Weight pushing and binarization for fixed-grammar parsing

Matt Post and **Daniel Gildea**
Department of Computer Science
University of Rochester
Rochester, NY 14627

Abstract

We apply the idea of *weight pushing* (Mohri, 1997) to CKY parsing with fixed context-free grammars. Applied after rule binarization, weight pushing takes the weight from the original grammar rule and pushes it down across its binarized pieces, allowing the parser to make better pruning decisions earlier in the parsing process. This process can be viewed as generalizing weight pushing from transducers to hypergraphs. We examine its effect on parsing efficiency with various binarization schemes applied to tree substitution grammars from previous work. We find that weight pushing produces dramatic improvements in efficiency, especially with small amounts of time and with large grammars.

1 Introduction

Fixed grammar-parsing refers to parsing that employs grammars comprising a finite set of rules that is fixed before inference time. This is in contrast to markovized grammars (Collins, 1999; Charniak, 2000), variants of tree-adjoining grammars (Chiang, 2000), or grammars with wildcard rules (Bod, 2001), all of which allow the construction and use of rules not seen in the training data. Fixed grammars must be binarized (either explicitly or implicitly) in order to maintain the $\mathcal{O}(n^3|G|)$ (n the sentence length, $|G|$ the grammar size) complexity of algorithms such as the CKY algorithm.

Recently, Song et al. (2008) explored different methods of binarization of a PCFG read directly from the Penn Treebank (the Treebank PCFG), showing that binarization has a significant effect on both the number of rules and new nonterminals introduced, and subsequently on parsing time. This variation occurs because different binarization schemes produce different amounts of shared rules, which are rules produced during the binarization process from more than one rule in the original grammar. Increasing sharing reduces the amount of state that the parser must explore. Binarization has also been investigated in the context of parsing-based approaches to machine translation, where it has been shown that paying careful attention to the binarization scheme can produce much faster decoders (Zhang et al., 2006; Huang, 2007; DeNero et al., 2009).

The choice of binarization scheme will not affect parsing results if the parser is permitted to explore the whole search space. In practice, however, this space is too large, so parsers use pruning to discard unlikely hypotheses. This presents a problem for bottom-up parsing algorithms because of the way the probability of a rule is distributed among its binarized pieces: The standard approach is to place all of that probability on the top-level binarized rule, and to set the probabilities of lower binarized pieces to 1.0. Because these rules are reconstructed from the bottom up, pruning procedures do not have a good estimate of the complete cost of a rule until the entire original rule has been reconstructed. It is preferable to have this information earlier on, especially for larger rules.

In this paper we adapt the technique of *weight pushing* for finite state transducers (Mohri, 1997) to arbitrary binarizations of context-free grammar rules. Weight pushing takes the probability (or, more generally, the weight) of a rule in the original grammar and pushes it down across the rule's binarized pieces. This helps the parser make bet-

Proceedings of the 11th International Conference on Parsing Technologies (IWPT), pages 89–98,
Paris, October 2009.

ter pruning decisions, and to make them earlier in the bottom-up parsing process. We investigate this algorithm with different binarization schemes and grammars, and find that it improves the time vs. accuracy tradeoff for parsers roughly proportionally to the size of the grammar being binarized.

This paper extends the work of Song et al. (2008) in three ways. First, weight pushing further reduces the amount of time required for parsing. Second, we apply these techniques to Tree Substitution Grammars (TSGs) learned from the Treebank, which are both larger and more accurate than the context-free grammar read directly from the Treebank.[1] Third, we examine the interaction between binarization schemes and the inexact search heuristic of beam-based and k-best pruning.

2 Weight pushing

2.1 Binarization

Not all binarization schemes are equivalent in terms of efficiency of representation. Consider the grammar in the lefthand column of Figure 1 (rules 1 and 2). If this grammar is right-binarized or left-binarized, it will produce seven rules, whereas the optimal binarization (depicted) produces only 5 rules due to the fact that two of them are shared. Since the complexity of parsing with CKY is a function of the grammar size as well as the input sentence length, and since in practice parsing requires significant pruning, having a smaller grammar with maximal shared substructure among the rules is desirable.

We investigate two kinds of binarization in this paper. The first is right binarization, in which nonterminal pairs are collapsed beginning from the two rightmost children and moving leftward. The second is a greedy binarization, similar to that of Schmid (2004), in which the most frequently occurring (grammar-wide) nonterminal pair is collapsed in turn, according to the algorithm given in Figure 2.

Binarization must ensure that the product of the probabilities of the binarized pieces is the same as that of the original rule. The easiest way to do this is to assign each newly-created binarized rule a probability of 1.0, and give the top-level rule the complete probability of the original rule. In the following subsection, we describe a better way.

[1]The mean rule rank in a Treebank PCFG is 2.14, while the mean rank in our sampled TSG is 8.51. See Table 1.

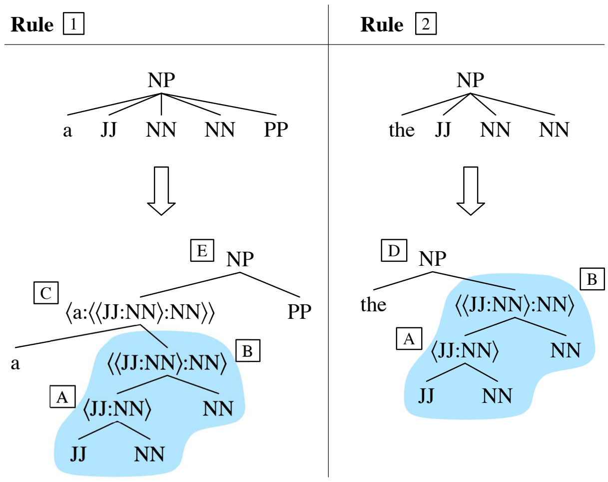


Figure 1: A two-rule grammar. The greedy binarization algorithm produces the binarization shown, with the shared structure highlighted. Binarized rules A, B, and C are initially assigned a probability of 1.0, while rules D and E are assigned the original probabilities of rules 2 and 1, respectively.

2.2 Weight pushing

Spreading the weight of an original rule across its binarized pieces is complicated by sharing, because of the constraint that the probability of shared binarized pieces must be set so that the product of their probabilities is the same as the original rule, for each rule the shared piece participates in. Mohri (1997) introduced *weight pushing* as a step in the minimization of weighted finite-state transducers (FSTs), which addressed a similar problem for tasks employing finite-state machinery. At a high level, weight pushing moves the weight of a path towards the initial state, subject to the constraint that the weight of each path in the FST is unchanged. To do weight pushing, one first computes for each state q in the transducer the shortest distance $d(q)$ to any final state. Let $\sigma(q, a)$ be the state transition function, deterministically transitioning on input a from state q to state $\sigma(q, a)$. Pushing adjusts the weight of each edge $w(e)$ according to the following formula:

$$w'(e) = d(q)^{-1} \times w(e) \times d(\sigma(q, a)) \quad (1)$$

Mohri (1997, §3.7) and Mohri and Riley (2001) discuss how these operations can be applied using various semirings; in this paper we use the $(\max, \times)$ semiring. The important observation for our purposes is that pushing can be thought of as a sequence of local operations on individual nodes

1: **function** GREEDYBINARIZE(P)
2: **while** RANK(P) > 2 **do**
3: κ := UPDATECOUNTS(P)
4: **for** each rule $X \rightarrow x_1 x_2 \cdots x_r$ **do**
5: $b := \text{argmax}_{i \in (2 \cdots r)} \kappa[x_{i-1}, x_i]$
6: $l := \langle x_{b-1} : x_b \rangle$
7: add $l \rightarrow x_{b-1} x_b$ to P
8: replace $x_{b-1} x_b$ with l in rule
9: **function** UPDATECOUNTS(P)
10: $\kappa := \{\}$ ▷ a dictionary
11: **for** each rule $X \rightarrow x_1 x_2 \cdots x_r \in P$ **do**
12: **for** $i \in (2 \cdots r)$ **do**
13: $\kappa[x_{i-1}, x_i]$++
return κ

Figure 2: A greedy binarization algorithm. The rank of a grammar is the rank of its largest rule. Our implementation updates the counts in κ more efficiently, but we present it this way for clarity.

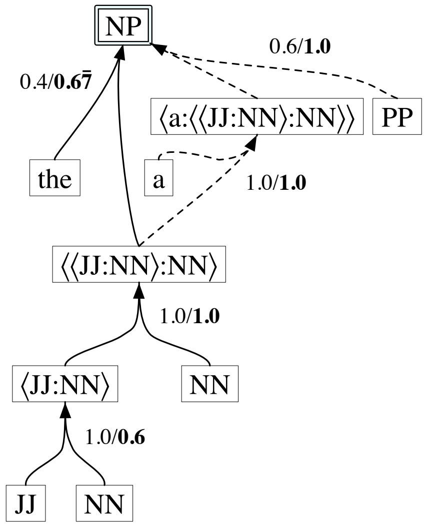


Figure 3: The binarized rules of Figure 1 arranged in a shared hypergraph forest. Each hyperedge is labeled with its weight before/**after** pushing.

q, shifting a constant amount of weight $d(q)^{-1}$ from q's outgoing edges to its incoming edges.

Klein and Manning (2003) describe an encoding of context-free grammar rule binarization that permits weight pushing to be applied. Their approach, however, works only with left or right binarizations whose rules can be encoded as an FST. We propose a form of weight pushing that works for arbitrary binarizations. Weight pushing across a grammar can be viewed as generalizing pushing from weighted transducers to a certain kind of weighted hypergraph. To begin, we use the following definition of a hypergraph:

Definition. A *hypergraph* H is a tuple $\langle V, E, F, R \rangle$, where V is a set of nodes, E is a set of hyperedges, $F \subset V$ is a set of final nodes, and R is a set of permissible weights on the hyperedges. Each hyperedge $e \in E$ is a triple $\langle T(e), h(e), w(e) \rangle$, where $h(e) \in V$ is its head node, $T(e)$ is a sequence of tail nodes, and $w(e)$ is its weight.

We can arrange the binarized rules of Figure 1 into a shared hypergraph forest (Figure 3), with nodes as nonterminals and binarized rules as hyperedges. We distinguish between final and nonfinal nodes and hyperedges. Nonfinal nodes are those in $V - F$. Nonfinal hyperdges E_{NF} are those in $\{e : h(e) \in V - F\}$, that is, all hyperedges whose head is a nonfinal node. Because all nodes introduced by our binarization procedure expand deterministically, each nonfinal node is the head of no more than one such hyperedge. Initially, all nonfinal hyperedges have a probability of 1, and final hyperedges have a probability equal to the that of the original unbinarized rule. Each path through the forest exactly identifies a binarization of a rule in the original grammar, and hyperpaths overlap where binarized rules are shared.

Weight pushing in this hypergraph is similar to weight pushing in a transducer. We consider each nonfinal node v in the graph and execute a local operation that moves weight in some way from the set of edges $\{e : v \in T(e)\}$ (v's outgoing hyperedges) to the edge e_h for which $v = h(e)$ (v's incoming hyperedge).

A critical difference from pushing in transducers is that a node in a hyperpath may be used more than once. Consider adding the rule NP→JJ NN JJ NN to the binarized two-rule grammar we have been considering. Greedy binarization could[2] binarize it in the following manner

$$\text{NP} \rightarrow \langle\text{JJ:NN}\rangle \ \langle\text{JJ:NN}\rangle$$
$$\langle\text{JJ:NN}\rangle \rightarrow \text{JJ NN}$$

which would yield the hypergraph in Figure 4. In order to maintain hyperpath weights, a pushing procedure at the ⟨JJ:NN⟩ node must pay attention to the number of times it appears in the set of tail nodes of each outgoing hyperedge.

[2]Depending on the order in which the argmax variable i of Line 5 from the algorithm in Figure 2 is considered. This particular binarization would not have been produced if the values $2 \ldots r$ were tested sequentially.

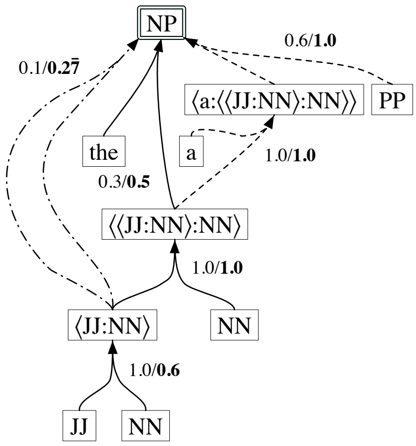


Figure 4: A hypergraph containing a hyperpath representing a rule using the same binarized piece twice. Hyperedge weights are again shown before/**after** pushing.

1: **function** DIFFUSEWEIGHTS(P_{BIN}, Π)
2: R := bottom-up sort of P_{BIN}
3: **for** each rule $r \in R$ **do**
4: $r.\mathrm{pr} := \max\{ \sqrt[c(r,p)]{p.\mathrm{pr}} : p \in \Pi(r)\}$
5: **for** each rule $p \in \Pi(r)$ **do**
6: $p.\mathrm{pr} := p.\mathrm{pr}/r.\mathrm{pr}^{c(r,p)}$

Figure 6: Maximal weight pushing algorithm applied to a binarized grammar, P_{BIN}. Π is a dictionary mapping from an internal binary rule to a list of top-level binary rules that it appeared under.

With these similarities and differences in mind, we can define the local weight pushing procedure. For each nonfinal node v in the hypergraph, we define e_h as the edge for which $h(e) = v$ (as before), $P = \{e : v \in T(e)\}$ (the set of outgoing hyperedges), and $c(v, T(e))$ as the number of times v appears in the sequence of tail nodes $T(e)$. The minimum amount of probability available for pushing is then

$$\max\{ \sqrt[c(v,T(e))]{w(e)} : e \in P\} \quad (2)$$

This amount can then be multiplied into $w(e_h)$ and divided out of each edge $e \in P$. This max is a lower bound because we have to ensure that the amount of probability we divide out of the weight of each outgoing hyperedge is *at least as large as* that of the maximum weight.

While finite state transducers each have a unique equivalent transducer on which no further pushing is possible, defined by Equation 1, this is not the case when operating on hypergraphs. In this generalized setting, the choice of which tail nodes to push weight across can result in different final solutions. We must define a strategy for choosing among sequences of pushing operations, and for this we now turn to a discussion of the specifics of our algorithm.

2.3 Algorithm

We present two variants. *Maximal* pushing, analogous to weight pushing in weighted FSTs, pushes the original rule's weight down as far as possible. Analysis of interactions between pruning and maximal pushing discovered situations where maximal pushing resulted in search error (see §4.2). To address this, we also discuss *nthroot* pushing, which attempts to distribute the weight more evenly across its pieces, by taking advantage of the fact that Equation 2 is a lower bound on the amount of probability available for pushing.

The algorithm for maximal pushing is listed in Figure 6, and works in the following manner. When binarizing we maintain, for each binarized piece, a list of all the original rules that share it. We then distribute that original rule's weight by considering each of these binarized pieces in bottom-up topological order and setting the probability of the piece to the maximum (remaining) probability of these parents. This amount is then divided out of each of the parents, and the process continues. See Figure 5 for a depiction of this process. Note that, although we defined pushing as a local operation between adjacent hyperedges, it is safe to move probability mass from the top-level directly to the bottom (as we do here). Intuitively, we can imagine this as a series of local pushing operations on all intervening nodes; the end result is the same.

For nthroot pushing, we need to maintain a dictionary δ which records, for each binary piece, the rank (number of items on the rule's righthand side) of the original rule it came from. This is accomplished by replacing line 4 in Figure 6 with

$$r.\mathrm{pr} := \max\{ \sqrt[(\delta(p)-1)\cdot c(r,p)]{p.\mathrm{pr}} : p \in \Pi(r)\}$$

Applying weight pushing to a binarized PCFG results in a grammar that is not a PCFG, because rule probabilities for each lefthand side no longer sum to one. However, the tree distribution, as well as the conditional distribution P(tree|string) (which are what matter for parsing) are unchanged. To show this, we argue from the algorithm in Figure 6, demonstrating that, for

step	A	B	C	D	E
0	1.0	1.0	1.0	x	y
1	$\max(x, y)$	·	·	$\frac{x}{\max(x,y)}$	$\frac{y}{\max(x,y)}$
2	·	$\max(z_{1,D}, z_{1,E})$	·	$\frac{z_{1,D}}{\max(z_{1,D},z_{1,E})}$	$\frac{z_{1,D}}{\max(z_{1,D},z_{1,E})}$
3	·	·	$\max(z_{2,D}, z_{2,E})$	$\frac{z_{2,D}}{\max(z_{2,D},z_{2,E})}$	$\frac{z_{2,E}}{\max(z_{2,D},z_{2,E})}$
4	·	·	·	·	·

Figure 5: Stepping through the maximal weight pushing algorithm for the binarized grammar in Figure 1. Rule labels A through E were chosen so that the binarized pieces are sorted in topological order. A (·) indicates a rule whose value has not changed from the previous step, and the value $z_{r,c}$ denotes the value in row r column c.

each rule in the original grammar, its probability is equal to the product of the probabilities of its pieces in the binarized grammar. This invariant holds at the start of the algorithm (because the probability of each original rule was placed entirely at the top-level rule, and all other pieces received a probability of 1.0) and is also true at the end of each iteration of the outer loop. Consider this loop. Each iteration considers a single binary piece (line 3), determines the amount of probability to claim from the parents that share it (line 4), and then removes this amount of weight from each of its parents (lines 5 and 6). There are two important considerations.

1. A binarized rule piece may be used more than once in the reconstruction of an original rule; this is important because we are assigning probabilities to binarized rule *types*, but rule reconstruction makes use of binarized rule *tokens*.

2. Multiplying together two probabilities results in a lower number: when we shift weight p from the parent rule to (n instances of) a binarized piece beneath it, we are creating a new set of probabilities p_c and p_p such that $p_c^n \cdot p_p = p$, where p_c is the weight placed on the binarized rule type, and p_p is the weight we leave at the parent. This means that we must choose p_c from the range $[p, 1.0]$.[3]

In light of these considerations, the weight removed from each parent rule in line 6 must be greater than or equal to each parent sharing the binarized rule piece. To ensure this, line 4 takes the maximum of the $c(r, p)$th root of each parent's probability, where $c(r, p)$ is the number of times binarized rule token r appears in the binarization of p.

Line 4 breaks the invariant, but line 6 restores it for each parent rule the current piece takes part in. From this it can be seen that weight pushing does not change the product of the probabilities of the binarized pieces for each rule in the grammar, and hence the tree distribution is also unchanged.

We note that, although Figures 3 and 4 show only one final node, any number of final nodes can appear if binarized pieces are shared across different top-level nonterminals (which our implementation permits and which does indeed occur).

3 Experimental setup

We present results from four different grammars:

1. The standard Treebank probabilistic context-free grammar (PCFG).

2. A "spinal" tree substitution grammar (TSG), produced by extracting n lexicalized subtrees from each length n sentence in the training data. Each subtree is defined as the sequence of CFG rules from leaf upward all sharing the same lexical head, according to the Magerman head-selection rules (Collins, 1999). We detach the top-level unary rule, and add in counts from the Treebank CFG rules.

3. A "minimal subset" TSG, extracted and then refined according to the process defined in Bod (2001). For each height $h, 2 \leq h \leq 14$, 400,000 subtrees are randomly sampled from the trees in the training data, and the counts

[3]The upper bound of 1.0 is set to avoid assigning a negative weight to a rule.

grammar	# rules	rank		
		median	mean	max
PCFG	46K	1	2.14	51
spinal	190K	3	3.36	51
sampled	804K	8	8.51	70
minimal	2,566K	10	10.22	62

Table 1: Grammar statistics. A rule's rank is the number of symbols on its right-hand side.

grammar	unbinarized	right	greedy
PCFG	46K	56K	51K
spinal	190K	309K	235K
sampled	804K	3,296K	1,894K
minimal	2,566K	15,282K	7,981K

Table 2: Number of rules in each of the complete grammars before and after binarization.

are summed. From these counts we remove (a) all unlexicalized subtrees of height greater than six and (b) all lexicalized subtrees containing more than twelve terminals on their frontier, and we add all subtrees of height one (i.e., the Treebank PCFG).

4. A sampled TSG produced by inducing derivations on the training data using a Dirichlet Process prior (described below).

The sampled TSG was produced by inducing a TSG derivation on each of the trees in the training data, from which subtree counts were read directly. These derivations were induced using a collapsed Gibbs sampler, which sampled from the posterior of a Dirichlet process (DP) defined over the subtree rewrites of each nonterminal. The DP describes a generative process that prefers small subtrees but occasionally produces larger ones; when used for inference, it essentially discovers TSG derivations that contain larger subtrees only if they are frequent in the training data, which discourages model overfitting. See Post and Gildea (2009) for more detail. We ran the sampler for 100 iterations with a stop probability of 0.7 and the DP parameter $\alpha = 100$, accumulating subtree counts from the derivation state at the end of all the iterations, which corresponds to the $(100, 0.7, \leq 100)$ grammar from that paper.

All four grammar were learned from all sentences in sections 2 to 21 of the Wall Street Journal portion of the Penn Treebank. All trees were preprocessed to remove empty nodes and nonterminal annotations. Punctuation was retained. Statistics for these grammars can be found in Table 1. We present results on sentences with no more than forty words from section 23.

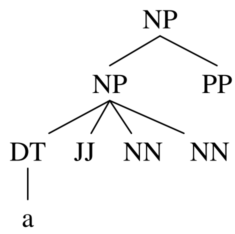


Figure 7: Rule [1] in Figure 1 was produced by flattening this rule from the sampled grammar.

Our parser is a Perl implementation of the CKY algorithm.[4] For the larger grammars, memory limitations require us to remove from consideration all grammar rules that could not possibly take part in a parse of the current sentence, which we do by matching the rule's frontier lexicalization pattern against the words in the sentence. All unlexicalized rules are kept. This preprocessing time is not included in the parsing times reported in the next section.

For pruning, we group edges into equivalence classes according to the following features:

- span (s, t) of the input
- level of binarization (0,1,2+)

The level of binarization refers to the height of a nonterminal in the subtree created by binarizing a CFG rule (with the exception that the root of this tree has a binarization level of 0). The naming scheme used to create new nonterminals in line 6 of Figure 2 means we can determine this level by counting the number of left-angle brackets in the nonterminal's name. In Figure 1, binarized rules D and E have level 0, C has level 3, B has level 2, and A has level 1.

Within each bin, only the β highest-weight items are kept, where $\beta \in (1, 5, 10, 25, 50)$ is a parameter that we vary during our experiments. Ties are broken arbitrarily. Additionally, we maintain a beam within each bin, and an edge is pruned if its score is not within a factor of 10^{-5} of the highest-scoring edge in the bin. Pruning takes place when the edge is added and then again at the end of each

[4] It is available from `http://www.cs.rochester.edu/~post/`.

span in the CKY algorithm (but before applying unary rules).

In order to binarize TSG subtrees, we follow Bod (2001) in first flattening each subtree to a depth-one PCFG rule that shares the subtree's root nonterminal and leaves, as depicted in Figure 7. Afterward, this transformation is reversed to produce the parse tree for scoring. If multiple TSG subtrees have identical mappings, we take only the most probable one. Table 2 shows how grammar size is affected by binarization scheme.

We note two differences in our work that explain the large difference between the scores reported for the "minimal subset" grammar in Bod (2001) and here. First, we did not implement the smoothed "mismatch parsing", which introduces new subtrees into the grammar at parsing time by allowing lexical leaves of subtrees to act as wildcards. This technique reportedly makes a large difference in parsing scores (Bod, 2009). Second, we approximate the most probable parse with the single most probable derivation instead of the top 1,000 derivations, which Bod also reports as having a large impact (Bod, 2003, §4.2).

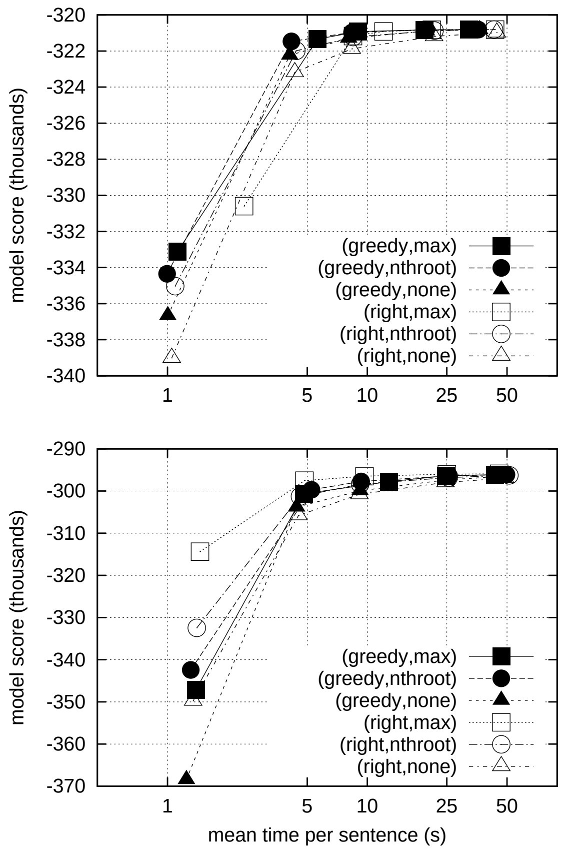


Figure 8: Time vs. model score for the PCFG (top) and the sampled grammar (bottom). Note that the y-axis range differs between plots.

4 Results

Figure 8 displays search time vs. model score for the PCFG and the sampled grammar. Weight pushing has a significant impact on search efficiency, particularly for the larger sampled grammar. The spinal and minimal graphs are similar to the PCFG and sampled graphs, respectively, which suggests that the technique is more effective for the larger grammars.

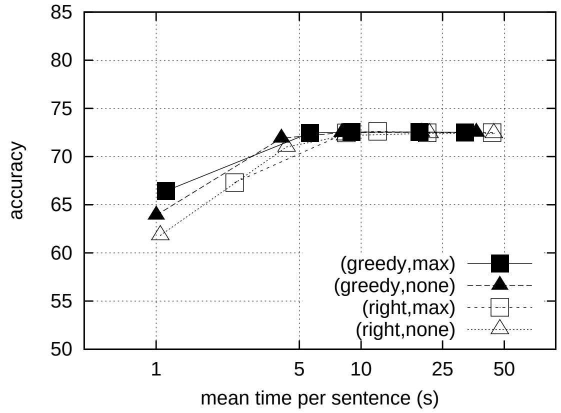
For parsing, we are ultimately interested in accuracy as measured by F_1 score.[5] Figure 9 displays graphs of time vs. accuracy for parses with each of the grammars, alongside the numerical scores used to generate them. We begin by noting that the improved search efficiency from Figure 8 carries over to the time vs. accuracy curves for the PCFG and sampled grammars, as we expect. Once again, we note that the difference is less pronounced for the two smaller grammars than for the two larger ones.

4.1 Model score vs. accuracy

The tables in Figure 9 show that parser accuracy is not always a monotonic function of time; some of the runs exhibited peak performance as early as at a bin size of $\beta = 10$, and then saw drops in scores when given more time. We examined a number of instances where the F_1 score for a sentence was lower at a higher bin setting, and found that they can be explained as modeling (as opposed to search) errors. With the PCFG, these errors were standard parser difficulties, such as PP attachment, which require more context to resolve. TSG subtrees, which have more context, are able to correct some of these issues, but introduce a different set of problems. In many situations, larger bin settings permitted erroneous analyses to remain in the chart, which later led to the parser's discovery of a large TSG fragment. Because these fragments often explain a significant portion of the sentence more cheaply than multiple smaller rules multiplied together, the parser prefers them. More often than not, they are useful, but sometimes they are overfit to the training data, and result in an incorrect analysis despite a higher model score.

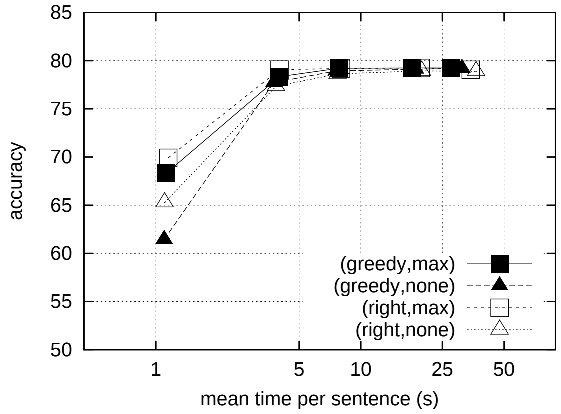
Interestingly, these dips occur most frequently for the heuristically extracted TSGs (four of six

[5] $F_1 = \frac{2 \cdot P \cdot R}{P+R}$, where P is precision and R recall.



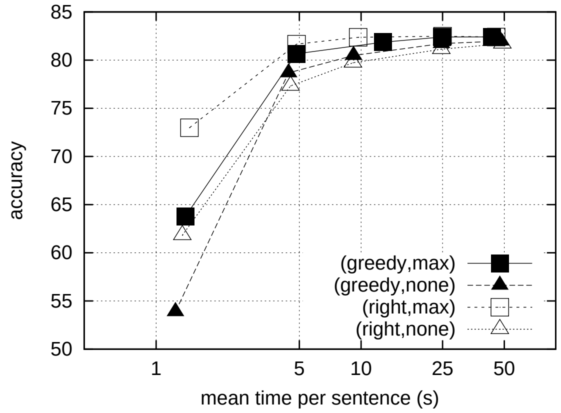
PCFG

run	1	5	10	25	50
■ (g,m)	66.44	72.45	72.54	72.54	72.51
● (g,n)	65.44	72.21	72.47	72.45	72.47
▲ (g,-)	63.91	71.91	72.48	72.51	72.51
□ (r,m)	67.30	72.45	72.61	72.47	72.49
○ (r,n)	64.09	71.78	72.33	72.45	72.47
△ (r,-)	61.82	71.00	72.18	72.42	72.41



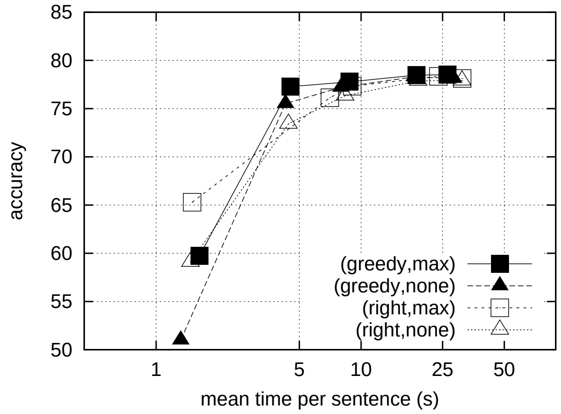
spinal

run	1	5	10	25	50
■ (g,m)	68.33	78.35	79.21	79.25	79.24
● (g,n)	64.67	78.46	79.04	79.07	79.09
▲ (g,-)	61.44	77.73	78.94	79.11	79.20
□ (r,m)	69.92	79.07	79.18	79.25	79.05
○ (r,n)	67.76	78.46	79.07	79.04	79.04
△ (r,-)	65.27	77.34	78.64	78.94	78.90



sampled

run	1	5	10	25	50
■ (g,m)	63.75	80.65	81.86	82.40	82.41
● (g,n)	61.87	79.88	81.35	82.10	82.17
▲ (g,-)	53.88	78.68	80.48	81.72	81.98
□ (r,m)	72.98	81.66	82.37	82.49	82.40
○ (r,n)	65.53	79.01	80.81	81.91	82.13
△ (r,-)	61.82	77.33	79.72	81.13	81.70



minimal

run	1	5	10	25	50
■ (g,m)	59.75	77.28	77.77	78.47	78.52
● (g,n)	57.54	77.12	77.82	78.35	78.36
▲ (g,-)	51.00	75.52	77.21	78.30	78.13
□ (r,m)	65.29	76.14	77.33	78.34	78.13
○ (r,n)	61.63	75.08	76.80	77.97	78.31
△ (r,-)	59.10	73.42	76.34	77.88	77.91

Figure 9: Plots of parsing time vs. accuracy for each of the grammars. Each plot contains four sets of five points ($\beta \in (1, 5, 10, 25, 50)$), varying the binarization strategy (right (r) or greedy (g)) and the weight pushing technique (maximal (m) or none (-)). The tables also include data from nthroot (n) pushing.

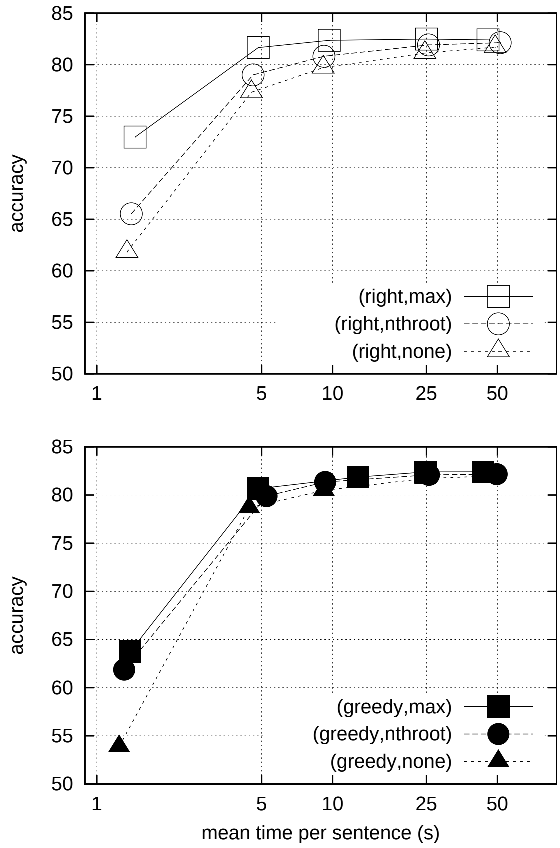


Figure 10: Time vs. accuracy (F_1) for the sampled grammar, broken down by binarization (right on top, greedy on bottom).

runs for the spinal grammar, and two for the minimal grammar) and for the PCFG (four), and least often for the model-based sampled grammar (just once). This may suggest that rules selected by our sampling procedure are less prone to overfitting on the training data.

4.2 Pushing

Figure 10 compares the nthroot and maximal pushing techniques for both binarizations of the sampled grammar. We can see from this figure that there is little difference between the two techniques for the greedy binarization and a large difference for the right binarization. Our original motivation in developing nthroot pushing came as a result of analysis of certain sentences where maximal pushing and greedy binarization resulted in the parser producing a lower model score than with right binarization with no pushing. One such example was binarized fragment A from Figure 1; when parsing a particular sentence in the development set, the correct analysis required the rule from Figure 7, but greedy binarization and maximal pushing resulted in this piece getting pruned early in the search procedure. This pruning happened because maximal pushing allowed too much weight to shift down for binarized pieces of competing analyses relative to the correct analysis. Using nthroot pushing solved the search problem in that instance, but in the aggregate it does not appear to be helpful in improving parser efficiency as much as maximal pushing. This demonstrates some of the subtle interactions between binarization and weight pushing when inexact pruning heuristics are applied.

4.3 Binarization

Song et al. (2008, Table 4) showed that CKY parsing efficiency is not a monotonic function of the number of constituents produced; that is, enumerating fewer edges in the dynamic programming chart does not always correspond with shorter run times. We see here that efficiency does not always perfectly correlate with grammar size, either. For all but the PCFG, right binarization improves upon greedy binarization, regardless of the pushing technique, despite the fact that the right-binarized grammars are always larger than the greedily-binarized ones.

Weight pushing and greedy binarization both increase parsing efficiency, and the graphs in Figures 8 and 9 suggest that they are somewhat complementary. We also investigated left binarization, but discontinued that exploration because the results were nearly identical to that of right binarization. Another popular binarization approach is head-outward binarization. Based on the analysis above, we suspect that its performance will fall somewhere among the binarizations presented here, and that pushing will improve it as well. We hope to investigate this in future work.

5 Summary

Weight pushing increases parser efficiency, especially for large grammars. Most notably, it improves parser efficiency for the Gibbs-sampled tree substitution grammar of Post and Gildea (2009).

We believe this approach could alo benefit syntax-based machine translation. Zhang et al. (2006) introduced a synchronous binarization technique that improved decoding efficiency and accuracy by ensuring that rule binarization avoided gaps on both the source and target sides (for rules where this was possible). Their binarization was designed to share binarized pieces among rules, but their approach to distributing weight was the default (nondiffused) case found in this paper to be least efficient: The entire weight of the original rule is placed at the top binarized rule and all internal rules are assigned a probability of 1.0.

Finally, we note that the weight pushing algorithm described in this paper began with a PCFG and ensured that the tree distribution was not changed. However, weight pushing need not be limited to a probabilistic interpretation, but could be used to spread weights for grammars with discriminatively trained features as well, with necessary adjustments to deal with positively and negatively weighted rules.

Acknowledgments We thank the anonymous reviewers for their helpful comments. This work was supported by NSF grants IIS-0546554 and ITR-0428020.

References

Rens Bod. 2001. What is the minimal set of fragments that achieves maximal parse accuracy? In *Proceedings of the 39th Annual Conference of the Association for Computational Linguistics (ACL-01)*, Toulouse, France.

Rens Bod. 2003. Do all fragments count? *Natural Language Engineering*, 9(4):307–323.

Rens Bod. 2009. Personal communication.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proceedings of the 2000 Meeting of the North American chapter of the Association for Computational Linguistics (NAACL-00)*, Seattle, Washington.

David Chiang. 2000. Statistical parsing with an automatically-extracted tree adjoining grammar. In *Proceedings of the 38th Annual Conference of the Association for Computational Linguistics (ACL-00)*, Hong Kong.

Michael John Collins. 1999. *Head-driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania, Philadelphia.

John DeNero, Mohit Bansal, Adam Pauls, and Dan Klein. 2009. Efficient parsing for transducer grammars. In *Proceedings of the 2009 Meeting of the North American chapter of the Association for Computational Linguistics (NAACL-09)*, Boulder, Colorado.

Liang Huang. 2007. Binarization, synchronous binarization, and target-side binarization. In *North American chapter of the Association for Computational Linguistics Workshop on Syntax and Structure in Statistical Translation (NAACL-SSST-07)*, Rochester, NY.

Dan Klein and Christopher D. Manning. 2003. A* parsing: Fast exact Viterbi parse selection. In *Proceedings of the 2003 Meeting of the North American chapter of the Association for Computational Linguistics (NAACL-03)*, Edmonton, Alberta.

Mehryar Mohri and Michael Riley. 2001. A weight pushing algorithm for large vocabulary speech recognition. In *European Conference on Speech Communication and Technology*, pages 1603–1606.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

Matt Post and Daniel Gildea. 2009. Bayesian learning of a tree substitution grammar. In *Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics (ACL-09)*, Suntec, Singapore.

Helmut Schmid. 2004. Efficient parsing of highly ambiguous context-free grammars with bit vectors. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING-04)*, Geneva, Switzerland.

Xinying Song, Shilin Ding, and Chin-Yew Lin. 2008. Better binarization for the CKY parsing. In *2008 Conference on Empirical Methods in Natural Language Processing (EMNLP-08)*, Honolulu, Hawaii.

Hao Zhang, Liang Huang, Daniel Gildea, and Kevin Knight. 2006. Synchronous binarization for machine translation. In *Proceedings of the 2006 Meeting of the North American chapter of the Association for Computational Linguistics (NAACL-06)*, New York, NY.