A Pipeline Framework for Dependency Parsing

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 natural language processing. The key problem of this model is that it results in error accumulation and suffers from its inability to correct mistakes in previous stages. We develop a framework for decisions made via in pipeline models, which addresses these difficulties, and presents and evaluates it in the context of *bottom up dependency parsing* for English. We show improvements in the accuracy of the inferred trees relative to existing models. Interestingly, the proposed algorithm shines especially when evaluated globally, at a sentence level, where our results are significantly better than those of existing approaches.

1 Introduction

A pipeline process over the decisions of learned classifiers is a common computational strategy in natural language processing. In this model a task is decomposed into several stages that are solved sequentially, where the computation in the ith stage typically depends on the outcome of computations done in previous stages. For example, a semantic role labeling program (Punyakanok et al., 2005) may start by using a part-of-speech tagger, then apply a shallow parser to chunk the sentence into phrases, identify predicates and arguments and then classify them to types. In fact, any left to right processing of an English sentence may be viewed as a pipeline computation as it processes a token and, potentially, makes use of this result when processing the token to the right.

The pipeline model is a standard model of computation in natural language processing for good reasons. It is based on the assumption that some decisions might be easier or more reliable than others, and their outcomes, therefore, can be counted on when making further decisions. Nevertheless, it is clear that it results in error accumulation and suffers from its inability to correct mistakes in previous stages. Researchers have recently started to address some of the disadvantages of this model. E.g., (Roth and Yih, 2004) suggests a model in which global constraints are taken into account in a later stage to fix mistakes due to the pipeline. (Punyakanok et al., 2005; Marciniak and Strube, 2005) also address some aspects of this problem. However, these solutions rely on the fact that all decisions are made with respect to the same input; specifically, all classifiers considered use the same examples as their input. In addition, the pipelines they study are shallow.

This paper develops a general framework for decisions in pipeline models which addresses these difficulties. Specifically, we are interested in *deep pipelines* – a large number of predictions that are being chained.

A pipeline process is one in which decisions made in the ith stage (1) depend on earlier decisions and (2) feed on input that depends on earlier decisions. The latter issue is especially important at evaluation time since, at training time, a gold standard data set might be used to avoid this issue.

We develop and study the framework in the context of a *bottom up approach to dependency parsing*. We suggest that two principles to guide the pipeline algorithm development:
(i) Make local decisions as reliable as possible.
(ii) Reduce the number of decisions made.

Using these as guidelines we devise an algorithm for dependency parsing, prove that it satisfies these principles, and show experimentally that this improves the accuracy of the resulting tree.

Specifically, our approach is based on a shift-reduced parsing as in (Yamada and Matsumoto, 2003). Our general framework provides insights that allow us to improve their algorithm, and to principally justify some of the algorithmic decisions. Specifically, the first principle suggests to improve the reliability of the local predictions, which we do by improving the set of actions taken by the parsing algorithm, and by using a look-ahead search. The second principle is used to justify the control policy of the parsing algorithm – which edges to consider at any point of time. We prove that our control policy is optimal in some sense, and that the decisions we made, guided by these, principles lead to a significant improvement in the accuracy of the resulting parse tree.

1.1 Dependency Parsing and Pipeline Models

Dependency trees provide a syntactic reresentation that encodes functional relationships between words; it is relatively independent of the grammar theory 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), which is one reason for the recent interest in learning these structures (Eisner, 1996; McDonald et al., 2005; Yamada and Matsumoto, 2003; Nivre and Scholz, 2004).

Eisner's work – $O(n^3)$ parsing time generative algorithm – embarked the interest in this area. His model, however, seems to be limited when dealing with complex and long sentences. (McDonald et al., 2005) build on this work, and use a global discriminative training approach to improve the edges' scores, along with Eisner's algorithm, to yield the expected improvement. A different approach was studied by (Yamada and Matsumoto, 2003), that develop a bottom-up approach and learn the parsing decisions between consecutive words in the 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, as was done in other successful parsers, e.g. (Ratnaparkhi, 1997), in that the classifiers are trained on individual decisions rather than on the overall quality of the parser, and chained to yield the global structure. Clearly, it suffers from the limitations of pipeline processing, such as accumulation of errors, but nevertheless, yields very competitive parsing results. A somewhat similar approach was used in (Nivre and Scholz, 2004) to develop a hybrid bottom-up/top-down approach; there, the edges are also labeled with semantic types, yielding lower accuracy than the works mentioned above.

The overall goal of dependency parsing (DP) learning is to infer a tree structure. A common way to do that is to predict with respect to each potential edge (i, j) in the tree, and then choose a global structure that (1) is a tree and that (2) maximizes some score. In the context of DPs, this "edge based factorization method" was proposed by (Eisner, 1996). In other contexts, this is similar to the approach of (Roth and Yih, 2004) in that scoring each edge depends only on the raw data observed and not on the classifications of other edges, and that global considerations can be used to overwrite the local (edge-based) decisions.

On the other hand, 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. However, given that these decisions are noisy, there is a need to devise policies for reducing the number of predictions in order to make the parser more robust. This is exemplified in (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).

In the rest of this paper, we propose and justify a framework for improving pipeline processing based on the principles mentioned above: (i) make local decisions as reliably as possible, and (ii) reduce the number of decisions made. We use the proposed principles to examine the (Yamada and Matsumoto, 2003) parsing algorithm and show that this results in modifying some of the decisions made there and, consequently, better overall dependency trees.

2 Efficient Dependency Parsing

This section describes our DP algorithm and justifies its advantages as a pipeline model. We propose an improved pipeline framework based on the mentioned principles.

For many languages such as English, Chinese and Japanese (with a few exceptions), projective dependency trees (that is, DPs without edge crossings) are sufficient to analyze most sentences. Our work is therefore concerned only with projective trees, which we define below.

For words x, y in the sentence T we introduce the following notations:

$x \rightarrow y$: x is the *direct parent* of y.
$x \rightarrow^* y$: x is an *ancestor* of y;
$x \leftrightarrow y$: $x \rightarrow y$ or $y \rightarrow x$.
$x < y$: x is *to the left* 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 A Pipeline DP Algorithm

Our 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. This is a bottom-up approach that uses machine learning algorithms to learn the parsing decisions (actions) between consecutive words in the sentences. The basic actions used in this model, as in (Yamada and Matsumoto, 2003), are:

S*hift*: there is no relation between a and b, or the action is deferred because the relationship between a and b cannot be determined at this point.

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

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

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. And, clearly, decisions make with respect to a pair of words affect what is considered next by the algorithm.

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*: when the algorithm considers the pair (a, b), $a < b$, we call the word a the current *focus point*.

Next we describe several policies for determining the focus point of the algorithm following an action. We note that, with a few exceptions, determining the focus point does not affect the *correctness* of the algorithm. It is easy to show that for (almost) any focus point chosen, if the correct action is selected for the corresponding edge, the algorithm will eventually yield the correct tree (but may require multiple cycles through the sentence). In practice, the actions selected are 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.

After **S**, the focus point always moves one word to the right. After **L** or **R** there are there natural placement policies to consider:

Start Over: Move focus to the first word in T.

Stay: Move focus 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.

In practice, different placement policies have a significant effect on the number of pairs considered by the algorithm and, therefore, on the final accuracy[1]. The following analysis justifies the **Step Back** policy. We claim that if **Step Back** is used, the algorithm will not waste any action. Thus, it achieves the goal of minimizing the number of actions in pipeline algorithms. Notice that using 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 is a necessary movement to reduce the number of actions.

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) as a function of the focus point placement policy. The statistics are taken with the correct (gold-standard) actions.

It is clear from Table 1 that the policies result

[1]Note that (Yamada and Matsumoto, 2003) mention that they move the focus point back after **R**, but do not state what they do after executing **L** actions, and why. (Yamada, 2006) indicates that they also move focus point back after **L**.

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

```
Let $t$ represents for a word token
For sentence $T = \{t_1, t_2, \dots, t_n\}$
$focus = 1$
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
        $assignParent(t_{focus}, t_{focus+1}, \alpha)$
        $deleteWord(T, focus, \alpha)$
        // performing Step Back here
        $focus = focus - 1$
    else
        $focus = focus + 1$
    end if
end while
```

in very different number of actions and that **Step Back** is the best choice. Note that, since the actions are the gold-standard actions, the policy affects only the number of **S** actions used, and not the **L** and **R** actions, which are a direct function of the correct tree. The number of required actions in the testing stage shows the same trend and the **Step Back** also gives the best dependency accuracy. Algorithm 2 depicts the parsing algorithm.

2.2 Correctness and Pipeline Properties

We can prove two properties of our algorithm. First we show that the algorithm builds the dependency tree in only one pass over the sentence. Then, we show that the algorithm does not waste actions in the sense that it never considers a word pair twice in the same situation. Consequently, this shows that under the assumption of a perfect action predictor, our algorithm makes the smallest possible 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 one can contrive examples in which an algorithm that makes several passes on a sentence can actually make fewer actions than a single pass algorithm. In practice, however, as our experimental data shows, this is unlikely.

Lemma 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 a relation between a and b and the algorithm can perform **L** or **R** actions on that pair when it is 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 in T (not necessary in the original sentence).*
3. *No other word in T is the child of a or b. (a and b are now part of 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 $\{a, b, c\}$ s.t. $a \leftrightarrow b$, $a < c < b$ and $\neg(a \rightarrow c \vee b \rightarrow c)$. However, this violates the properties of a projective language.

Assume $\{a, b, d\}$ are three consecutive words in T. 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** action before the focus point meets (a, b), and will make one of these actions then. It's possible that $(c, a \vee b)$ becomes a free pair after removing a or b in T so we need to move the focus point back. However, 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 completely parsed. □

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

Proof. : We will show that a pair (a, b) will never be considered again given the same situation, that is, when there is no additional information about relations a or b participate in. Note that if **R** or

L is taken, either a or b will become a child word and be eliminate from further consideration by the algorithm. Therefore, if the action taken on (a, b) is **R** or **L**, it will never be considered again.

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. Note that it is possible that there is a relation between a and b, but we cannot perform **R** or **L** now. Therefore, we should consider (a, b) again only if a child of a or b has changed. When *Step Back* is used, we will consider (a, b) again only if the next action is **L**. (If next action is **R**, b will be eliminated.) This is true because the focus point will move back after performing **L**, which implies that b has a new child so we are indeed in a new situation. Since, from Lemma 1, the algorithm only requires one round. we therefore consider (a, b) again only if the situation has changed. □

2.3 Improving the Parsing Action Set

In order to improve the accuracy of the action predictors, we suggest a new (hierarchical) set of actions: *Shift, Left, Right, WaitLeft, WaitRight*. We believe that predicting these is easier due to finer granularity – the **S** action is broken to sub-actions in a natural way.

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.

Thus, we also change the algorithm to perform **S** only if there is no relationship between a and b[2]. The new set of actions is shown to better support our parsing algorithm, when tested on different placement policies. When **W***ait***L***eft* or **W***ait***R***ight* is performed, the focus will move to the next word. 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 more accurate – a more natural class of actions, with a smaller number of candidate actions.

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 five 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 the action classifier, are 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 look ahead inference step described next.

3 A Pipeline Model with Look Ahead

The advantage of a pipeline model is that it can use more information, based on the outcomes of previous predictions. As discussed earlier, this may result in accumulating error. The importance of having a reliable action predictor in a pipeline model motivates the following approach. We devise a look ahead 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 causes a chain reaction and results in further mistakes. This stresses the importance of correct early decisions, and motivates our look ahead policy.

Let (w, x, y, z) be a sentence of four words, and assume that the correct dependency relations are as shown in the top part of Figure 1. If the system mistakenly predicts that x is a child of w before y and z becomes x's children, we can only consider the relationship between w and y in the next stage. Consequently, we will never find the correct parent for y and z. The previous prediction error propagates and impacts future predictions. On the other hand, if the algorithm makes a correct prediction, in the next stage, we do not need to consider w and y. As shown, getting useful rather than misleading information in a pipeline model, requires correct early predictions. Therefore, it is necessary to utilize some inference framework to that may help resolving the error accumulation problem.

In order to improve the accuracy of the action prediction, we might want to examine all possible combinations of action sequences and choose the one that maximizes some score. It is clearly in-

[2]Interestingly, (Yamada and Matsumoto, 2003) mention the possibility of an additional *single* **W***ait* action, but do not add it to the model.

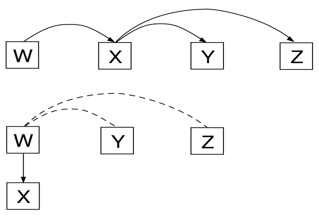

Figure 1: Top figure: the correct dependency relations between w, x, y and z. Bottom figure: if the algorithm mistakenly decides that x is a child of w before deciding that y and z are x's children, we cannot find the correct parent for y and z.

tractable to find the global optimal prediction sequences in a pipeline model of the depth we consider. Therefore, we use a look ahead strategy, implemented via a local search framework, which uses additional information but is still tractable.

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

As our learning algorithm we use a regularized variation of the perceptron update rule, as incorporated in SNoW (Roth, 1998; Carlson et al., 1999), a multi-class classifier that is tailored for large scale learning tasks and has been used successfully in a large number of NLP tasks (e.g., (Punyakanok et al., 2005)). SNoW uses softmax over the raw activation values as its confidence measure, which can be shown to produce a reliable approximation of the labels' conditional probabilities.

The parameter *depth* is to determine the depth of the search procedure. *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 word). Note that *State* changes when a prediction is made and that the features extracted for the action classifier also depend on *State*.

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 selected and performed. Then, the *State* is updated, the new features for the action classifiers are computed and *search* is called again.

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

Algorithm 3 Pseudo code for the look ahead algorithm. $\mathbf{y}$ represents a action sequence. The function *search* considers all possible action sequences with $|depth|$ actions and returns the sequence with the 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)
        $s = s +$ score($\mathbf{y}[i]$, $x$, model)
        TmpState = update(TmpState, $\mathbf{y}[i]$)
    end for
    if $s > maxScore$ then
        $\hat{\mathbf{y}} = \mathbf{y}$
        maxScore = $s$
    end if
end for
return $\hat{\mathbf{y}}$
```

Since the algorithm uses a look ahead policy, it also uses future predictions. The significance of this becomes clear in Section 4.

There are several parameters, in addition to *depth* that can be used to improve the efficiency of the framework. For example, given that the action predictor is a multi-class classifier, we do not need to consider all future possibilities in order to decide the current action. For example, in our experiments, we only consider two actions with highest score at each level (which was shown to produce almost the same accuracy as considering all four actions).

4 Experiments and Results

We use the standard corpus for this task, the Penn Treebank (Marcus et al., 1993). The training set consists of sections 02 to 21 and the testing set is section 23. The POS tags for the evaluation data sets were provided by the tagger of (Toutanova et al., 2003) (which has an accuracy of 97.2% section

23 of the Penn Treebank).

4.1 Features for Action Classification

For each word pair (w_1, w_2) we use the words, their POS tags and also these features of the children of w_1 and w_2. We also include the lexicon and POS tags of 2 words before w_1 and 4 words after w_2 (as in (Yamada and Matsumoto, 2003)). The key additional feature we use, relative to (Yamada and Matsumoto, 2003), is that we include the previous predicted action as a feature. We also add conjunctions of above features to ensure expressiveness of the model. (Yamada and Matsumoto, 2003) makes use of polynomial kernels of degree 2 which is equivalent to using even more conjunctive features. Overall, the average number of active features in an example is about 50.

4.2 Evaluation

We use the same evaluation metrics as in (McDonald et al., 2005). Dependency accuracy (DA) is the proportion of non-root words that are assigned the correct head. Complete accuracy (CA) indicates the fraction of sentences that have a complete correct analysis. We also measure that root accuracy (RA) and leaf accuracy (LA), as in (Yamada and Matsumoto, 2003). When evaluating the result, we exclude the punctuation marks, as done in (McDonald et al., 2005) and (Yamada and Matsumoto, 2003).

4.3 Results

We present the results of several of the experiments that were intended to help us analyze and understand several of the design decisions in our pipeline algorithm.

To see the effect of the additional action, we present in Table 2 a comparison between a system that does not have the *WaitLeft* action (similar to the (Yamada and Matsumoto, 2003) approach) with one that does. In both cases, we do not use the look ahead procedure. Note that, as stated above, the action *WaitRight* is never needed for our parsing algorithm. It is clear that adding *WaitLeft* increases the accuracy significantly.

Table 3 investigates the effect of the look ahead, and presents results with different *depth* parameters (*depth*= 1 means "no search"), showing a consistent trend of improvement.

Table 4 breaks down the results as a function of the sentence length; it is especially noticeable that the system also performs very well for long sentences, another indication for its global performance robustness.

method	DA	RA	CA	LA
w/o *WaitLeft*	90.27	90.73	39.28	93.87
w *WaitLeft*	90.53	90.76	39.74	93.94

Table 2: The significant of the action *WaitLeft*.

method	DA	RA	CA	LA
depth=1	90.53	90.76	39.74	93.94
depth=2	90.67	91.51	40.23	93.96
depth=3	90.69	92.05	40.52	93.94
depth=4	90.79	92.26	40.68	93.95

Table 3: The effect of different *depth* settings.

Table 5 shows the results with three settings of the POS tagger. The best result is, naturally, when we use the gold standard also in testing. However, it is worthwhile noticing that it is better to train with the same POS tagger available in testing, even if its performance is somewhat lower.

Table 6 compares the performances of several of the state of the art dependency parsing systems with ours. When comparing with other dependency parsing systems it is especially worth noticing that our system gives significantly better accuracy on completely parsed sentences.

Interestingly, in the experiments, we allow the parsing algorithm to run many rounds to parse a sentece in the testing stage. However, we found that over 99% sentences can be parsed in a single round. This supports for our justification about the correctness of our model.

5 Further Work and Conclusion

We have addressed the problem of using learned classifiers in a pipeline fashion, where a task is decomposed into several stages and stage classifiers are used sequentially, where each stage may use the outcome of previous stages as its input. This is a common computational strategy in natural language processing and is known to suffer from error accumulation and an inability to correct mistakes in previous stages.

Sent. Len.	DA	RA	CA	LA
<11	93.4	96.7	85.2	94.6
11-20	92.4	93.7	56.1	94.7
21-30	90.4	91.8	32.5	93.4
31-40	90.4	89.8	16.8	94.0
>40	89.7	87.9	8.7	93.3

Table 4: The effect of sentences length. The experiment is done with *depth* $= 4$.

Train-Test	DA	RA	CA	LA
gold−pos	90.7	92.0	40.8	93.8
pos−pos	90.8	92.3	40.7	94.0
gold−gold	92.0	93.9	43.6	95.0

Table 5: Comparing different sources of POS tagging in a pipeline model. We set *depth*$= 4$ in all the experiments of this table.

System	DA	RA	CA	LA
Y&M03	90.3	91.6	38.4	93.5
N&S04	87.3	84.3	30.4	N/A
M&C&P05	90.9	94.2	37.5	N/A
Current Work	90.8	92.3	40.7	94.0

Table 6: The comparison between the current work with other dependency parsing systems.

We abstracted two natural principles, one which calls for making the local classifiers used in the computation more reliable and a second, which suggests to devise the pipeline algorithm in such a way that minimizes the number of decisions (actions) made.

We study this framework in the context of designing a *bottom up dependency parsing*. Not only we manage to use this framework to justify several design decisions, but we also show experimentally that following these results in improving the accuracy of the inferred trees relative to existing models. Interestingly, we can show that the trees produced by our algorithm are relatively good even for long sentences, and that our algorithm is doing especially well when evaluated globally, at a sentence level, where our results are significantly better than those of existing approaches – perhaps showing that the design goals were achieved.

Our future work includes trying to generalize this work to non-projective dependency parsing, as well as attempting to incorporate additional sources of information (e.g., shallow parsing information) into the pipeline process.

6 Acknowledgements

We thank Ryan McDonald for providing the annotated data set and to Vasin Punyakanok for useful comments and suggestions.

This research is supported by the Advanced Research and Development Activity (ARDA)'s Advanced Question Answering for Intelligence (AQUAINT) Program and a DOI grant under the Reflex program.

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.

J. Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proc. the International Conference on Computational Linguistics (COLING)*, 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.

T. Marciniak and M. Strube. 2005. Beyond the pipeline: Discrete optimization in NLP. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, pages 136–143, Ann Arbor, Michigan, June. 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 *Proc. of the Annual Meeting of the ACL*, pages 91–98, Ann Arbor, Michigan.

J. Nivre and M. Scholz. 2004. Deterministic dependency parsing of english text. In *COLING2004*, pages 64–70.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *IWPT*, Nancy, France.

V. Punyakanok, D. Roth, and W. Yih. 2005. The necessity of syntactic parsing for semantic role labeling. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1117–1123.

A. Ratnaparkhi. 1997. A linear observed time statistical parser based on maximum entropy models. In *EMNLP-97, The Second Conference on Empirical Methods in Natural Language Processing*, pages 1–10.

D. Roth and W. Yih. 2004. A linear programming formulation for global inference in natural language tasks. In Hwee Tou Ng and Ellen Riloff, editors, *Proc. of the Annual Conference on Computational Natural Language Learning (CoNLL)*, pages 1–8. Association for Computational Linguistics.

D. Roth. 1998. Learning to resolve natural language ambiguities: A unified approach. In *Proc. National Conference on Artificial Intelligence*, pages 806–813.

K. Toutanova, D. Klein, and C. Manning. "2003". Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of HLT-NAACL 03*.

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

H. Yamada. 2006. Private communication.