Investigating an implementation of Joakim Nivre's algorithm for projective dependency parsing of Swedish text.

Jörgen Hartman
Lund University Computer Science
jorgen.hartman.398@student.lu.se

Abstract

This paper presents some statistics on an implementation of Joakim Nivre's algorithm. The implementation in Prolog have a coverage of 100% because of the backtracking mechanism in Prolog. The rank of the correct graph within all produced graphs are not very good with this implementation. This paper also shows that the rank can be improved at the cost of the coverage.

1 Introduction

How hard is it to find the correct dependency graph from Swedish sentences? I will investigate an implementation of the algorithm described by Nivre [1]. The implementation itself was made by Pierre Nugues at Lunds Tekniska Högskola. I will try the implementation on an annotated Swedish Treebank called Talbanken. Talbanken is tagged using a probabilistic part-of-speech-tagger trained on the Stockholm Umeå Corpus (SUC). Talbanken consists of about 5000 sentences.

2 The algorithm.

Nivre's algorithm uses a basic shift reduce algorithm extended with some more parse actions:

- Left arc - Adds an arc from head to left dependent if there is a dependency rule that allows it.
- Right arc - Adds an arc from head to right dependent if there is a dependency rule that allows it.
- Reduce - Pops the node on top of the stack if it has a head.
- Shift - Pushes next word of the input onto the stack.

The algorithm makes sure that the graph will be acyclic, connected, and projective. The algorithm will always find a graph for any sentence provided there are lexical rules for all dependencies in the sentence. The algorithm and its different actions are further described in Nivre [1].

2.1 The implementation of the algorithm.

The implementation is done in Prolog programming language. Prolog uses a backtracking mechanism that allows the algorithm to produce a new alternative graph until the correct one is found or all alternatives are found. For example; if the dependency between two words can be determined by two different rules, Prolog will try the first rule, and if it does not produce a correct graph it will go back and try the other rule.

3 Investigation of the implementation.

The idea was to see if the algorithm always could find the correct graph. Prolog uses a backtracking mechanism that makes it possible to find several different graphs that can be generated from a single sentence. It would be interesting to see the rank of the correct graph within all the generated graphs.

I started with a Treebank called Talbanken. It's an annotated Treebank with Swedish sentences. It contains information like index and class of the words, index for the sentence, and dependencies between words. I used that information to extract the dependency graph from Talbanken and compare that graph with the graph produced by nivre's algorithm. I wrote a perl script that extracts the sentences from Talbanken into a file that can easily be read by prolog, and I made a script that extracted the graphs as well. The algorithm needs a set of dependency rules that covers the dependencies in the sentences. If a rule was missing you would not be able to find the correct graph. Since all information is available in Talbanken, I wrote a perl script that extracts all the dependency rules as well.

The test was then done in prolog. The testing program reads the first sentence and then uses the implementation of Nivre's algorithm to produce all possible dependency graphs. It then compares this list of graphs with the dependency graph taken from Talbanken and writes the rank of the correct graph to a result file. If no correct graph was found, the rank would be set to zero.

I immediately ran into problems with stack overflows. Making prolog do a list containing all possible graphs for a sentence required a lot of memory, especially if the sentence was long. I made changes in all three perl scripts so you could set a maximum sentence length. It would now be easy to only extract sentences up to a specific length along with the corresponding graphs and dependency rules.

Tests with sentences of maximum length five showed that the coverage (number of correct graphs found) was 100%, and 74% of the correct graphs was ranked number one.

I found that some of the sentences in Talbanken were only one word long. The reason is that there are for example titles inside the annotated text in Talbanken. One-word sentences will of course always produce the correct dependency graph since it is equal to an empty list. I decided to implement an option to skip over one-word sentences when extracting information. With the same sentences but excluding those consisting of only one word, I still got 100% coverage, but only 47% of the graphs were ranked number one. So I will from now on exclude one-word sentences.

I tried with maximum sentence length ten, but it was not able to complete the algorithm due to memory shortage. The number of graphs generated from a sentence with ten words would be huge. I now understood that I would not be able to get ALL the graphs. I needed a way to limit the calculations. I started looking in the dependency rules direction, since every new rule would generate a large amount of graph combinations.

I experimented with the 1098 first sentences from Talbanken and got these results:

Maximum sentence length eight generated 252 sentences and graphs, and 110 rules.

100% coverage and 18% of the graphs ranked number one (average rank 387).

The same 252 sentences and rules extracted for sentences of maximum 7 words, I got 93 rules, 93% coverage and 20% of the graphs ranked number one (average rank 233).

The same 252 sentences and rules extracted for sentences of maximum 6 words, I got 84 rules, 88% coverage and 22% of the graphs ranked number one (average rank 202).

The result shows that if you have fewer rules the rank will get better, but the coverage will decrease.

Since talbanken is tagged by a probabilistic POS-tagger, some of the dependencies might be incorrect. Statistically, faulty dependency rules would be more common for longer sentences than for shorter.

The dependency rules can be either left or right oriented and when checking for a matching rule inside the algorithm you will try both rules. This generates a lot of backtracking in prolog, and requires a lot of memory. Extracting the dependency rules from shorter sentences will provide the algorithm with fewer rules to match. This will of course lower the coverage of the algorithm since some rules might be missing completely, but it will increase the speed, lower the memory usage of the implementation, and increase the rank of the graphs that are correct.

When you build a sentence you first make the core of the sentence, for example 'bilen röd'. Then you apply different rules to make the sentence more readable, for example linking the noun and adjective like 'bilen är röd'. You also add determiners like 'Den bilen är röd'. One of the last things you do before the sentence is complete is topicalization to restructure the phrases like 'bilen den är röd'. That would generate a dependency rule; 'From noun to determiner where determiner is to the right of the noun'. This looks like a strange rule, and it's not common at all. It would cause the implementation in prolog to make a lot of extra graphs for all sentences containing determiners. Topicalizations are more common in longer sentences than in shorter ones. This shows why the rank can be improved at the cost of the coverage by extracting the dependency rules from shorter sentences only.

Another way of limiting the number of dependency rules would be to split up the data. Using smaller chunks of text from Talbanken, I suspected I could increase the rank. I also suspected that the coverage would go down since each rule would percentage wise be a larger part of the rules needed for 100% coverage.

With 500 sentences from Talbanken and maximum sentence length eight, the perl script generated 102 sentences and 83 dependency rules. The test result I got now was:

100% coverage, 22% was rank one, average rank was 344. The same sentences but with rules extracted for sentences of maximum 7 words, I got 61 rules, 84% coverage, 31% was rank one, average rank was 91. The same sentences but with rules extracted for sentences of maximum 6 words, I got 54 rules, 79% coverage, 35% was rank one, average rank was 81.

With 200 sentences from Talbanken and maximum sentence length eight, the perl script generated 38 sentences and 42 dependency rules. The test result I got now was: 100% coverage, 39% was rank one, average rank was 88. The same sentences but with rules extracted for sentences of maximum 7 words, I got 32 rules, 79% coverage, 53% was rank one, average rank was 38. The same sentences but with rules extracted for sentences of maximum 6 words, I got 25 rules, 66% coverage, 60% was rank one, average rank was 23.

Ultimately, you could extract the rules dynamically from Talbanken, giving a set of rules for each sentence. The algorithm would then use only the rules attached to a specific sentence. That would give 100% coverage and a very good rank. Of course this would not apply to real life cases, because you then need to have all rules available at all times, but it would give a good idea on how robust the algorithm is.

There are more ideas on how to improve the rank of the dependency graphs. One of them would be to implement a probability check for the dependency rules within the algorithm. If the algorithm had two rules to choose from, it would choose the one which are most often used.

Talbanken is tagged with features in addition to the part-of-speech tag. The feature is more granular than the part-of-speech tag and describe for example tense (present, preterite, supinum, and infinite) and degree (positive, comparative, and superlative). Keeping these features will make each dependency rule apply to less words. It would therefore be interesting to investigate the rank when keeping the features within the dependency rules.

4 Conclusion

The implementation of Nivre's algorithm is very robust and will always produce a graph provided that there are dependency rules covering the word classes in the sentences. The correct graph will always be produced because of the backtracking mechanism in prolog. Too many rules or faulty rules will make the implementation produce lower (worse) ranked graphs.

5 Acknowledgements

Pierre Nugues, Lunds Tekniska Högskola: Project leader, algorithm implementation.
Klas Sigbo, Lunds Tekniska Högskola: Answering Prolog questions.
Richard Andersson, Student at Lund University Cognitive Science: Linguistic discussions.

References

[1] Joakim Nivre: An efficient algorithm for projective dependency parsing. School of Mathematics and Systems Engineering, Växjö University.
[2] Pierre Nugues: An Introduction to Language Processing with Perl and Prolog, August 2004. Unfinished book used as course material, Lunds Tekniska Högskola.
[3] Joakim Nivre and Mario Scholz: Deterministic dependency parsing of English text. School of Mathematics and Systems Engineering, Växjö University.

Appendix:
Users manual.

Files needed:
conv_xml2sent.perl
conv_xml2graph.perl
conv_xml2drules.perl
calcResult.perl
nivre_2.pl
readfiles.pl
TalbankenMalt.xml

TalbankenMalt.xml can be downloaded from http://w3.msi.vxu.se/~nivre/research/talbanken.html

Make sure you have an untouched version of the file TalbankenMalt.xml or at least a part of the original file that contains the sentences you wish to work with.

If you want to limit the tests to only work with sentences up to a specific length, you need to edit the value of the variable sentence_length_limit. This is done in all three perl scripts (conv_xml2...). Please note that the script that extracts the sentences and the graphs need to have the same value.

if you want to exclude sentences that are only one word long (they will always give correct graph) from the tests, you need to change the variable named $removeSingleWordSentences. The value true will make the perl script skip all one-word sentences. This variable needs to be changed in both conv_xml2sent.perl and conv_xml2graph.perl. This variable is not found in the dependency rules extraction script because there are no dependencies in a one-word sentence.

NOTE: Make sure you work with the SAME input file when running the three different perl scripts.

Convert the xml to sentences that nivres algorithm can read in prolog:
perl conv_xml2sent.perl TalbankenMalt.xml
This will produce a <Sentences> file.

The dependency rules needed by Nivre's algorithm are extracted by a perl script:
perl conv_xml2drules.perl TalbankenMalt.xml
This will produce a <Dependency rules> file.

The correct graphs as given by TalbankenMalt.xml is extracted by a perl script:
perl conv_xml2graph.perl TalbankenMalt.xml
This will produce a <Graphs> file.

start prolog with:
pl -G20m
This increases the global stack size to 20 Mb (4 Mb default). You might have to increase even more for longer sentences.

consult needed files (algorithm, dependency rules, help predicates):
consult([nivre_2,drules,readfile]).

The command:
readfiles(<Sentences>,<Graphs>,<Result>).
should perform the tests and write the result to the <Result> file.

Run the perl script called calcResult.perl to calculate the coverage, rank, and other statistics for the result:
perl calcResult.perl result
This will give the statistics in the terminal window.