Peking: Profiling Syntactic Tree Parsing Techniques for Semantic Graph Parsing

Yantao Du, Fan Zhang, Weiwei Sun and **Xiaojun Wan**
Institute of Computer Science and Technology, Peking University
The MOE Key Laboratory of Computational Linguistics, Peking University
{duyantao,ws,wanxiaojun}@pku.edu.cn, zhangf717@gmail.com

Abstract

Using the SemEval-2014 Task 8 data, we profile the syntactic tree parsing techniques for semantic graph parsing. In particular, we implement different transition-based and graph-based models, as well as a parser ensembler, and evaluate their effectiveness for semantic dependency parsing. Evaluation gauges how successful data-driven dependency graph parsing can be by applying existing techniques.

1 Introduction

Bi-lexical dependency representation is quite powerful and popular to encode syntactic or semantic information, and parsing techniques under the dependency formalism have been well studied and advanced in the last decade. The major focus is limited to tree structures, which fortunately correspond to many computationally good properties. On the other hand, some leading linguistic theories argue that more general graphs are needed to encode a wide variety of deep syntactic and semantic phenomena, e.g. topicalization, relative clauses, etc. However, algorithms for statistical graph spanning have not been well explored before, and therefore it is not very clear how good data-driven parsing techniques developed for tree parsing can be for graph generating.

Following several well-established syntactic theories, SemEval-2014 task 8 (Oepen et al., 2014) proposes using graphs to represent semantics. Considering that semantic dependency parsing is a quite new topic and there is little previous work, we think it worth appropriately profiling successful tree parsing techniques for graph parsing. To this end, we build a hybrid system that combines several important data-driven parsing techniques and evaluate their impact with the given data. In particular, we implement different transition-based and graph-based models, as well as a parser ensembler.

Our experiments highlight the following facts:

- Graph-based models are more effective than transition-based models.
- Parser ensemble is very useful to boost the parsing accuracy.

2 Architecture

We explore two kinds of basic models: One is transition-based, and the other is tree approximation. Transition-based models are widely used for dependency tree parsing, and they can be adapted to graph parsing (Sagae and Tsujii, 2008; Titov et al., 2009). Here we implement 5 transition-based models for dependency graph parsing, each of which is based on different transition system.

The motivation of developing tree approximation models is to apply existing graph-based tree parsers to generate graphs. At the training time, we convert the dependency graphs from the training data into dependency trees, and train second-order arc-factored models[1]. At the test phase, we parse sentences using this tree parser, and convert the output trees back into semantic graphs. We think tree approximation can appropriately evaluate the possible effectiveness of graph-based models for graph spanning.

Finally, we integrate the outputs of different models with a simple voter to boost the performance. The motivation of using system combination and the choice of voting is mainly due to the experiments presented by (Surdeanu and Manning, 2010). When we obtain all the outputs of

[1]The mate parser (code.google.com/p/mate-tools/) is used.

these models, we combine them into a final result, which is better than any of them. For combination, we explore various systems for this task, since empirically we know that variety leads to better performance.

3 Transition-Based Models

Transition-based models are usually used for dependency tree parsing. For this task, we exploit it for dependency graph parsing.

A transition system S contains a set $\mathcal{C}$ of configurations and a set $\mathcal{T}$ of transitions. A configuration $c \in \mathcal{C}$ generally contains a stack σ of nodes, a buffer β of nodes, and a set A of arcs. The elements in A is in the form (x, l, y), which denotes a arc from x to y labeled l. A transition $t \in \mathcal{T}$ can be applied to a configuration and turn it into a new one by adding new arcs or manipulating elements of the stack or the buffer. A statistical transition-based parser leverages a classifier to approximate an oracle that is able to generate target graphs by transforming the initial configuration $c_s(x)$ into a terminal configuration $c_t \in \mathcal{C}_t$.

An oracle of a given graph on sentence x is a sequence of transitions which transform the initial configuration to the terminal configuration the arc set A_{c_t} of which is the set of the arcs of the graph.

3.1 Our Transition Systems

We implemented 5 different transition systems for graph parsing. Here we describe two of them in detail, one is the Titov system proposed in (Titov et al., 2009), and the other is our Naive system. The configurations of the two systems each contain a stack σ, a buffer β, and a set A of arcs, denoted by $\langle \sigma, \beta, A \rangle$. The initial configuration of a sentence $x = w_1 w_2 \cdots w_n$ is $c_s(x) = \langle [0], [1, 2, \cdots, n], \{\} \rangle$, and the terminal configuration set $\mathcal{C}_t$ is the set of all configurations with empty buffer. These two transition systems are shown in 1.

The transitions of the Titov system are:

- LEFT-ARC$_l$ adds an arc from the front of the buffer to the top of the stack, labeled l, into A.
- RIGHT-ARC$_l$ adds an arc from the top of the stack to the front of the buffer, labeled l, into A.
- SHIFT removes the front of the buffer and push it onto the stack;
- POP pops the top of the stack.
- SWAP swaps the top two elements of the stack.

This system uses a transition SWAP to change the node order in the stack, thus allowing some crossing arcs to be built.

The transitions of the Naive system are similar to the Titov system's, except that we can directly manipulate all the nodes in the stack instead of just the top two. In this case, the transition SWAP is not neeeded.

The Titov system can cover a great proportion, though not all, of graphs in this task. For more discussion, see (Titov et al., 2009). The Naive system, by comparison, covers all graphs. That is to say, with this system, we can find an oracle for any dependency graph on a sentence x. Other transition systems we build are also designed for dependency graph parsing, and they can cover dependency graphs without self loop as well.

3.2 Statistical Disambiguation

First of all, we derive oracle transition sequences for every sentence, and train *Passive-Aggressive* models (Crammer et al., 2006) to predict next transition given a configuration. When it comes to parsing, we start with the initial configuration, predicting next transition and updating the configuration with the transition iteratively. And finally we will get a terminal configuration, we then stop and output the arcs of the graph contained in the final configuration.

We extracted rich feature for we utilize a set of rich features for disambiguation, referencing to Zhang and Nivre (2011). We examine the several tops of the stack and the one or more fronts of the buffer, and combine the lemmas and POS tags of them in many ways as the features. Additionally, we also derive features from partial parses such as heads and dependents of these nodes.

3.3 Sentence Reversal

Reversing the order the words of a given sentence is a simple way to yield heterogeneous parsing models, thus improving parsing accuracy of the model ensemble (Sagae, 2007). In our experiments, one transition system produces two models, one trained on the normal corpus, and the other on the corpus of reversed sentences. Therefore we can get 10 parse of a sentence based on 5 transition systems.

LEFT-ARC$_l$	$(\sigma\|i,j\|\beta,A) \Rightarrow (\sigma\|i,j\|\beta,A \cup \{(j,l,i)\})$
RIGHT-ARC$_l$	$(\sigma\|i,j\|\beta,A) \Rightarrow (\sigma\|i,j\|\beta,A \cup \{(i,l,j)\})$
SHIFT	$(\sigma,j\|\beta,A) \Rightarrow (\sigma\|j,\beta,A)$
POP	$(\sigma\|i,\beta,A) \Rightarrow (\sigma,\beta,A)$
SWAP	$(\sigma\|i\|j,\beta,A) \Rightarrow (\sigma\|j\|i,\beta,A)$

Titov System

LEFT-ARC$_l^k$	$(\sigma\|i_k\|\ldots\|i_2\|i_1,j\|\beta,A) \Rightarrow (\sigma\|i_k\|\ldots\|i_2\|i_1,j\|\beta,A \cup \{(j,l,i_k)\})$
RIGHT-ARC$_l^k$	$(\sigma\|i_k\|\ldots\|i_2\|i_1,j\|\beta,A) \Rightarrow (\sigma\|i_k\|\ldots\|i_2\|i_1,j\|\beta,A \cup \{(i_k,l,j)\})$
SHIFT	$(\sigma,j\|\beta,A) \Rightarrow (\sigma\|j,\beta,A)$
POP^k	$(\sigma\|i_k\|i_{k-1}\|\ldots\|i_2\|i_1,\beta,A) \Rightarrow (\sigma\|i_{k-1}\|\ldots\|i_2\|i_1,\beta,A)$

Naive System

Figure 1: Two of our transition systems.

4 Tree Approximation Models

Parsing based on graph spanning is quite challenging since computational properties of the semantic graphs given by the shared task are less explored and thus still unknown. On the other hand, finding the best higher-order spanning for general graph is NP complete, and therefore it is not easy, if not impossible, to implement arc-factored models with exact inference. In our work, we use a practical idea to indirectly profile the graph-based parsing techniques for dependency graph parsing. Inspired by the PCFG approximation idea (Fowler and Penn, 2010; Zhang and Krieger, 2011) for deep parsing, we study tree approximation approaches for graph spanning.

This tree approximation technique can be applied to both transition-based and graph-based parsers. However, since transition systems that can directly handle build graphs have been developed, we only use this technique to evaluate the possible effectiveness of graph-based models for semantic parsing.

4.1 Graph-to-Tree Transformation

In particular, we develop different methods to convert a semantic graph into a tree, and use edge labels to encode dependency relations as well as structural information which helps to transform a converted tree back to its original graph. By the graph-to-tree transformation, we can train a tree parser with a graph-annotated corpus, and utilize the corresponding tree-to-graph transformation to generate target graphs from the outputs of the tree parser. Given that the tree-to-graph transformation is quite trivial, we only describe the graph-to-tree transformation approach.

We use graph traversal algorithms to convert a directed graph to a directed tree. The transformation implies that we may lose, add or modify some dependency relations in order to make the graph a tree.

4.2 Auxiliary Labels

In the transformed trees, we use auxiliary labels to carry out information of the original graphs. To encode multiple edges to one, we keep the original label on the directed edge but may add other edges' information. On the other hand, throughout most transformations, some edges must be reversed to make a tree, so we need a symbol to indicate a edge on the tree is reversed during transformation. The auxiliary labels are listed below:

- Label with following ∼R: The symbol ∼R means this directed edge is reversed from the original directed graph.
- Separator: Semicolon separates two encoded original edges.
- $[N]$ followed by label: The symbol $[N]$ (N is an integer) represents the head of the edge. The dependent is the current one, but the head is the dependent's N-th ancestor where 1st ancestor is its father and 2nd ancestor is its father's father.

See Figure 2 for example.

4.3 Traversal Strategies

Given directed graph (V, E), the task is to traverse all edges on the graph and decide how to change the labels or not contain the edge on the output. We use 3 strategies for traversal. Here we use $x \rightarrow_g y$ to denote the edge on graph, and $x \rightarrow_t y$ the edge on tree.

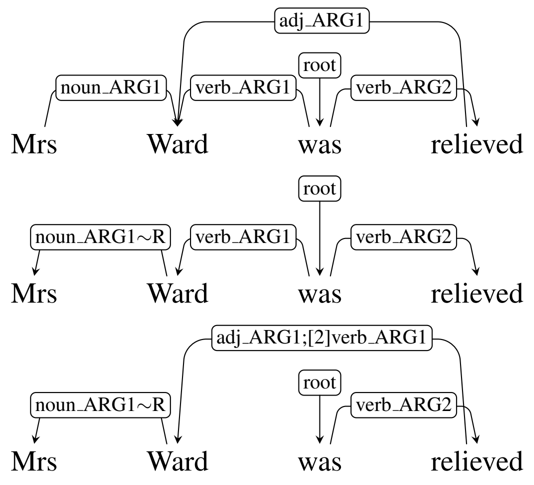

Figure 2: One dependency graph and two possible dependency trees after converting.

Depth-first-search We try graph traversal by depth-first-search starting from the root on the directed graph ignoring the direction of edges. During the traversal, we add edges to the directed tree with (perhaps new) labels. We traverse the graph recursively. Suppose the depth-first-search is running at the node x and the nodes set A which have been searched. And suppose we find node y is linked to x on the graph ($x \rightarrow_g y$ or $y \rightarrow_g x$). If $y \notin A$, we add the directed edge $x \rightarrow_t y$ to the tree immediately. In the case of $y \rightarrow_g x$, we add ∼R to the edge label. If $y \in A$, then y must be one of the ancestors of x. In this case, we add this information to the label of the existing edge $z \rightarrow_t x$. Since the distance between two nodes x and y is sufficient to indicate the node y, we use the distance to represent the head or dependent of this directed edge and add the label and the distance to the label of $z \rightarrow_t x$. It is clear that the auxiliary label $[N]$ can be used for multiple edge encoding. Under this strategy, all edges can be encoded on the tree.

Breadth-first-search An alternative traversal strategy is based on breadth-first-search starting from the root. This search ignores the direction of edge too. We regard the search tree as the dependency tree. During the breadth-first-search, if (x, l, y) exists but node y has been searched, we just ignore the edge. Under this strategy, we may lose some edges.

Iterative expanding This strategy is based on depth-first-search but slightly different. The strategy only searches through the forward edges on the directed graph at first. When there is no forward edge to expend, a traversed node linked to some nodes that are not traversed must be the dependent of them. Then we choose an edge and add it (reversed) to the tree and continue to expand the tree. Also, we ignore the edges that does not satisfy the tree constraint. We call this strategy iterative expanding. When we need to expand output tree, we need to design a strategy to decide which edge to be add. The measure to decide which node should be expanded first is its possible location on the tree and the number of nodes it can search during depth-first-search. Intuitively, we want the reversed edges to be as few as possible. For this purpose, this strategy is practical but not necessarily the best. Like the Breadth-first-search strategy, this strategy may also cause edge loss.

4.4 Forest-to-Tree

After a primary searching process, if there is still edge $x \rightarrow_g y$ that has not been searched yet, we start a new search procedure from x or y. Eventually, we obtain a forest rather than a tree. To combine disconnected trees in this forest to the final dependency tree, we use edges with label *None* to link them. Let the node set W be the set of roots of the trees in the forest, which are not connected to original graph root. The mission is to assign a node $v \notin W$ for each $w \in W$. If we assign v_i for w_i, we add the edge $v_i \rightarrow w_i$ labeled by *None* to the final dependency tree. We try 3 strategies in this step:

- For each $w \in W$ we look for the first node $v \notin W$ on the left of w.
- For each $w \in W$ we look for the first node $v \notin W$ on the right of w.
- By defining the distance between two nodes as how many words are there between the two words, we can select the *nearest* node. If the distances of more than one node are equal, we choose v randomly.

We also tried to link all of the nodes in W directly to the root, but it does not work well.

5 Model Ensemble

We have 19 heterogeneous basic models (10 transition-based models, 9 tree approximation models), and use a simple voter to combine their outputs.

Algorithm	DM	PAS	PCEDT
DFS	0	0	0
BFS	0.0117	0.0320	0.0328
FEF	0.0127	0.0380	0.0328

Table 1: Edge loss of transformation algorithms.

For each pair of words of a sentence, we count the number of the models that give positive predictions. If the number is greater than a threshold, we put this arc to the final graph, and label the arc with the most common label of what the models give.

Furthermore, we find that the performance of the tree approximation models is better than the transition based models, and therefore we take weights of individual models too. Instead of just counting, we sum the weights of the models that give positive predictions. The tree approximation models are assigned higher weights.

6 Experiments

There are 3 subtasks in the task, namely DM, PAS, and PCEDT. For subtask DM, we finally obtained 19 models, just as stated in previous sections. For subtask PAS and PCEDT, only 17 models are trained due to the tight schedule.

The tree approximation algorithms may cause some edge loss, and the statistics are shown in Table 1. We can see that DFS does not cause edge loss, but edge losses of other two algorithm are not negligible. This may result in a lower recall and higher precision, but we can tune the final results during model ensemble. Edge loss in subtask DM is less than those in subtask PAS and PCEDT.

We present the performance of several representative models in Table 2. We can see that the tree approximation models performs better than the transition-based models, which highlights the effective of arc-factored models for semantic dependency parsing. For model ensemble, besides the accuracy of each single model, it is also important that the models to be ensembled are very different. As shown in Table 2, the evaluation between some of our models indicates that our models do vary a lot.

Following the suggestion of the task organizers, we use section 20 of the train data as the development set. With the help of development set, we tune the parameters of the models and ensembling. We set the weight of each transition-based model 1, and tree approximation model 2 in run 1, 3 in run 2. The threshold is set to a half of the total weight. The final results given by the organizers are shown in Table 3. Compared to Table 2 demonstrates the effectiveness of parser ensemble.

Models	DM	PAS	PCEDT
Titov	0.8468	0.8754	0.6978
Titov$_r$	0.8535	0.8928	0.7063
Naive	0.8481	-	-
DFS$_n$	0.8692	0.9034	0.7370
DFS$_l$	0.8692	0.9015	0.7246
BFS$_n$	0.8686	0.8818	0.7247
Titov vs. Titov$_r$	0.8607	0.8831	0.7613
Titov vs. Naive	0.9245	-	-
Titov vs. DFS$_n$	0.8590	0.8865	0.7650
DFS$_n$ vs. DFS$_l$	0.9273	0.9579	0.8688
DFS$_n$ vs. BFS$_n$	0.9226	0.9169	0.8367

Table 2: Evaluation between some of our models. Labeled f-score on test set is shown. Titov$_r$ stands for reversed Titov, DFS$_n$ for DFS+nearest, DFS$_l$ for DFS+left, and BFS$_n$ for BFS+nearest. The upper part gives the performance, and the lower part gives the agreement between systems.

Format	LP	LR	LF	LM
DM	0.9027	0.8854	0.8940	0.2982
PAS	0.9344	0.9069	0.9204	0.3872
PCEDT	0.7875	0.7396	0.7628	0.1120

Table 3: Final results of the ensembled model.

7 Conclusion

Data-driven dependency parsing techniques have been greatly advanced during the parst decade. Two dominant approaches, i.e. transition-based and graph-based methods, have been well studied. In addition, parser ensemble has been shown very effective to take advantages to combine the strengthes of heterogeneous base parsers. In this work, we propose different models to profile the three techniques for semantic dependency parsing. The experimental results suggest several directions for future study.

Acknowledgement

The work was supported by NSFC (61300064, 61170166 and 61331011) and National High-Tech R&D Program (2012AA011101).

References

Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. 2006. Online passive-aggressive algorithms. *JOURNAL OF MACHINE LEARNING RESEARCH*, 7:551–585.

Timothy A. D. Fowler and Gerald Penn. 2010. Accurate context-free parsing with combinatory categorial grammar. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, Uppsala, Sweden, July.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Dan Flickinger, Jan Hajič, Angelina Ivanova, and Yi Zhang. 2014. SemEval 2014 Task 8: Broad-coverage semantic dependency parsing. In *Proceedings of the 8th International Workshop on Semantic Evaluation*, Dublin, Ireland.

Kenji Sagae and Jun'ichi Tsujii. 2008. Shift-reduce dependency DAG parsing. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 753–760, Manchester, UK, August. Coling 2008 Organizing Committee.

Kenji Sagae. 2007. Dependency parsing and domain adaptation with lr models and parser ensembles. In *In Proceedings of the Eleventh Conference on Computational Natural Language Learning*.

Mihai Surdeanu and Christopher D. Manning. 2010. Ensemble models for dependency parsing: Cheap and good? In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, Los Angeles, California, June.

Ivan Titov, James Henderson, Paola Merlo, and Gabriele Musillo. 2009. Online graph planarisation for synchronous parsing of semantic and syntactic dependencies. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 1562–1567, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Yi Zhang and Hans-Ulrich Krieger. 2011. Large-scale corpus-driven PCFG approximation of an hpsg. In *Proceedings of the 12th International Conference on Parsing Technologies*, Dublin, Ireland, October.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Portland, Oregon, USA, June.