Dependency Parsing with a Linear Pipeline Model

Ming-Wei Chang **Quang Do** **Dan Roth**
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{mchang21, quangdo2, danr}@uiuc.edu

Abstract

Pipeline computation, in which a task is decomposed into several stages that are solved sequentially, is a common computational strategy in NLP. The key problem of this model is that it results in error accumulation and cannot correct mistakes in previous stages. We develop a framework for decisions in pipeline models which addresses these difficulties and apply it in the context of *bottom up dependency parsing* for English. We show significant improvements in the accuracy of the inferred trees relative to existing models. Interestingly, the proposed algorithm shines especially when evaluated globally, where our results are significantly better than those of existing approaches.

1 Dependency Parsing by Pipeline Models

We are interested in this paper in *deep pipelines* – in the sense that a large number of predictions are being chained. We define pipeline to be a process in which (1) decisions made in stage ith depend on decisions made earlier, and (2) the input supplied to stage ith depends on decisions made earlier.

Our work is done in the context of developing a dependency parser for projective languages. Dependency trees provide a syntactic representation that encodes functional relationships between words; it is relatively independent of the theory of grammar and can be used to represent the structure of sentences in different languages. Dependency structures are more efficient to parse (Eisner, 1996) and are believed to be easier to learn, yet they still capture much of the predicate-argument information needed in applications (Haghighi et al., 2005). This is one reason for the recent interest in learning dependency structures (Eisner, 1996; McDonald et al., 2005; Yamada and Matsumoto, 2003).

Eisner developed a generative algorithm with $O(n^3)$ parsing time. His model, however, seems to be limited in dealing with complex and long sentences. (McDonald et al., 2005) build on this work, and learn to re-rank the top trees proposed by Eisner. A completely different approach is proposed by (Yamada and Matsumoto, 2003), that develop a bottom-up approach and learn the parsing decisions between consecutive pairs of words in a sentence. Local actions are used to generate a dependency tree using a shift-reduce parsing approach (Aho et al., 1986). This is a true pipeline approach, in that the classifiers are trained on individual decisions rather than on the overall quality of the parse, and chained to yield the global structure.

The key in a pipeline model is that making a decision with respect to the edge (i, j) may gain from taking into account decisions already made with respect to neighboring edges. Clearly, if the edge predictor is correct, n^2 predictions are sufficient to construct the correct tree. In reality this isn't the case, and there is a need to devise a robust policy.

This policy is exemplified in the work of (Yamada and Matsumoto, 2003), a bottom up approach, that is most related to the work presented here. Their model is a "traditional" pipeline model – a classifier suggests a decision that, once taken, determines the next action to be taken (as well as the input the next action observes).

Like the shift-reduce parsing algorithm in (Aho et al., 1986), our dependency parsing algorithm uses the actions to build to tree. We suggest a new (hierarchical) set of actions: *Shift, Left, Right, WaitLeft, WaitRight*. The new set of actions also better supports our improved parsing algorithm. In our work, machine learning is used to predict the appropriate actions in parsing. We suggest that for a pipeline algorithm to be accurate, it needs to reduce the number of action it takes. We then use local search based inference over consecutive actions and better exploit the dependencies among them.

We propose a framework for improving pipeline processing based on the following principles:
(1) Devise an algorithm that reduces the number of actions taken when inferring the dependence tree, and
(2) Use local inference in order to make more robust local decisions.

2 Efficient Dependency Parsing

This section describes our dependency parsing (DP) algorithm and justifies its advantages as a pipeline DP model. We propose an improved pipeline framework of (Yamada and Matsumoto, 2003) based on the principles we proposed. First, improved accuracy of the predicted actions. Second, we claim that the parsing algorithm does not perform the unnecessary actions.

For many languages such as English, Chinese and Japanese (with a few exceptions), projective dependency trees are sufficient to analyze most sentences. Our work is therefore concerned only with projective trees, which we define below. For simplicity, our definitions and notations are presented in the context of English.

Let x, y be two words in the sentence T. By $x \rightarrow y$ we mean that x is the *direct parent* of y. $x \rightarrow^* y$ denotes that x is the *ancestor* of y; $x \leftrightarrow y$ denotes that $x \rightarrow y$ or $y \rightarrow x$; and, $x < y$ represents that x is *in front* of y in T.

Definition 1 (Projective Language) *(Nivre, 2003)* $\forall a, b, c \in T, a \leftrightarrow b$ *and* $a < c < b$ *imply that* $a \rightarrow^* c$ *or* $b \rightarrow^* c$.

2.1 Efficient Dependency Parsing Algorithm

(Yamada and Matsumoto, 2003) proposed a bottom-up dependency parsing algorithm. This is a bottom-up approach that uses SVMs to learn the parsing decisions between consecutive pairs of words in the sentences. The local actions, chosen among *Shift, Left, Right*, are used to generate a dependency tree using a shift-reduce parsing approach. This is a true pipeline approach in that the classifiers are trained on individual decisions rather than on the overall quality of the parsing, and chained to yield the global structure. It suffers from the limitations of pipeline processing, such as accumulation of errors, nevertheless, yields very competitive parsing results.

In particular, we suggest that for a pipeline algorithm to be accurate, it needs to minimize the number of actions it takes. In the rest of this section we describe the algorithm, analyze it and prove that it achieves this goal.

The parsing algorithm is a modified shift-reduce parser that makes use of the actions described below and applies them in a left to right manner on consecutive pairs of words (a, b) $(a < b)$ in the sentence T. The actions are used as follows:

S*hift*: there is no relation between a and b,

R*ight*: b is the parent of a,

L*eft*: a is the parent of b.

In order to complete the description of the algorithm we need to describe which edge to consider once an action is taken. We describe it via the notion of the *focus point*. In fact, with a few exceptions, determining the focus point does not affect the correctness of the algorithm. It is easy to show that (almost) any edge in the sentence can be used to determine the focus point and, if the correct action is chosen for the corresponding edge, this will eventually yield the correct tree (but may necessitate multiple cycles through the sentence).

In practice, the actions chosen will be noisy, and a wasteful focus point policy will result in a large number of actions, and thus in error accumulation. To minimize the number of actions taken, we want to find a good focus point placement policy.

There are three natural placement policies that we considered. In all cases, after **S** the focus point moves one word to the right. After **L** or **R** we can do one of the following:

Start Over The focus moves to the first word in T.

Stay The focus moves to the next word to the right. That is, for $T = (a, b, c)$, and focus being a, an **L** action will result is the focus being a, while **R** action results in the focus being b.

Step Back The focus moves to the previous word (on the left). That is, for $T = (a, b, c)$, and focus being b, in both cases, a will be the focus point.

Note that the different placement policies affect the final accuracy a lot. In (Yamada and Matsumoto, 2003), they mention that they move the focus point back after **R** is performed, but it is not clear the movement after executing **L** actions in their algorithms. [1] We claim that if **Step Back** is used, the algorithm will not waste the action. Therefore, we achieve the goal of reducing the number of actions in pipeline algorithms. Notice that using the this policy, when **L** is taken, the pair (a, b) is reconsidered, but with new information, since now it is known that c is the child of b. Although this seems wasteful, we will show this to be the best policy, as it allows us to parse the sentence using a single left to right pass.

As mentioned above, each of these policies yields the correct tree. Table 1 compares the three policies in terms of the number of actions required to build a tree.

Policy	#*Shift*	#*Left*	#*Right*
Start over	156545	26351	27918
Stay	117819	26351	27918
Step back	43374	26351	27918

Table 1: The number of actions required to build all the trees for the sentences in section 23 of Penn Treebank (Marcus et al., 1993). These statistics is taken with correct (gold-standard) actions.

It is clear from Table 1 that the policies are very different and **Step Back** is a best choice. Note also that the number of **L** and **R** actions used is the same across models since the actions are the gold-standard actions. Algorithm 2 depicts the parsing algorithm.

[1]By private communication, we realize that they also move the focus point back after **L** is performed. However, it is not clear why they made this choice from their paper. In other word, we provide some insight and build the framework for analyzing this type of algorithms in this paper.

Algorithm 2 Pseudo Code of the dependency parsing algorithm. *getFeatures* extracts the features describing the currently considered pair of words; *getAction* determines the appropriate action for the pair; *assignParent* assigns the parent for the child word based on the action; and *deleteEdge* deletes the edge in T at *focuspoint* once the action is taken.

```
Let $t$ represents for a word and its part of speech
For sentence $T = \{t_1, t_2, \ldots, t_n\}$
$focus = 1$
while $focus < |T|$ do
  $\vec{v} = getFeatures(t_{focus}, t_{focus+1})$
  $\alpha = getAction(t_{focus}, t_{focus+1}, \vec{v})$
  if $\alpha = \mathbf{L}$ or $\alpha = \mathbf{R}$ then
    $assignParent(t_{focus}, t_{focus+1}, \alpha)$
    $deleteEdge(T, focus, \alpha)$
    // performing Step Back here
    $focus = focus - 1$
  else
    $focus = focus + 1$
  end if
end while
```

2.2 Correctness and Pipeline Properties

We can prove two properties of our algorithm. First we show that the algorithm requires only one pass over the sentence to build the dependency tree. Then, we show that the algorithm does not waste actions in the sense that it never considers a pair twice in the same situations. Consequently, this shows that under the assumption of a perfect action predictor, our algorithm makes the smallest number of actions, among all algorithms that build a tree sequentially in one pass.

Note that this may not be true if the action classifier is not perfect, and that one can contrive examples in which an algorithm that makes several passes on a sentence can actually make less actions than a single pass algorithm. In practice, however, this is unlikely to occur.

Claim 1 *A dependency parsing algorithm that uses the Step Back policy completes the tree when it reaches the end of the sentence for the first time.*

In order to prove the algorithm we need the following definition. We call a pair of words (a, b) a *free pair* if and only if there is relation between two consecutive words a and b and the algorithm can perform **L** or **R** action on that pairs right after they are considered. Formally,

Definition 2 (*free pair) A pair (a, b) considered by the algorithm is a free pair, if it satisfies the following conditions.*

1. *$a \leftrightarrow b$*
2. *a, b are consecutive*
3. *the child word is a complete subtree.*

Proof. : It is easy to see that there is at least one *free pair* in T, with $|T| > 1$. The reason is that if no such pair exists, there must be three words $\{a, b, c\}$ where $a \leftrightarrow b$, $a < c < b$ and $\neg(a \rightarrow c \vee b \rightarrow c)$. However, this violates the properties of a projective language.

Now, we claim that, when using Step Back, the focus point is always to the left of all free pairs in T. This is clearly true when the algorithm starts. Assume that (a, b) is the first *free pair* in T and let c be just to the left of a and b. Then, the algorithm will not make a **L** or **R** before the focus point meets (a, b), and will make one of these actions then. It's possible that $(a \vee b, c)$ becomes a free pair after removing $b \vee a$ in T. We also know that there is no *free pair* to the left of c. Therefore, during the algorithm, the focus point will always remain to the left of all free pairs. So, when we reach the end of the sentence, every free pair in the sentence has been taken care of, and the sentence has been parsed completed. □

Claim 2 *All actions made by a dependency parsing algorithm that uses the Step Back policy are necessary.*

Proof. : We will show that a pair will be considered only once in a given a situation.

A pair (a, b) will never be considered again if the action taken on it is **R** or **L**, since either a or b will become a child word. Assume that the action taken is **S**, and, w.l.o.g. that this is the rightmost **S** action taken before a non **S** action happens. There are two cases when **S** happens. First, b itself is a parent and cannot be attached to a at this time. Second, there is no relationship between them. Assume the first case happens. The focus point will continue moving to the right, but cannot meet any **R** until it meets an **L** action. At this point it will go back to solve the pair (a, b). Otherwise, it contradicts the fact that the sentence is projective. When considering (a, b) this time, b is a parent, so the information given about the pair (a, b) now is different than what it was in the previous consideration. This shows that we must reconsider (a, b) again to build a tree. Therefore, no actions are wasted. □

2.3 Improving the Parsing Action Set

To improve the accuracy, we suggest a new (hierarchical) set of actions: *Shift, Left, Right, WaitLeft, WaitRight*. Predicting these turns out to be easier due to finer granularity.

W*ait***L***eft*: $a < b$. a is the parent of b, but it's possible that b is a parent of other nodes. Action is deferred. If we perform **L***eft* instead, the child of b can not find its parents later.

W*ait***R***ight*: $a < b$. b is the parent of a, but it's possible that a is a parent of other nodes. Similar to **WL**, action is deferred.

The new set of actions better supports our parsing algorithm. While testing on different placement policies, the new action set is always helpful. When **W***ait***L***eft* or **W***ait***R***ight* is performed, the focus will move to the next word in **Step Back** placement policy. However, it is very interesting to notice that **W***ait***R***ight* is not needed in projective languages if **Step Back** is used. This give us another strong reason to use **Step Back**, since the classification becomes much easier when number of classes are fewer.

Once the parsing algorithm, along with the focus point policy, is determined, we can train the action classifiers. Given an annotated corpus, the parsing algorithm is used to determine the action taken for each consecutive pair; this is used to train a classifier to predict one of the four actions. The details of the classifier and the feature used are given in Section 4.

When the learned model is evaluated on new data, the sentence is processed left to right and the parsing algorithm, along with classifier, is used to produce the dependency tree. The evaluation process is somewhat more involved, since the action classifier is not used as is, but rather via a local search inference step. This is described in Section 3.

3 A Pipeline Model with Local Search

The advantage of a pipeline model is that it can use more information in a given prediction, information that is taken from the outcomes of previous prediction. However, this may result in accumulating error. The results of Section 2 show that it is essential for our algorithm to use a reliable action predictor. This motivates the following approach for making the local prediction in a pipeline model more reliable. Informally, we devise a local search algorithm and use it as a look ahead policy, when determining the predicted action.

This approach can be used in any pipeline model but we illustrate it below in the context of our dependency parser.

The following example illustrates a situation in which an early mistake in predicting an action cause chain reaction and results in further mistakes. This stresses the importance of correct early decision, and motivates our look ahead policy.

Assume there are four words w, x, y and z in a sentence. The correct dependency is shown in the top part of Figure 1. If the system gives a wrong prediction and makes x a child of w before y and z becomes x's child, we can only consider the relationship between w and y in the next stage. Then, we will never find the correct parent for y and z. The previous prediction error indeed propagates to the next prediction. On the other hand, if the algorithm makes a correct prediction, in the next stage, we do not need to consider w and y. In the pipeline model, we can get additional information if we can avoid errors. Therefore, it is necessary to have a search or an inference framework to help resolve the error accumulation problem.

In order to improve the accuracy, we might want to examine all the combination of actions proposed and choose the one that maximizes the score. It is clearly not tractable to find the global optimal prediction sequence in pipeline model of the depth we consider. The size of the possible label sequence increases exponentially so that we can not examine every possibility. Since in this work, the feature vectors are depend on previous predictions, we cannot consider the dynamic programming here. Therefore, a local search framework which uses additional information, however, is suitable and tractable.

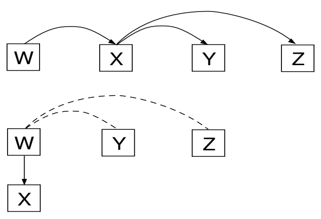

Figure 1: Top figure: the correct dependency between w, x, y and z. Bottom figure: if the algorithm makes a mistake to put x as a child of w before y and z become x's children, we can not find the correct parent for y and z.

The local search algorithm is presented in Fig. 3. The algorithm accepts two parameters, *model* and *depth*. We assume a classifier that can give a confidence in its prediction. This is represented here by *model*.

In this work we use as our learning algorithm a regularized variation of the perceptron update rule as incorporated in SNoW (Roth, 1998; Carlson et al., 1999), a multi-class classifier that is specifically tailored for large scale learning tasks. SNoW uses softmax over the raw activation values as its confidence measure, and it can be shown to be a reliable approximation of the labels' probabilities.

depth is the parameter the determines the depth of the local search. *State* encodes the configuration of the environment (in the context of the dependency parsing this includes the sentence, the focus point and the current parent and children for each node). Note that *State* changes when a prediction is made and that the features extracted for the action classifier also depend on *State*. GoalTest is a function which tests if the *State* is the goal state or not.

The search algorithm will perform a search of length *depth*. Additive scoring is used to score the sequence, and the first action in this sequence is performed. Then, the *State* is updated, determining the next features for the action classifiers and *search* is called again.

One interesting property of this framework is that we use future information in addition to past information. The pipeline model naturally allows access

Algorithm 3 Pseudo code for the local search algorithm. In the algorithm, $\mathbf{y}$ represents the a action sequence. The function *search* considers all possible action sequences with $|depth|$ actions and returns the sequence with highest score.

```
Algo predictAction(model, depth, State)
$x$ = getNextFeature(State)
$\mathbf{y}$ = search($x$, depth, model, State)
$lab = \mathbf{y}[1]$
State = update(State, lab)
return lab

Algo search($x$, depth, model, State)
$maxScore = -\infty$
$F = \{\mathbf{y} \mid \|\mathbf{y}\| = depth\}$
for $\mathbf{y}$ in $F$ do
  $s = 0$, $TmpState = State$
  for $i = 1 \ldots depth$ do
    $x$ = getNextFeature(TmpState)
    $s = s + \log(\text{score}(\mathbf{y}[i], x))$
    TmpState = update(TmpState, $\mathbf{y}[i]$)
  end for
  if $s > maxScore$ then
    $\hat{\mathbf{y}} = \mathbf{y}$
    $maxScore = s$
  end if
end for
return $\hat{\mathbf{y}}$
```

to all the past information. But, since our algorithm uses the search as a look ahead policy, it makes use also of future predictions. The significance of this is clear becomes the experiments in Section 4.

Our algorithm also considers constraints among actions in a sequence. The constraints were extracted automatically from the training action sequences. It turns out that some action sequences are illegal for a projective language and our algorithm exploits that in the local search process, by eliminating these from consideration.

4 Experiments and Results

In our experiment, we used SNoW learning architecture (Roth, 1998) with the perceptron algorithm. We used as training and test data the standard corpus for this task, the Penn Treebank (Marcus et al., 1993). The training set consists of sections 02 to 21, the development set is section 22, and the test set is section 23.

System	Dependency	Sentence	Leaf
Y&M03	90.3	38.4	93.5
N&S04	87.3	30.4	-
M&C&P05	**90.9**	37.5	-
Our Alg.	90.7	**40.4**	**94.0**

Table 2: Comparing the performance of different dependency parsing systems.

We use the same evaluation metrics with (McDonald et al., 2005). Dependency accuracy (*Dependency*) is the proportion of non-root words that are assigned the correct head. Complete accuracy (*Sentence*) indicates the ratio of number of complete correct sentences divided by number of sentences. (*Leaf*) is the dependency accuracy on the leaf nodes. Table 2 show the performances of other dependency parsing systems and ours.

When comparing with other dependency parsing systems it is especially worth noticing that our system gives significantly better accuracy on completely parsed sentences.

5 Conclusion

The results of our *bottom up dependency parsing* show significant improvements in the accuracy of the inferred trees relative to existing models and, interestingly, is doing especially well when evaluated globally, at a sentence level, where our results are better than those of existing approaches – perhaps showing the design goals were achieved.

References

A. V. Aho, R. Sethi, and J. D. Ullman. 1986. Compilers: Principles, techniques, and tools. In *Addison-Wesley Publishing Company, Reading, MA*.

A. Carlson, C. Cumby, J. Rosen, and D. Roth. 1999. The SNoW learning architecture. Technical Report UIUCDCS-R-99-2101, UIUC Computer Science Department, May.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pages 340–345, Copenhagen, August.

A. Haghighi, A. Ng, and C. Manning. 2005. Robust textual inference via graph matching. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 387–394, Vancouver, British Columbia, Canada, October. Association for Computational Linguistics.

M. P. Marcus, B. Santorini, and M. Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, June.

R. McDonald, K. Crammer, and F. Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 91–98, Ann Arbor, Michigan, June. Association for Computational Linguistics.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *8th Int'l Workshop on Parsing Technologies (IWPT 03)*, Nancy, France.

D. Roth. 1998. Learning to resolve natural language ambiguities: A unified approach. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 806–813.

H. Yamada and Y. Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *IWPT2003*.