Implementation of the Dependency Parser using Spanning Trees Algorithms

Chen Huang
School of Information
chhuang@umich.edu
April 15, 2008

Abstract
In this paper, I analyzed a dependency parsing method by Ryan McDonald and etc. [3]. Then I implement the parser and tested against the corpus provided by the [1]. The implementation was able to correctly identify 75% of all the dependency relations. This is the almost the same with [3]'s results.

1 Introduction

Compared with lexicalized phrase structures, Dependency representation of natural language syntactic structure is simpler and more efficient to learn and parse while still encoding much of the predicate-argument information needed in applications. Thus, dependency parsing is applied more recently in applications such as relation extraction, machine translation, synonym generation and lexical resource augmentation [5]. In [3], an new dependency parsing approach has been proposed, by transforming the weighted dependency parsing to search for maximum spanning trees in directed graphs: consider a sentence as a dependency graph, in which each word represent a node, edges are dependency relation, the algorithm then return the max spanning tree of the graph as the dependency structure.

I reimplement the approach to see if my result would be comparable to the original and to offer a Perl version of the open source tool for dependency parsing research.

2 Background

2.1 Dependency representations

Dependency representations is the representation of words linked to their arguments. Figure 1 shows a dependency tree for the sentence "John hit the ball with the bat" [3]. The representation is a tree, as each word depends on exactly one parent, either another word or the root. The example tree is projective, meaning that there is no crossing of edges in the tree. Trees with crossing edges
are regarded as non-projective tree, which is used in languages with more flexible word order such as German. Note that the example is untyped, means there is no grammatical type on the edge. The method I implemented is for non-projective parsing and untyped dependency tree.

3 Dependency Parsing and Spanning Trees Algorithm

The intuition is to use maximum spanning tree algorithm to find the dependency tree in a weighted graph. The weighted graph is a complete weighted graph represent a sentence, in which nodes represent words and edges are the possible dependency relation between words. The method then find the maximum spanning tree in the graph and return it as the non-projective dependency tree [3].

3.1 Edge based factorization

Weights on edges is calculated as follow [3]: suppose a sentence \( x \) can be represented as \( x = x_1x_2...x_n \), in which \((i, j)\) stands for the relation between word \( x_i \) and \( x_j \). Suppose \( y \) represent the dependency tree found in the graph. The score of a dependency tree is regarded as the sum of the score of all edges of that tree. If we represent all edges in \( y \) as a vector, then the score for edge \((i, j)\) will be

\[
s(i, j) = w \ast f(i, j)
\]

, where \( f(i, j) \) is the feature vector, contain all possible features edge \((i, j)\) represent and each feature in \( f(i, j) \) has value 1, if \((i, j)\) exists. Thus, the score for \( y \) is:

\[
s(x, y) = \sum_{(i,j)\in y} s(i, j) = \sum_{(i,j)\in y} w \ast f(i, j)
\]

, the weight vector \( w \) can calculated using online learning algorithm [4], we assume \( w \) is known in the following. Then the task transform to find the feature tree \( y \) with the largest score.

3.2 Maximum spanning tree algorithm

Now suppose the graph representation of sentence , then the directed graph constitutes nodes \( x_1, x_2, ..., x_n \), edges \((i, j)\), \( i, j \in x \). Define the graph \( G_x = (V_x, E_x) \) as follows [3]:

\[
V_x = \{x|x \in \{\text{root}, x_1, ..., x_n\}\}
E_x = \{(i, j)|i \neq j, (i, j) \in [0 : n] \ast [1 : n]\}
\]

Figure 1: An example dependency tree[3]
In directed graph $G_x$, vertices’s represent words in the sentence, with an extra root as the pointer for the dependency tree, directed and weighted edges represent the dependency relation between every two words. Then tree with the highest score is the maximum spanning tree. The maximum spanning tree can be found using Chu-Liu-Edmond algorithm [3, 6].

4 Detailed Description

The whole parsing process has two parts: the online learning of the parsing model and the dependency parsing. Figure 2 displays the framework [5]. The learning algorithm reads in annotated corpus to produce the parsing model, constituted with all extracted features and their related weights. The parser then extracts features from input data and calculates the score for every possible relation between words. The inference algorithm returns the dependency tree with the maximum sum of score of edges. For this term project, I implement the parsing part, which includes the feature extraction from sentences and the Chu-Liu-Edmond maximum spanning tree algorithm. The parser is written in Perl and is currently made up of 1208 lines of codes.

4.1 Feature extraction

The suitable feature representation $f(i, j)$ for each edge is crucial to the accuracy of parsing. [3] used four types of features. Table 1 outlined the feature sets used in the parser [3]. The first feature set is used to represent single word and its pos tag. The second feature set is used to represent the dependency relation between the two-word pair. The third and fourth features take into consideration of the context. The third feature set will record the pos tag occurred between the examined word pair and the fourth feature set record the surrounding pos tags. All features are combined with the direction of the relation and the distance between. In practical, the parser also considered the very specific features (the feature containing the actual word appeared in the training set) and less specific features (feature containing the pos of the word and feature containing the initial of the actual word) [4]. For words with a length larger than 5, the parser consider the feature with the whole word and 5-gram prefix by stem them. Appendix A give a complete list of features for a relation.
4.2 Constructing the weighted directed graph

After the feature list for each edge is extracted, the score of each edge is calculated from a trained model. The trained model contains all possible features with related weights from the trained set. The weight is calculated with an online learning algorithm such that each trained sentence in the training set get the highest score for its correct dependency tree [4]. In order to calculate the score for each edge, the extracted features are matched with features in the trained model. For an edge, the sum of all its features' weights is the score of the edge. For a sentence with \( n \) length, \( n(n-1) \) relations’ scores are calculated, this formed a weighted completed graph with \( n(n-1) \) edges.

4.3 Finding the maximum spanning tree

![Chu-Liu-Edmonds algorithm](image)

**Chu-Liu-Edmonds**

Graph \( G = (V, E) \)

Edge weight function \( s : E \to \mathbb{R} \)

1. Let \( M = \{(x^*, x) : x \in V, x^* = \arg \max_{x'} s(x', x)\} \)
2. Let \( G_M = (V, M) \)
3. If \( G_M \) has no cycles, then it is an MST: return \( G_M \)
4. Otherwise, find a cycle \( C \) in \( G_M \)
5. Let \( G_C = \text{contract}(G, C, s) \)
6. Let \( y = \text{Chu-Liu-Edmonds}(G_C, s) \)
7. Find a vertex \( x \in C \) s.t. \( x', x) \in y \)
8. return \( y \cup C - \{(x', x)\} \)

**contract**

1. Let \( G_C \) be the subgraph of \( G \) excluding nodes in \( C \)
2. Add a node \( c \) to \( G_C \) representing cycle \( C \)
3. For \( x \in V - C : \exists x' \in C \) s.t. \( x', x) \in E \)
   Add edge \( (c, x) \) to \( G_C \) with \( s(c, x) = \max_{x' \in C} s(x', x) \)
4. For \( x \in V - C \)
   Add edge \( (x, c) \) to \( G_C \) with \( s(x, c) = \max_{x' \in C} [s(x, x') - s(a(x'), x') + s(C)] \)
   where \( a(v) \) is the predecessor of \( v \) in \( C \)
   and \( s(C) = \sum_{v \in C} s(a(v), v) \)
5. return \( G_C \)

Figure 3: Chu-Liu-Edmond algorithm for finding the maximum spanning tree[5]

Given the weighted directed graph, the highest scoring non-projective tree
can be calculated by finding the maximum spanning tree in the graph. Many algorithms are applicable for undirected graphs [2]. For directed graphs, the parser used Chu-Liu-Edmond algorithm [6]. Figure 3 gives a outline of the algorithm [3]. The algorithm is a greedy algorithm, by starting with each node greedily search the highest score incoming edge. Then it check if there is a tree exists. If a tree exists, it's the maximum spanning tree. If not, it contracts the cycle into a new node, lock all nodes in the cycle, then calculating the weight for new edges connecting the new node and all unlocked nodes. To calculate the weights of new edges, the maximum score between nodes in the cycle and unlocked nodes are selected and the node in the cycle is recorded. The algorithm then recursively find tree in the new graph till a tree exists. By break the cycle into tree in each recurrence, the maximum spanning tree will be returned. The cycle in a graph can be found using depth frist search. The algorithm is proved to be $O(n^2)$ in the average case [4]. In practical situation, there are times when a parsed tree don’t have the “root” node as the “root” of the tree, thus the algorithm need to be modified slightly by substituting one node’s parent node to “root”, the node with the maximum incoming edge from “root” is modified, by substituting it’s parent node’s parent to “root”.

5 Results

On corpus provided by Tree-bank trees using the conversion described by [1], The implementation achieved the same result with the author:

<table>
<thead>
<tr>
<th>Corrected tokens:</th>
<th>Ryan’s Result</th>
<th>My Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3502/4639</td>
<td>3479/4639</td>
</tr>
<tr>
<td>Corrected rate:</td>
<td>75.5%</td>
<td>75.0%</td>
</tr>
<tr>
<td>Corrected Sentences:</td>
<td>21/200</td>
<td>20/200</td>
</tr>
<tr>
<td>Corrected rate:</td>
<td>10.5%</td>
<td>10%</td>
</tr>
</tbody>
</table>

The result is slightly off from the authors’ result, this is because the authors had done some substitution when processing data (e.g. change 11.11 to $<\text{num}>$), the result will be the same if similar substitutions are made.

6 Conclusion

The approach used in this paper achieved respectable results. The result enforced the idea that using maximum spanning tree algorithm is an effective and efficient way of parsing dependency relations. Below are some of the observations and possible future directions.

For every new sentence, a lot of features are extracted, but only a small portion features can be found in the trained model (approximately less than 40%), most of the unused features are very specific features, the speed of the parser might improved by using more abstracted feature representation.

The feature extracted didn’t represent all possible information contained in dependency relations, other information like sentence length or word length may be helpful for the accuracy.

The sentences’ correction rate is relatively low, most sentences are wrong in only one word, future work could explore other ways to calculate the weight for
features, such as use the frequency of the feature's occurrence.
Finally, performance on other types of text than [1] could be evaluated.
The source code is downloadable at http://www.umich.edu/~chhuang/MSTParser.tar

7 Appendix

A Possible Features for dependency relation

```perl
# feature posR posMid posL
for ($i = $small+1; $i < $large; $i++) {
    $allPos = "$pos[$small] $pos[$i] $pos[$large]";
    $feature = "PC=$allPos$dirDist";
    $feature = "1PC=$allPos";
    $feature = "XPC=$allPosA$dirDist";
    $feature = "X1PC=$allPosA";
}

# feature posL-1 posL posR posR+1
$feature = "PT=$pLeft $pos[$small] $pos[$large] $pRight$dirDist";
$feature = "PT1=$pos[$small] $pos[$large] $pRight$dirDist";
$feature = "PT2=$pLeft $pos[$small] $pos[$large]$dirDist";
$feature = "PT3=$pLeft $pos[$large] $pRight$dirDist";
$feature = "PT4=$pLeft $pos[$small] $pRight$dirDist";
$feature = "1PT=$pLeft $pos[$small] $pos[$large] $pRight";
$feature = "1PT1=$pos[$small] $pos[$large] $pRight";
$feature = "1PT2=$pLeft $pos[$small] $pos[$large]";
$feature = "1PT3=$pLeft $pos[$large] $pRight";
$feature = "1PT4=$pLeft $pos[$small] $pRight";
```

$feature = "XPT3=$pLeftA $posA[$large] $pRightA$dirDist";
$feature = "XPT4=$pLeftA $posA[$small] $pRightA$dirDist";
$feature = "X1PT2=$pLeftA $posA[$small] $posA[$large]";
$feature = "X1PT3=$pLeftA $posA[$large] $pRightA";
$feature = "X1PT4=$pLeftA $posA[$small] $pRightA";

# feature posL posL+1 posR-1 posR

$feature = "1APT1=$posA[$small] $pRightLeft $posA[$large]";
$feature = "1APT2=$posA[$small] $pLeftRight $posA[$large]";
$feature = "1APT3=$pLeftRight $pRightLeftA $posA[$large]dirDist";
$feature = "1APT4=$posA[$small] $pLeftRight $pRightLeft$dirDist";


$feature = "X1APT1=$posA[$small] $pRightLeftA $posA[$large]";
$feature = "X1APT3=$pLeftRightA $pRightLeftA $posA[$large]";

# also consider the actual words
if ($direction eq "RA") {
    $head = $tokens[$small];
    $headP = $pos[$small];
    $child = $tokens[$large];
    $childP = $pos[$large];
} else {
    $head = $tokens[$large];
    $headP = $pos[$large];
    $child = $tokens[$small];
$childP = $pos[$small];
}
# Print $head, "|", $headP, "|", $child, "|", $childP, "\n";
$all = "$head $headP $child $childP";
$hPos = "$headP $child $childP";
$cPos = "$head $headP $childP";
$hP = "$headP $child";
$cP = "$head $childP";
$oPos = "$headP $childP";
$oLex = "$head $child";

# Unigram features
$feature = "A=$all$dirDist";
$feature = "B=$hPos$dirDist";
$feature = "C=$cPos$dirDist";
$feature = "D=$hP$dirDist";
$feature = "E=$cP$dirDist";
$feature = "F=$oLex$dirDist";
$feature = "G=$oPos$dirDist";
$feature = "H=$head $headP$dirDist";
$feature = "I=$headP$dirDist";
$feature = "J=$head$dirDist";
$feature = "K=$child $childP$dirDist";
$feature = "L=$childP$dirDist";
$feature = "M=$child$dirDist";

# Unigram features without direction and distance information
$feature = "AA=$all";
$feature = "BB=$hPos";
$feature = "CC=$cPos";
$feature = "DD=$hP";
$feature = "EE=$cP";
$feature = "FF=$oLex";
$feature = "GG=$oPos";
$feature = "HH=$head $headP";
$feature = "II=$headP";
$feature = "JJ=$head";
$feature = "KK=$child $childP";
$feature = "LL=$childP";
$feature = "MM=$child";

# 5-gram stuff
if (length $head > 5 || length $child > 5) {
   $hL = length $head;
   $cL = length $child;
   $head = substr $head, 0, 5 if $hL > 5;
   $child = substr $child, 0, 5 if $cL > 5;
   $all = "$head $headP $child $childP";
   $hPos = "$headP $child $childP";
   $cPos = "$head $headP $childP";
   $hP = "$headP $child";
   $cP = "$head $childP";
   $oPos = "$headP $childP";
}
$oLex = "$head $child$;  
# S stands for short
$feature = "SA=$all$dirDist$; 
$feature = "SF=$oLex$dirDist$;  
$feature = "SAA=$all$;  
$feature = "SFF=$oLex$;  
if ($cL > 5) {  
  $feature = "SB=$hPos$dirDist$;  
  $feature = "SD=$hP$dirDist$;  
  $feature = "SK=$child $childP$dirDist$;  
  $feature = "SM=$child$dirDist$;  
  $feature = "SBB=$hPos$;  
  $feature = "SDD=$hP";  
  $feature = "SKK=$child $childP$;  
  $feature = "SMM=$child$;  
}  
if ($hL > 5) {  
  $feature = "SC=$cPos$dirDist$;  
  $feature = "SE=$cP$dirDist$;  
  $feature = "SH=$head $headP$dirDist$;  
  $feature = "SJ=$head$dirDist$;  
  $feature = "SCC=$cPos$;  
  $feature = "SEE=$cP$;  
  $feature = "SHH=$head $headP$;  
  $feature = "SJJ=$head$;  
}  
}

B Source Code

The source code is available at http://www.umich.edu/~chhuang/MSTParser.tar

References


[4] Ryan McDonald, Koby Crammer and Fernando Pereira Online Large-Margin Training of Dependency Parsers Association for Computational Linguistics (ACL), 2005