Fast and Accurate Arc Filtering for Dependency Parsing

Shane Bergsma
Department of Computing Science
University of Alberta
sbergsma@ualberta.ca

Colin Cherry
Institute for Information Technology
National Research Council Canada
colin.cherry@nrc-cnrc.gc.ca

Abstract

We propose a series of learned arc filters to speed up graph-based dependency parsing. A cascade of filters identify implausible head-modifier pairs, with time complexity that is first linear, and then quadratic in the length of the sentence. The linear filters reliably predict, in context, words that are roots or leaves of dependency trees, and words that are likely to have heads on their left or right. We use this information to quickly prune arcs from the dependency graph. More than 78% of total arcs are pruned while retaining 99.5% of the true dependencies. These filters improve the speed of two state-of-the-art dependency parsers, with low overhead and negligible loss in accuracy.

1 Introduction

Dependency parsing finds direct syntactic relationships between words by connecting head-modifier pairs into a tree structure. Dependency information is useful for a wealth of natural language processing tasks, including question answering (Wang et al., 2007), semantic parsing (Poon and Domingos, 2009), and machine translation (Galley and Manning, 2009).

We propose and test a series of **arc filters** for graph-based dependency parsers, which rule out potential head-modifier pairs before parsing begins. In doing so, we hope to eliminate implausible links early, saving the costs associated with them, and speeding up parsing. In addition to the scaling benefits that come with faster processing, we hope to enable richer features for parsing by constraining the set of arcs that need to be considered. This could allow extremely large feature sets (Koo et al., 2008), or the look-up of expensive corpus-based features such as word-pair mutual information (Wang et al., 2006). These filters could also facilitate expensive learning algorithms, such as semi-supervised approaches (Wang et al., 2008).

We propose three levels of filtering, which are applied in a sequence of increasing complexity:

Rules: A simple set of machine-learned rules based only on parts-of-speech. They prune over 25% of potential arcs with almost no loss in coverage. Rules save on the wasted effort for assessing implausible arcs such as DT $\rightarrow$ DT.

Linear: A series of classifiers that tag words according to their possible roles in the dependency tree. By treating each word independently and ensuring constant-time feature extraction, they operate in linear time. We view these as a dependency-parsing analogue to the span-pruning proposed by Roark and Hollingshead (2008). Our fast linear filters prune 54.2% of potential arcs while recovering 99.7% of true pairs.

Quadratic: A final stage that looks at pairs of words to prune unlikely arcs from the dependency tree. By employing a light-weight feature set, this high-precision filter can enable more expensive processing on the remaining plausible dependencies.

Collectively, we show that more than 78% of total arcs can be pruned while retaining 99.5% of the true dependencies. We test the impact of these filters at both train and test time, using two state-of-the-art discriminative parsers, demonstrating speed-ups of between 1.9 and 5.6, with little impact on parsing accuracy.

Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010), pages 53–61,
Beijing, August 2010

Figure 1: An example dependency parse.

2 Dependency Parsing

A dependency tree represents the syntactic structure of a sentence as a directed graph (Figure 1), with a node for each word, and arcs indicating head-modifier pairs (Meĺčuk, 1987). Though dependencies can be extracted from many formalisms, there is a growing interest in predicting dependency trees directly. To that end, there are two dominant approaches: graph-based methods, characterized by arc features in an exhaustive search, and transition-based methods, characterized by operational features in a greedy search (McDonald and Nivre, 2007). We focus on graph-based parsing, as its exhaustive search has the most to gain from our filters.

Graph-based dependency parsing finds the highest-scoring tree according to a scoring function that decomposes under an exhaustive search (McDonald et al., 2005). The most natural decomposition scores individual arcs, represented as head-modifier pairs $[h, m]$. This enables search by either minimum spanning tree (West, 2001) or by Eisner's (1996) projective parser. This paper focuses on the projective case, though our techniques transfer to spanning tree parsing. With a linear scoring function, the parser solves:

$$\text{parse}(s) = \text{argmax}_{t \in s} \sum_{[h,m] \in t} \bar{w} \cdot \bar{f}(h, m, s)$$

The weights $\bar{w}$ are typically learned using an online method, such as an averaged perceptron (Collins, 2002) or MIRA (Crammer and Singer, 2003). 2nd-order searches, which consider two siblings at a time, are available with no increase in asymptotic complexity (McDonald and Pereira, 2006; Carreras, 2007).

The complexity of graph-based parsing is bounded by two processes: parsing (carrying out the argmax) and arc scoring (calculating $\bar{w} \cdot \bar{f}(h, m, s)$). For a sentence with n words, projective parsing takes $O(n^3)$ time, while the spanning tree algorithm is $O(n^2)$. Both parsers require scores for arcs connecting each possible $[h, m]$ pair in s; therefore, the cost of arc scoring is also $O(n^2)$, and may become $O(n^3)$ if the features include words in s between h and m (Galley and Manning, 2009). Arc scoring also has a significant constant term: the number of features extracted for an $[h, m]$ pair. Our in-house graph-based parser collects on average 62 features for each potential arc, a number larger than the length of most sentences. With the cluster-based features suggested by Koo et al. (2008), this could easily grow by a factor of 3 or 4.

The high cost of arc scoring, coupled with the parsing stage's low grammar constant, means that graph-based parsers spend much of their time scoring potential arcs. Johnson (2007) reports that when arc scores have been precomputed, the dynamic programming component of his 1st-order parser can process an amazing 3,580 sentences per second.[1] Beyond reducing the number of features, the easiest way to reduce the computational burden of arc scoring is to score only plausible arcs.

3 Related Work

3.1 Vine Parsing

Filtering dependency arcs has been explored primarily in the form of vine parsing (Eisner and Smith, 2005; Dreyer et al., 2006). Vine parsing establishes that, since most dependencies are short, one can parse quickly by placing a hard constraint on arc length. As this coarse filter quickly degrades the best achievable performance, Eisner and Smith (2005) also consider conditioning the constraint on the part-of-speech (PoS) tags being linked and the direction of the arc, resulting in a separate threshold for each $[\text{tag}(h), \text{tag}(m), \text{dir}(h, m)]$ triple. They sketch an algorithm where the thresholded length for each triple starts at the highest value seen in the training data. Thresholds are then decreased in a greedy fashion, with each step producing the smallest possible reduction in reachable training arcs. We employ this algorithm as a baseline in our experiments. To our knowledge, vine parsing

[1] To calibrate this speed, consider that the publicly available 1st-order MST parser processes 16 sentences per second on modern hardware. This includes I/O costs in addition to the costs of arc scoring and parsing.

has not previously been tested with a state-of-the-art, discriminative dependency parser.

3.2 CFG Cell Classification

Roark and Hollingshead (2008) speed up another exhaustive parsing algorithm, the CKY parser for CFGs, by classifying each word in the sentence according to whether it can open (or close) a multi-word constituent. With a high-precision tagger that errs on the side of permitting constituents, they show a significant improvement in speed with no reduction in accuracy.

It is difficult to port their idea directly to dependency parsing without committing to a particular search algorithm,[2] and thereby sacrificing some of the graph-based formalism's modularity. However, some of our linear filters (see Section 4.3) were inspired by their constraints.

3.3 Coarse-to-fine Parsing

Another common method employed to speed up exhaustive parsers is a coarse-to-fine approach, where a cheap, coarse model prunes the search space for later, more expensive models (Charniak et al., 2006; Petrov and Klein, 2007). This approach assumes a common forest or chart representation, shared by all granularities, where one can efficiently track the pruning decisions of the coarse models. One could imagine applying such a solution to dependency parsing, but the exact implementation of the coarse pass would vary according to the choice in search algorithm. Our filters are much more modular: they apply to both 1^{st}-order spanning tree parsing and 2^{nd}-order projective parsing, with no modification.

Carreras et al. (2008) use coarse-to-fine pruning with dependency parsing, but in that case, a graph-based dependency parser provides the coarse pass, with the fine pass being a far-more-expensive tree-adjoining grammar. Our filters could become a 0^{th} pass, further increasing the efficiency of their approach.

4 Arc Filters

We propose arc filtering as a preprocessing step for dependency parsing. An arc filter removes implausible head-modifier arcs from the complete dependency graph (which initially includes all head-modifier arcs). We use three stages of filters that operate in sequence on progressively sparser graphs: 1) rule-based, 2) linear: a single pass through the n nodes in a sentence ($O(n)$ complexity), and 3) quadratic: a scoring of all remaining arcs ($O(n^2)$). The less intensive filters are used first, saving time by leaving fewer arcs to be processed by the more intensive systems.

Implementations of our rule-based, linear, and quadratic filters are publicly available at:
`http://code.google.com/p/arcfilter/`

4.1 Filter Framework

Our filters assume the input sentences have been PoS-tagged. We also add an artificial root node to each sentence to be the head of the tree's root. Initially, this node is a potential head for all words in the sentence.

Each filter is a supervised classifier. For example, the quadratic filter directly classifies whether a proposed head-modifier pair is *not* a link in the dependency tree. Training data is created from annotated trees. All possible arcs are extracted for each training sentence, and those that are present in the annotated tree are labeled as class -1, while those not present are $+1$. A similar process generates training examples for the other filters. Since our goal is to only filter very implausible arcs, we bias the classifier to high precision, increasing the cost for misclassifying a true arc during learning.[3]

Class-specific costs are command-line parameters for many learning packages. One can interpret the learning objective as minimizing regularized, weighted loss:

$$\min_{\bar{w}} \frac{1}{2}||\bar{w}||^2 + C_1 \sum_{i:y_i=1} l(\bar{w}, y_i, \bar{x}_i) + C_2 \sum_{i:y_i=-1} l(\bar{w}, y_i, \bar{x}_i) \quad (1)$$

where $l()$ is the learning method's loss function, $\bar{x}_i$ and y_i are the features and label for the ith

[2] Johnson's (2007) split-head CFG could implement this idea directly with little effort.

[3] Learning with a cost model is generally preferable to first optimizing error rate and then thresholding the prediction values to select a high-confidence subset (Joachims, 2005), but the latter approach was used successfully for cell classification in Roark and Hollingshead (2008).

not a h	" " , . ; | CC PRP$ PRP EX -RRB- -LRB-
no $* \leftarrow m$	EX LS POS PRP$
no $m \rightarrow *$	. RP
not a root	, DT
no $h \leftarrow m$	DT$\leftarrow${DT,JJ,NN,NNP,NNS,.} CD$\leftarrow$CD NN$\leftarrow${DT,NNP} NNP$\leftarrow${DT,NN,NNS}
no $m \rightarrow h$	{DT,IN,JJ,NN,NNP}$\rightarrow$DT NNP$\rightarrow$IN IN$\rightarrow$JJ

Table 1: Learned rules for filtering dependency arcs using PoS tags. The rules filter 25% of possible arcs while recovering 99.9% of true links.

training example, $\bar{w}$ is the learned weight vector, and C_1 and C_2 are the class-specific costs. High precision is obtained when $C_2 >> C_1$. For an SVM, $l(\bar{w}, y_i, \bar{x}_i)$ is the standard hinge loss.

We solve the SVM objective using LIBLINEAR (Fan et al., 2008). In our experiments, each filter is a linear SVM with the typical L1 loss and L2 regularization.[4] We search for the best combination of C_1 and C_2 using a grid search on development data. At test time, an arc is filtered if $\bar{w} \cdot \bar{x} > 0$.

4.2 Rule-Based Filtering

Our rule-based filters seek to instantly remove those arcs that are trivially implausible on the basis of their head and modifier PoS tags. We first extract labeled examples from gold-standard trees for whenever a) a word is not a head, b) a word does not have a head on the left (resp. right), and c) a pair of words is not linked. We then trained high-precision SVM classifiers. The only features in $\bar{x}$ are the PoS tag(s) of the head and/or modifier. The learned feature weights identify the tags and tag-pairs to be filtered. For example, if a tag has a positive weight in the not-a-head classifier, all arcs having that node as head are filtered.

The classier selects a small number of high-precision rules, shown in Table 1. Note that the rules tend to use common tags with well-defined roles. By focusing on weighted loss as opposed to arc frequency, the classifier discovers structural zeros (Mohri and Roark, 2006), events which could have been observed, but were not. We consider this an improvement over the frequency-based length thresholds employed previously in tag-specific vine parsing.

4.3 Linear-Time Filtering

In the linear filtering stage, we filter arcs on the basis of single nodes and their contexts, passing through the sentences in linear time. For each node, eight separate classifiers decide whether:

1. It is *not* a head (i.e., it is a leaf of the tree).
2. Its head is on the left/right.
3. Its head is within 5 nodes on the left/right.
4. Its head is immediately on the left/right.
5. It is the root.

For each of these decisions, we again train high-precision SVMs with $C_2 >> C_1$, and filter directly based on the classifier output.

If a word is not a head, all arcs with the given word as head can be pruned. If a word is deemed to have a head within a certain range on the left or right, then all arcs that do not obey this constraint can be pruned. If a root is found, no other words should link to the artificial root node. Furthermore, in a projective dependency tree, no arc will cross the root, i.e., there will be no arcs where a head and a modifier lie on either side of the root. We can therefore also filter arcs that violate this constraint when parsing projectively.

Søgaard and Kuhn (2009) previously proposed a tagger to further constrain a vine parser. Their tags are a subset of our decisions (items 4 and 5 above), and have not yet been tested in a state-of-the-art system.

Development experiments show that if we could perfectly make decisions 1-5 for each word, we could remove 91.7% of the total arcs or 95% of negative arcs, close to the upper bound.

Features

Unlike rule-based filtering, linear filtering uses a rich set of features (Table 2). Each feature is a

[4] We also tried L1-regularized filters. L1 encourages most features to have zero weight, leading to more compact and hence faster models. We found the L1 filters to prune fewer arcs at a given coverage level, providing less speed-up at parsing time. Both L1 and L2 models are available in our publicly available implementation.

PoS-tag features	Other features
tag_i	word_i
tag_i, tag_{i-1}	word_{i+1}
tag_i, tag_{i+1}	word_{i-1}
tag_{i-1}, tag_{i+1}	shape_i
tag_{i-2}, tag_{i-1}	prefix_i
tag_{i+1}, tag_{i+2}	suffix_i
tag_j, Left, $_{j=i-5\ldots i-1}$	i
tag_j, Right, $_{j=i+1\ldots i+5}$	i, n
tag_j, (i-j), $_{j=i-5\ldots i-1}$	n - i
tag_j, (i-j), $_{j=i+1\ldots i+5}$	

Table 2: Linear filter features for a node at position i in a sentence of length n. Each feature is also conjoined (unless redundant) with word_i, tag_i, shape_i, prefix_i, and suffix_i (both 4 letters). The shape is the word normalized using the regular expressions [A-Z]+ $\rightarrow$ A and [a-z]+ $\rightarrow$ a.

Binary features	
sign(h-m)	tags_{hm}
tag_{m-1}, tags_{hm}	tag_{m+1}, tags_{hm}
tag_{h-1}, tags_{hm}	tag_{h+1}, tags_{hm}
sign(h-m), tag_h, word_m	
sign(h-m), word_h, tag_m	
Real features $\Rightarrow$ values	
sign(h-m) $\Rightarrow$ h-m	
tag_h, tag_m $\Rightarrow$ h-m	
tag_k, tags_{hm} $\Rightarrow$ Count($\text{tag}_k \in \text{tags}_{h\ldots m}$)	
word_k, tags_{hm} $\Rightarrow$ Count($\text{word}_k \in \text{words}_{h\ldots m}$)	

Table 3: Quadratic filter features for a head at position h and a modifier at position m in a sentence of length n. Here tags_{hm} = (sign(h-m), tag_h, tag_m), while $\text{tags}_{h\ldots m}$ and $\text{words}_{h\ldots m}$ are all the tags (resp. words) between h and m, but within ± 5 positions of h or m.

binary indicator feature. To increase the speed of applying eight classifiers, we use the same feature vector for each of the decisions; learning gives eight different weight vectors, one corresponding to each decision function. Feature extraction is constrained to be $O(1)$ for each node, so that overall feature extraction and classification remain a fast $O(n)$ complexity. Feature extraction would be $O(n^2)$ if, for example, we had a feature for *every* tag on the left or right of a node.

Combining linear decisions

We originally optimized the C_1 and C_2 parameter separately for each linear decision function. However, we found we could substantially improve the collective performance of the linear filters by searching for the optimal combination of the component decisions, testing different levels of precision for each component. We selected a few of the best settings for each decision when optimized separately, and then searched for the best combination of these candidates on development data (testing 12960 combinations in all).

4.4 Quadratic-Time Filtering

In the quadratic filtering stage, a single classifier decides whether each head-modifier pair should be filtered. It is trained and applied as described in Section 4.1.

While theoretically of the same complexity as the parser's arc-scoring function ($O(n^2)$), this process can nevertheless save time by employing a compact feature set. We view quadratic filtering as a light preprocessing step, using only a portion of the resources that might be used in the final scoring function.

Features

Quadratic filtering uses both binary *and* real-valued features (Table 3). Real-valued features promote a smaller feature space. For example, one value can encode distance rather than separate features for different distances. We also generalize the "between-tag features" used in McDonald et al. (2005) to be the count of each tag between the head and modifier. The count may be more informative than tag presence alone, particularly for high-precision filters. We follow Galley and Manning (2009) in using only between-tags within a fixed range of the head or modifier, so that the extraction for each pair is $O(1)$ and the overall feature extraction is $O(n^2)$.

Using only a subset of the between-tags as features has been shown to improve speed but impair parser performance (Galley and Manning, 2009). By filtering quickly first, then scoring all remaining arcs with a cubic scoring function in the parser, we hope to get the best of both worlds.

5 Filter Experiments

Data

We extract dependency structures from the Penn Treebank using the Penn2Malt extraction tool,[5] which implements the head rules of Yamada and Matsumoto (2003). Following convention, we divide the Treebank into train (sections 2–21), development (22) and test sets (23). The development and test sets are re-tagged using the Stanford tagger (Toutanova et al., 2003).

Evaluation Metrics

To measure intrinsic filter quality, we define **Reduction** as the proportion of total arcs removed, and **Coverage** as the proportion of true head-modifier arcs retained. Our evaluation asks, for each filter, what Reduction can be obtained at a given Coverage level? We also give **Time**: how long it takes to apply the filters to the test set (excluding initialization).

We compute an **Upper Bound** for Reduction on development data. There are 1.2 million potential dependency links in those sentences, 96.5% of which are not present in a gold standard dependency tree. Therefore, the maximum achievable Reduction is 96.5%.

Systems

We evaluate the following systems:

- **Rules**: the rule-based filter (Section 4.2)
- **Lin.**: the linear-time filters (Section 4.3)
- **Quad.**: the quadratic filter (Section 4.4)

The latter two approaches run on the output of the previous stage. We compare to the two vine parsing approaches described in Section 3.1:

- **Len-Vine** uses a hard limit on arc length.
- **Tag-Vine** (later, **Vine**) learns a maximum length for dependency arcs for every head/modifier tag-combination and order.

5.1 Results

We set each filter's parameters by selecting a Coverage-Reduction tradeoff on development data (Figure 2). The Lin curve is obtained by varying both the C_1/C_2 cost parameters and the combination of components (plotting the best Reduction at each Coverage level). We chose the linear filters with 99.8% Coverage at a 54.2% Reduction. We apply Quad on this output, varying the cost parameters to produce its curve. Aside from Len-Vine, all filters remove a large number of arcs with little drop in Coverage.

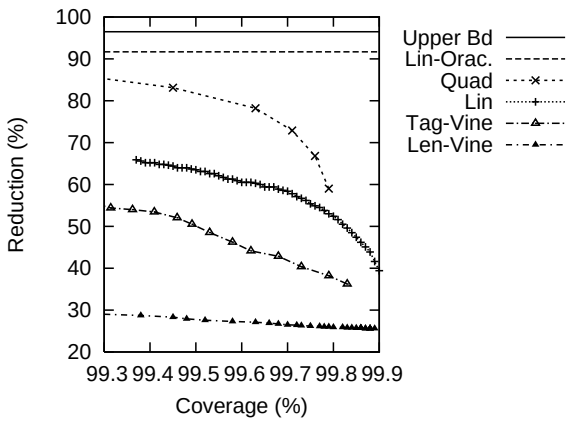

Figure 2: Filtering performance for different filters and cost parameters on development data. Lin-Orac indicates the percentage filtered using perfect decisions by the linear components.

Filter	Coverage	Reduct.	Time (s)
Vine	99.62	44.0	2.9s
Rules	99.86	25.8	1.3s
Lin.	99.73	54.2	7.3s
Quad.	99.50	78.4	16.1s

Table 4: Performance (%) of filters on test data.

After selecting a desired trade-off for each classifier, we move to final filtering experiments on unseen test data (Table 4). The linear filter removes well over half the links but retains an astounding 99.7% of correct arcs. Quad removes 78.4% of arcs at 99.5% Coverage. It thus reduces the number of links to be scored by a dependency parser by a factor of five.

The time for filtering the 2416 test sentences varies from almost instantaneous for Vine and Rules to around 16 seconds for Quad. Speed numbers are highly machine, design, and implemen-

[5] http://w3.msi.vxu.se/~nivre/research/Penn2Malt.html

Decision	Precision	Recall
No-Head	99.9	44.8
Right-$\emptyset$	99.9	28.7
Left-$\emptyset$	99.9	39.0
Right-5	99.8	31.5
Left-5	99.9	19.7
Right-1	99.7	6.2
Left-1	99.7	27.3
Root	98.6	25.5

Table 5: Linear Filters: Test-set performance (%) on decisions for components of the combined 54.2 Reduct./99.73 Coverage linear filter.

Type	Coverage	Reduct.	Oracle
All	**99.73**	**54.2**	**91.8**
All\No-Head	99.76	46.4	87.2
All\Left-$\emptyset$	99.74	53.2	91.4
All\Right-$\emptyset$	99.75	53.6	90.7
All\Left-5	99.74	53.2	89.7
All\Right-5	99.74	51.6	90.4
All\Left-1	99.75	53.5	90.8
All\Right-1	99.73	53.9	90.6
All\Root	99.76	50.2	90.0

Table 6: Contribution of different linear filters to test set performance (%). Oracle indicates the percentage filtered by perfect decisions.

tation dependent, and thus we have stressed the asymptotic complexity of the filters. However, the timing numbers show that arc filtering can be done quite quickly. Section 6 confirms that these are very reasonable costs in light of the speed-up in overall parsing.

5.2 Linear Filtering Analysis

It is instructive to further analyze the components of the linear filter. Table 5 gives the performance of each classifier on its specific decision. **Precision** is the proportion of positive classifications that are correct. **Recall** is the proportion of positive instances that are classified positively (e.g. the proportion of actual roots that were classified as roots). The decisions correspond to items 1-5 in Section 4.3. For example, *Right-*$\emptyset$ is the decision that a word has *no* head on the right.

Most notably, the optimum *Root* decision has much lower Precision than the others, but this has little effect on its overall accuracy as a filter (Table 6). This is perhaps because the few cases of false positives are still likely to be main verbs or auxiliaries, and thus still still likely to have few links crossing them. Thus many of the filtered links are still correct.

Table 6 provides the performance of the classifier combination when each linear decision is excluded. *No-Head* is the most important component in the oracle and the actual combination.

6 Parsing Experiments

6.1 Set-up

In this section, we investigate the impact of our filters on graph-based dependency parsers. We train each parser unfiltered, and then measure its speed and accuracy once filters have been applied. We use the same training, development and test sets described in Section 5. We evaluate unlabeled dependency parsing using head **accuracy**: the percentage of words (ignoring punctuation) that are assigned the correct head.

The filters bypass feature extraction for each filtered arc, and replace its score with an extremely low negative value. Note that 2^{nd}-order features consider $O(n^3)$ $[h, m_1, m_2]$ triples. These triples are filtered if at least one component arc ($[h, m_1]$ or $[h, m_2]$) is filtered.

In an optimal implementation, we might also have the parser re-use features extracted during filtering when scoring the remaining arcs. We did not do this. Instead, filtering was treated as a preprocessing step, which maximizes the portability of the filters across parsers. We test on two state-of-the art parsers:

MST We modified the publicly-available MST parser (McDonald et al., 2005)[6] to employ our filters before carrying out feature extraction. MST is trained with 5-best MIRA.

DepPercep We also test an in-house dependency parser, which conducts projective first and 2^{nd}-order searches using the split-head CFG described by Johnson (2007), with a weight vector trained using an averaged perceptron (Collins,

[6] http://sourceforge.net/projects/mstparser/

Filter	Cost	DepPercep-1 Acc.	DepPercep-1 Time	DepPercep-2 Acc.	DepPercep-2 Time	MST-1 Acc.	MST-1 Time	MST-2 Acc.	MST-2 Time
None	+0	91.8	348	92.5	832	91.2	153	91.9	200
Vine	+3	91.7	192	92.3	407	91.2	99	91.8	139
Rules	+1	91.7	264	92.4	609	91.2	125	91.9	167
Linear	+7	91.7	168	92.4	334	91.2	88	91.8	121
Quad.	+16	91.7	79	92.3	125	91.2	58	91.8	80

Table 7: The effect of filtering on the speed and accuracy on 1st and 2nd-order dependency parsing.

2002). Its features are a mixture of those described by McDonald et al. (2005), and those used in the Koo et al. (2008) baseline system; we do not use word-cluster features.

DepPercep makes some small improvements to MST's 1st-order feature set. We carefully determined which feature types should have distance appended in addition to direction. Also, inspired by the reported utility of mixing PoS tags and word-clusters (Koo et al., 2008), we created versions of all of the "Between" and "Surrounding Word" features described by McDonald et al. (2005) where we mix tags and words.[7]

DepPercep was developed with quadratic filters in place, which enabled a fast development cycle for feature engineering. As a result, it does not implement many of the optimizations in place in MST, and is relatively slow unfiltered.

6.2 Results

The parsing results are shown in Table 7, where times are given in seconds, and **Cost** indicates the additional cost of filtering. Note that the impact of all filters on accuracy is negligible, with a decrease of at most 0.2%. In general, parsing speed-ups mirror the amount of arc reduction measured in our filter analysis (Section 5.1).

Accounting for filter costs, the benefits of quadratic filtering depend on the parser. The extra benefit of quadratic over linear is substantial for DepPercep, but less so for 1st-order MST.

MST shows more modest speed-ups than DepPercep, but MST is already among the fastest publicly-available data-driven parsers. Under quadratic filtering, MST-2 goes from processing 12 sentences per second to 23 sentences.[8]

DepPercep-2 starts slow, but benefits greatly from filtering. This is because, unlike MST-2, it does not optimize feature extraction by factoring its ten 2nd-order features into two triple ($[h, m_1, m_2]$) and eight sibling ($[m_1, m_2]$) features. This suggests that filtering could have a dramatic effect on a parser that uses more than a few triple features, such as Koo et al. (2008).

7 Conclusion

We have presented a series of arc filters that speed up graph-based dependency parsing. By treating filtering as weighted classification, we learn a cascade of increasingly complex filters from tree-annotated data. Linear-time filters prune 54% of total arcs, while quadratic-time filters prune 78%. Both retain at least 99.5% of true dependencies. By testing two state-of-the-art dependency parsers, we have shown that our filters produce substantial speed improvements in even carefully-optimized parsers, with negligible losses in accuracy. In the future we hope to leverage this reduced search space to explore features derived from large corpora.

References

Carreras, Xavier, Michael Collins, and Terry Koo. 2008. TAG, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *CoNLL*.

Carreras, Xavier. 2007. Experiments with a higher-order projective dependency parser. In *EMNLP-CoNLL*.

[7] This was enabled by using word features only when the word is among the 800 most frequent in the training set.

[8] This speed accounts for 25 total seconds to apply the rules, linear, and quadratic filters.

Charniak, Eugene, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R. Shrivaths, Jeremy Moore, Michael Pozar, and Theresa Vu. 2006. Multilevel coarse-to-fine PCFG parsing. In *HLT-NAACL*.

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

Crammer, Koby and Yoram Singer. 2003. Ultraconservative online algorithms for multiclass problems. *JMLR*, 3:951–991.

Dreyer, Markus, David A. Smith, and Noah A. Smith. 2006. Vine parsing and minimum risk reranking for speed and precision. In *CoNLL*.

Eisner, Jason and Noah A. Smith. 2005. Parsing with soft and hard constraints on dependency length. In *IWPT*.

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

Fan, Rong-En, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *JMLR*, 9:1871–1874.

Galley, Michel and Christopher D. Manning. 2009. Quadratic-time dependency parsing for machine translation. In *ACL-IJCNLP*.

Joachims, Thorsten. 2005. A support vector method for multivariate performance measures. In *ICML*.

Johnson, Mark. 2007. Transforming projective bilexical dependency grammars into efficiently-parsable CFGs with unfold-fold. In *ACL*.

Koo, Terry, Xavier Carreras, and Michael Collins. 2008. Simple semi-supervised dependency parsing. In *ACL-08: HLT*.

McDonald, Ryan and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *EMNLP-CoNLL*.

McDonald, Ryan and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *EACL*.

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

Meĺčuk, Igor A. 1987. *Dependency syntax: theory and practice*. State University of New York Press.

Mohri, Mehryar and Brian Roark. 2006. Probabilistic context-free grammar induction based on structural zeros. In *HLT-NAACL*.

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

Poon, Hoifung and Pedro Domingos. 2009. Unsupervised semantic parsing. In *EMNLP*.

Roark, Brian and Kristy Hollingshead. 2008. Classifying chart cells for quadratic complexity context-free inference. In *COLING*.

Søgaard, Anders and Jonas Kuhn. 2009. Using a maximum entropy-based tagger to improve a very fast vine parser. In *IWPT*.

Toutanova, Kristina, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *NAACL*.

Wang, Qin Iris, Colin Cherry, Dan Lizotte, and Dale Schuurmans. 2006. Improved large margin dependency parsing via local constraints and Laplacian regularization. In *CoNLL*.

Wang, Mengqiu, Noah A. Smith, and Teruko Mitamura. 2007. What is the Jeopardy model? A quasi-synchronous grammar for QA. In *EMNLP-CoNLL*.

Wang, Qin Iris, Dale Schuurmans, and Dekang Lin. 2008. Semi-supervised convex training for dependency parsing. In *ACL-08: HLT*.

West, D. 2001. *Introduction to Graph Theory*. Prentice Hall, 2nd edition.

Yamada, Hiroyasu and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *IWPT*.