus (\otimes_i ora[u_i])$
6:         $ora[v] \leftarrow ora[v] \Uparrow (1, \mathbf{1}_{v \in y^*})$
7:     **return** $F(y^+, y^*) = \max_t \frac{2 \cdot ora[\text{TOP}](t)}{t + |y^*|}$ ▷ oracle F$_1$

for all parses y_v of node v with exactly t brackets:

$$ora[v](t) \triangleq \max_{y_v : |y_v| = t} |y_v \cap y^*| \qquad (7)$$

When node v is combined with another node u along a hyperedge $e = \langle (v, u), w \rangle$, we need to combine the two oracle functions $ora[v]$ and $ora[u]$ by distributing the test brackets of w between v and u, and optimize the number of matched bracktes. To do this we define a *convolution operator* $\otimes$ between two functions f and g:

$$(f \otimes g)(t) \triangleq \max_{t_1 + t_2 = t} f(t_1) + g(t_2) \qquad (8)$$

For instance:

t	$f(t)$		t	$g(t)$		t	$(f \otimes g)(t)$
2	1	$\otimes$	4	4	$=$	6	5
3	2		5	4		7	6
						8	6

The oracle function for the head node w is then

$$ora[w](t) = (ora[v] \otimes ora[u])(t - 1) + \mathbf{1}_{w \in y^*} \quad (9)$$

where $\mathbf{1}$ is the indicator function, returning 1 if node w is found in the gold tree y^*, in which case we increment the number of matched brackets. We can also express Eq. 9 in a purely functional form

$$ora[w] = (ora[v] \otimes ora[u]) \Uparrow (1, \mathbf{1}_{w \in y^*}) \quad (10)$$

where $\Uparrow$ is a *translation operator* which shifts a function along the axes:

$$(f \Uparrow (a, b))(t) \triangleq f(t - a) + b \qquad (11)$$

Above we discussed the case of one hyperedge. If there is another hyperedge e' deriving node w, we also need to combine the resulting oracle functions from both hyperedges, for which we define a *pointwise addition* operator $\oplus$:

$$(f \oplus g)(t) \triangleq \max\{f(t), g(t)\} \qquad (12)$$

Shown in Pseudocode 4, we perform these computations in a bottom-up topological order, and finally at the root node TOP, we can compute the best global F-score by maximizing over different numbers of test brackets (line 7). The oracle tree y^+ can be recursively restored by keeping backpointers for each $ora[v](t)$, which we omit in the pseudocode.

The time complexity of this algorithm for a sentence of l words is $O(|E| \cdot l^{2(a-1)})$ where a is the arity of the forest. For a CKY forest, this amounts to $O(l^3 \cdot l^2) = O(l^5)$, but for general forests like those in our experiments the complexities are much higher. In practice it takes on average 0.05 seconds for forests pruned by $p = 10$ (see Section 4.2), but we can pre-compute and store the oracle for each forest before training starts.

4.2 Forest Pruning

Our forest pruning algorithm (Jonathan Graehl, p.c.) is very similar to the method based on marginal probability (Charniak and Johnson, 2005), except that ours prunes hyperedges as well as nodes. Basically, we use an Inside-Outside algorithm to compute the Viterbi inside cost $\beta(v)$ and the Viterbi outside cost $\alpha(v)$ for each node v, and then compute the **merit** $\alpha\beta(e)$ for each hyperedge:

$$\alpha\beta(e) = \alpha(head(e)) + \sum_{u_i \in tails(e)} \beta(u_i) \qquad (13)$$

Intuitively, this merit is the cost of the best derivation that traverses e, and the difference $\delta(e) = \alpha\beta(e) - \beta(\text{TOP})$ can be seen as the distance away from the globally best derivation. We prune away all hyperedges that have $\delta(e) > p$ for a threshold p. Nodes with all incoming hyperedges pruned are also pruned. The key difference from (Charniak and Johnson, 2005) is that in this algorithm, a node can "partially" survive the beam, with a subset of its hyperedges pruned. In practice, this method prunes on average 15% more hyperedges than their method.

5 Experiments

We compare the performance of our forest reranker against n-best reranking on the Penn English Treebank (Marcus et al., 1993). The baseline parser is the Charniak parser, which we modified to output a

Local	instances	Non-Local	instances
Rule	10,851	**ParentRule**	18,019
Word	20,328	**WProj**	27,417
WordEdges	454,101	**Heads**	70,013
CoLenPar	22	**HeadTree**	67,836
Bigram$^\diamond$	10,292	**Heavy**	1,401
Trigram$^\diamond$	24,677	**NGramTree**	67,559
HeadMod$^\diamond$	12,047	**RightBranch**	2
DistMod$^\diamond$	16,017		
Total Feature Instances: 800,582			

Table 2: Features used in this work. Those with a $^\diamond$ are from (Collins, 2000), and others are from (Charniak and Johnson, 2005), with simplifications.

packed forest for each sentence.[3]

5.1 Data Preparation

We use the standard split of the Treebank: sections 02-21 as the training data (39832 sentences), section 22 as the development set (1700 sentences), and section 23 as the test set (2416 sentences). Following (Charniak and Johnson, 2005), the training set is split into 20 folds, each containing about 1992 sentences, and is parsed by the Charniak parser with a model trained on sentences from the remaining 19 folds. The development set and the test set are parsed with a model trained on all 39832 training sentences.

We implemented both n-best and forest reranking systems in Python and ran our experiments on a 64-bit Dual-Core Intel Xeon with 3.0GHz CPUs. Our feature set is summarized in Table 2, which closely follows Charniak and Johnson (2005), except that we excluded the non-local features **Edges**, **NGram**, and **CoPar**, and simplified **Rule** and **NGramTree** features, since they were too complicated to compute.[4] We also added four *unlexicalized* local features from Collins (2000) to cope with data-sparsity.

Following Charniak and Johnson (2005), we extracted the features from the 50-best parses on the training set (sec. 02-21), and used a cut-off of 5 to prune away low-count features. There are 0.8M features in our final set, considerably fewer than that of Charniak and Johnson which has about 1.3M features

Figure 4: Forests (shown with various pruning thresholds) enjoy higher oracle scores and more compact sizes than n-best lists (on sec 23).

tures in the updated version.[5] However, our initial experiments show that, even with this much simpler feature set, our 50-best reranker performed equally well as theirs (both with an F-score of 91.4, see Tables 3 and 4). This result confirms that our feature set design is appropriate, and the averaged perceptron learner is a reasonable candidate for reranking.

The forests dumped from the Charniak parser are huge in size, so we use the forest pruning algorithm in Section 4.2 to prune them down to a reasonable size. In the following experiments we use a threshold of $p = 10$, which results in forests with an average number of 123.1 hyperedges per forest. Then for each forest, we annotate its forest oracle, and on each hyperedge, pre-compute its local features.[6] Shown in Figure 4, these forests have an forest oracle of 97.8, which is 1.1% higher than the 50-best oracle (96.7), and are 8 times smaller in size.

5.2 Results and Analysis

Table 3 compares the performance of forest reranking against standard n-best reranking. For both systems, we first use only the local features, and then all the features. We use the development set to determine the optimal number of iterations for averaged perceptron, and report the F_1 score on the test set. With only local features, our forest reranker achieves an F-score of 91.25, and with the addition of non-

---

[3]This is a relatively minor change to the Charniak parser, since it implements Algorithm 3 of Huang and Chiang (2005) for efficient enumeration of n-best parses, which requires storing the forest. The modified parser and related scripts for handling forests (e.g. oracles) will be available on my homepage.

[4]In fact, our **Rule** and **ParentRule** features are two special cases of the original **Rule** feature in (Charniak and Johnson, 2005). We also restricted **NGramTree** to be on bigrams only.
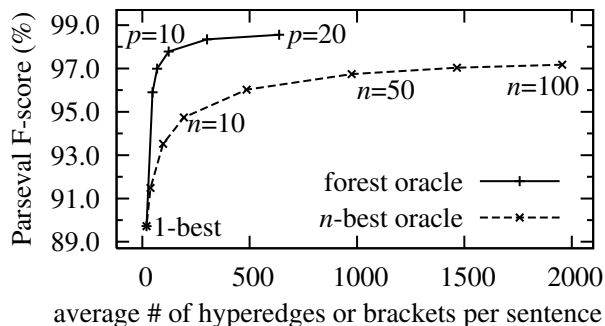
[5]http://www.cog.brown.edu/~mj/software.htm. We follow this version as it corrects some bugs from their 2005 paper which leads to a 0.4% increase in performance (see Table 4).

[6]A subset of local features, e.g. **WordEdges**, is independent of which hyperedge the node takes in a derivation, and can thus be annotated on nodes rather than hyperedges. We call these features *node-local*, which also include part of **Word** features.

baseline: 1-best Charniak parser				89.72
n-best reranking				
features	n	**pre-comp.**	**training**	**F_1%**
local	50	1.7G / 16h	3×0.1h	91.28
all	50	2.4G / 19h	4×0.3h	91.43
all	100	5.3G / 44h	4×0.7h	91.49
forest reranking			($p = 10$)	
features	k	**pre-comp.**	**training**	**F_1%**
local	-	1.2G / 2.9h	3×0.8h	91.25
all	15		4×6.1h	**91.69**

Table 3: Forest reranking compared to n-best reranking on sec. 23. The **pre-comp.** column is for feature extraction, and **training** column shows the number of perceptron iterations that achieved best results on the dev set, and average time per iteration.

local features, the accuracy rises to 91.69 (with beam size $k = 15$), which is a 0.26% absolute improvement over 50-best reranking.[7]

This improvement might look relatively small, but it is much harder to make a similar progress with n-best reranking. For example, even if we double the size of the n-best list to 100, the performance only goes up by 0.06% (Table 3). In fact, the 100-best oracle is only 0.5% higher than the 50-best one (see Fig. 4). In addition, the feature extraction step in 100-best reranking produces huge data files and takes 44 hours in total, though this part can be parallelized.[8] On two CPUs, 100-best reranking takes 25 hours, while our forest-reranker can also finish in 26 hours, with a much smaller disk space. Indeed, this demonstrates the severe redundancies as another disadvantage of n-best lists, where many subtrees are repeated across different parses, while the packed forest reduces space dramatically by sharing common sub-derivations (see Fig. 4).

To put our results in perspective, we also compare them with other best-performing systems in Table 4. Our final result (91.7) is better than any previously reported system trained on the Treebank, although

---

[7]It is surprising that 50-best reranking with local features achieves an even higher F-score of 91.28, and we suspect this is due to the aggressive updates and instability of the perceptron, as we do observe the learning curves to be non-monotonic. We leave the use of more stable learning algorithms to future work.

[8]The n-best feature extraction already uses *relative counts* (Johnson, 2006), which reduced file sizes by at least a factor 4.

type	system	F_1%
D	Collins (2000)	89.7
	Henderson (2004)	90.1
	Charniak and Johnson (2005)	91.0
	updated (Johnson, 2006)	91.4
	this work	**91.7**
G	Bod (2003)	90.7
	Petrov and Klein (2007)	90.1
S	McClosky et al. (2006)	**92.1**

Table 4: Comparison of our final results with other best-performing systems on the whole Section 23. Types D, G, and S denote discriminative, generative, and semi-supervised approaches, respectively.

McClosky et al. (2006) achieved an even higher accuracy (92.1) by leveraging on much larger unlabelled data. Moreover, their technique is orthogonal to ours, and we suspect that replacing their n-best reranker by our forest reranker might get an even better performance. Plus, except for n-best reranking, most discriminative methods require repeated parsing of the training set, which is generally impractical (Petrov and Klein, 2008). Therefore, previous work often resorts to extremely short sentences (≤ 15 words) or only looked at local features (Taskar et al., 2004; Henderson, 2004; Turian and Melamed, 2007). In comparison, thanks to the efficient decoding, our work not only scaled to the whole Treebank, but also successfully incorporated non-local features, which showed an absolute improvement of 0.44% over that of local features alone.

6 Conclusion

We have presented a framework for reranking on packed forests which compactly encodes many more candidates than n-best lists. With efficient approximate decoding, perceptron training on the whole Treebank becomes practical, which can be done in about a day even with a Python implementation. Our final result outperforms both 50-best and 100-best reranking baselines, and is better than any previously reported systems trained on the Treebank. We also devised a dynamic programming algorithm for forest oracles, an interesting problem by itself. We believe this general framework could also be applied to other problems involving forests or lattices, such as sequence labeling and machine translation.

References

Sylvie Billot and Bernard Lang. 1989. The structure of shared forests in ambiguous parsing. In *Proceedings of ACL '89*, pages 143–151.

Rens Bod. 2003. An efficient implementation of a new DOP model. In *Proceedings of EACL*.

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

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

David Chiang. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33(2):201–208.

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*.

James Henderson. 2004. Discriminative training of a neural network statistical parser. In *Proceedings of ACL*.

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

Liang Huang and David Chiang. 2007. Forest rescoring: Fast decoding with integrated language models. In *Proceedings of ACL*.

Mark Johnson. 2006. Features of statistical parsers. Talk given at the *Joint Microsoft Research and Univ. of Washington Computational Linguistics Colloquium*. http://www.cog.brown.edu/~mj/papers/ms-uw06talk.pdf.

Dan Klein and Christopher D. Manning. 2001. Parsing and Hypergraphs. In *Proceedings of the Seventh International Workshop on Parsing Technologies (IWPT-2001), 17-19 October 2001, Beijing, China*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19:313–330.

David McClosky, Eugene Charniak, and Mark Johnson. 2006. Effective self-training for parsing. In *Proceedings of the HLT-NAACL*, New York City, USA, June.

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

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

Slav Petrov and Dan Klein. 2008. Discriminative log-linear grammars with latent variables. In *Proceedings of NIPS 20*.

Libin Shen, Anoop Sarkar, and Franz Josef Och. 2005. Discriminative reranking for machine translation. In *Proceedings of HLT-NAACL*.

Ben Taskar, Dan Klein, Michael Collins, Daphne Koller, and Chris Manning. 2004. Max-margin parsing. In *Proceedings of EMNLP*.

Joseph Turian and I. Dan Melamed. 2007. Scalable discriminative learning for natural language parsing and translation. In *Proceedings of NIPS 19*.