Determinising the output of dependency parser by extending grammar rules with weights

Jacek Maciejewski Rafal Rzepka Kenji Araki
Graduate School of Information Science and Technology, Hokkaido University
{yaci, kabura, araki}@media.eng.hokudai.ac.jp

Abstract

This paper describes research in improving DGP (Dependency Graph Parser). DGP is one of the most efficient dependency parser for Polish today. However, it is non-deterministic which makes it insufficient when numerous graphs are proposed for one sentence. In our research we aim to overcome this problem by adding weights to grammar rules and thus making possible to rank the output trees and choose the most appropriate one.

1 Introduction

Automatic syntactic analysis plays a vital role in numerous applications such as search engines, machine translation systems, intelligent chatbots, etc. Dependency parsing is widely recognized as one of the best approaches for performing this difficult task. Dependency-based methods have become increasingly popular in the field of natural language processing in recent decade (Nivre, 2005). One of the reasons for this is the fact that dependency grammars seem to be better suited for free word order languages (such as Polish) than phrase structure grammars. In recent years various parsers were developed for many languages, however very few were created for Polish. Most likely the most efficient one today is DGP (Dependency Graph Parser). This method however is non-deterministic which is a big disadvantage in some applications like question-answering systems. In our research we aim to overcome this problem by adding weights to grammar rules and thus making possible to rank the output trees and choose the most appropriate one.

2 Grammar

DGP was developed together with complex dependency grammar (for Polish) by Obrebski (2002).

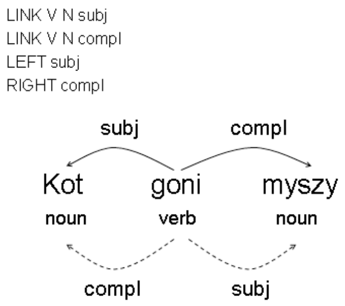

Figure 1: Rules LEFT and RIGHT preventing creation of unnecessary relations in a polish equivalent of the sentence ”A cat chases mice”

As the basic knowledge about this formalism is crucial, we will briefly explain it here. There are four types of rules in Obrebski's formalism.

2.1 LINK

LINK is the rule that allows connecting two words meeting certain lexical criteria with the relation R, like below.
LINK n_1 n_2 R
For example the rule
LINK V N $subj$
allows connecting verb and noun with the relation subject. LINK is the only rule that allows for creating relations. The others are restrictions that actually prevent from creating too many excessive relations.

2.2 LEFT (RIGHT)

LEFT R (and RIGHT R) allows for creating relation R only if the child word precedes (follows) its head in the parsed sentence.

2.3 SGL (Single)

SGL R prevents the situation in which there will be more than one relation R originating from the same head. For example
LINK V N $subj$

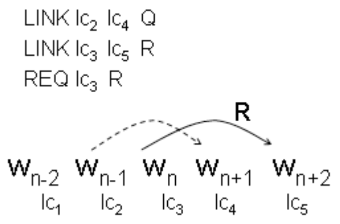

Figure 2: REQ would not allow for creating an egdge over w_n

SGL $subj$
is equivalent to statement that a given verb cannot have more than one subject.

2.4 REQ (Required)

The last and the most complex rule is
REQ w R
It has to be mentioned here that DGP is a projective parser. Due to this restriction if we create relation R between w_{n-1} and w_{n+1} in the example string
w_{n-2} w_{n-1} w_n w_{n+1} w_{n+2}
it would be impossible to link w_n with w_{n+2}. This may be sometimes unacceptable as it may lead to situation in which some nodes (words) will be left unconnected. REQ prevents creating any relation over the word meeting the certain lexical criteria W until W will be linked using relation R. For example REQ V $subj$ is equivalent to the statement that a verb must be linked with other word using relation $subj$.

3 Dependency Graph Parser (DGP)

Before proceeding we need to get familiar with basic concepts of DGP. DGP simply implements the Covington's algorithm (Covington, 2001) for parsing. It is of course extended with a few more steps checking abovementioned restrictions. Details are provided in (Obrebski, 2003) and (Obrebski, 2005)

The most notable feature of the parser is its memory efficiency. It stores all possible outputs as one dependency graph and the final trees are being extracted if needed by a separate tool. This is an important feature because as it was mentioned before DGP is a nondeterministic parser which means it can generate more than one parsing tree for each sentence. For long sentences more than one thousand output trees can be produced. DGP

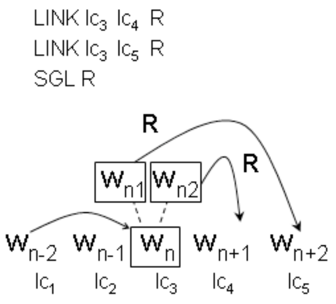

Figure 3: SGL rule forces copying nodes

was originally developed as a tool for extracting sentences meeting certain criteria (e.g. sentences containing noun phrases) from large corpora. Unlike efficiency, non-determinism was not an obstacle although it limits the possible applications. We concentrated on making DGP deterministic, however, as it would be explained later, implementing some of our ideas may seriously affect the parsing speed.

The way of creating a parsing graph and its structure were described in (Obrebski, 2005) so we will not go into details however we have to get familiar with the concept of copying nodes. Using SGL or REQ rule to create a relation affects the properties of head node. For example if we apply SGL rule to a relation R between nodes w_1 and w_2 then (assuming w_1 is the head) we cannot create the same relation between w_1 and w_3. This is a desired behavior, however note that maybe it would be possible to connect w_1 with w_3 instead w_2. This would fulfill SGL rule and could lead to receiving one more correct tree. To overcome this problem the idea of copying nodes were introduced in DGP. We first create a copy of w_1 and make it a head of w_2. Then we create another copy to which w_3 will be attached. There is no limit for the number of copies and the copies can be multi-level (we can copy a copied node).

4 Extending grammar

Our aim is to extend the grammar rules with weights. As a result this will let us to rank the output trees and choose the best one(s). We wanted to leave both grammar and the main parsing untouched, proposed changes do not require any serious modifications. All weigths will be from the range $< 0, 1 > \in \mathbb{R}$.

4.1 Extending LINK

The extended rule will look like the following
LINK n_1 n_2 R w
In a parsing graph this will assign the weight w to the edge representing the relation R.

4.2 Extending LEFT (RIGHT)

The rule will look like the following
LEFT R w (RIGHT R w)
However when it comes to mapping weights to parsing graph we will have three possibilities. Let's say we want to connect the nodes (words) w_1 and w_2 (w_1 precedes w_2) with the relation R where w_2 will be the head and w_1 dependant.

If the rule LEFT for relation R exists we connect nodes and multiply the LEFT's weight with the weight of appropriate LINK rule that has been used to create the relation R. We assign the result to the newly created edge.

If there is no LEFT rule but the RIGHT rule for the relation R exists, we proceed similar as above, but instead of using RIGHT's weight in multiplication, we take $1-w$.

If there is neither LINK nor RIGHT rule we also proceed similarly replacing missing weight with 0.5. Of course for RIGHT rule we perform the similar operations respectively.

In both LINK and LEFT (RIGHT) cases the weights depict how often the given relation or phenomenon occurs in natural language. We hope that extending abovementioned rules in a way we did and mapping to dependency graph is obvious to the reader. In the last case we are forced to add an artifical, neutral weight because without it the edge will get much higher weight than those where LEFT (RIGHT) rules where applied (as there would be no multiplication).

4.3 Extending SGL and REQ

We extend both SGL and REQ rules in similar way however the interpretation of weights and the way of mapping to parsing graph is totally different. Those weights will actually be punishments for violating the given rule and will be assigned to the nodes instead of edges. Assume we are applying an SGL rule to create relation R from w_1 to w_2. As it was described in above paragraph the node w_1 will be copied. If the rule is not violated it will get a neutral weight equal to 1. However if we were about to violate the rule and make w_1 the head of two the same relations R we would create another copy and assign it the weight (punishment) associated with the rule. We will proceed respectively for a REQ rule. To compute this kind of weight we will have to check in how many cases a given phenomenon occurs in real language (e.g. in how many cases a sentence has only one subject in comparison to having two or more). The final weight is $1-computedvalue$. We must mention here, that original Obrebski's algorithm does not allow for violating neither SGL nor REQ rule. By allowing such an operation we want to make possible to construct trees that could not be constructed otherwise (due to the strictness of the grammar). This is supposed to improve the parser's accuracy. However this can greatly increase the processing time as the number of output trees may increase significantly. In other words this extension should be implemented only if we value the parser's accuracy over processing time.

4.4 Selecting the best tree

The final rank of the tree is calculated in the following way: first we multiply the weight of each edge with the weight of its head. Then we multiply the weights from all edges. If the result is x we compute $\sqrt[k]{x}$ where k is the number of edges in the tree. The root is used to make the final rank independent of the number of edges and also the rank more human-readable (this step however can be omitted in most cases). As this may be a little bit unclear for the reader we will show the pseudocode for better understanding:

```
for each e in set of tree's edges
    e.weight = e.source_node.weight

for each e in set of tree's edges
    tree.rank = tree.rank * e.weight
```

The best tree will be the one with the heighest rank. It should be mentioned that as there might be more than one interpretation of a given sentence we don't have to limit ourselves to choosing only one tree.

5 Conclusion

The weights may be obtained by analyzing the tree bank. The way of doing may slightly differ depending on the language. Zelman (2004) is pointing that taking some structures present in Prague Dependency Treebank into account may affect the statistical model in a bad way. That's way we have to use caution while creating appropriate machine learning algorithm.

Unfortunately there is no dependency treebank for Polish that we could use. We experimented with a very small treebank (92 sentences, which is obviously not enough to assign weights to almost 30 000 grammar rules). The sentences we used to check parser's accuracy were manipulated to compensate for small amount of training data. Although the results were promising, they are not enough to draw any conlusions until more source data will be obtained and larger tests will be conducted. Currently we focus our research on building linguistic resources that would allow us to run more test. If the extensions we proposed will prove usefull, we will continue with making the parser working with Japanese and other languages.

References

1 Tomasz Obrebski. 2002. ”Automatyczna analiza skladniowa jezyka polskiego z wykorzystaniem gramatyki zaleznosciowej” [in Polish], PhD Thesis, Poznan University of Technology, Poland.

2 Tomasz Obrebski. 2003. ”Dependency parsing using dependency graph”, Proceedings of the 8th International Workshop on Parsing Technologies (IWPT).

3 Tomasz Obrebski. 2005. ”An all-paths parsing algorithm for constraint based dependency grammars of CF power”, Springer Berlin, pp. 139-146.

4 Michael Covington. 2001. ”A fundamental algorithm for dependency parsing”, Proceedings of the 39th annual ACM southeast conference.

5 Joakim Nivre. 2005. ”Dependency Grammar and Dependency Parsing”, MSI Report.

6 Daniel Zelman. 2004. ”Parsing with a statistical dependency model”, PhD Thesis, Karlov University, Prague