Dependency Parsing with Efficient Feature Extraction

Alexander Volokh and Günter Neumann

DFKI. Stuhlsatzenhausweg 3. 66123 Saarbrücken, Germany
{alexander.volokh,neumann}@dfki.de

Abstract. The fastest parsers currently can parse an average sentence in up to 2.5ms, a considerable improvement, since most of the older accuracy-oriented parsers parse only few sentences per second. It is generally accepted that the complexity of a parsing algorithm is decisive for the performance of a parser. However, we show that the most time consuming part of processing is feature extraction and therefore an algorithm which allows efficient feature extraction can outperform a less complex algorithm which does not. Our system based on quadratic Covington's parsing strategy with efficient feature extraction is able to parse an average English sentence in only 0.8ms without any parallelisation.

1 Introduction

Dependency parsers have recently become very popular and beneficial for many natural language processing (NLP) tasks, because of their ability to reliably capture useful information within a sentence. However, the quality of the result is not the only requirement. Many applications, especially those which work with huge amounts of data or applications where processing has to be done online within milliseconds, require parsing to be particularly fast in order to be eligible for use.

In the 2000s, the very popular CoNLL-X [2] shared tasks in dependency parsing brought a lot of progress to the field. However, the evaluation highly preferred accuracy and efficiency was neglected. Some of the most widely used parsers from that time, e.g. MaltParser_SVM [7], Stanford Parser [5] or MST Parser [6], have great accuracies but can only parse around 3 sentences per second. The more recent works: e.g. MaltParser_Liblinear, Ensemble [8], mate-tools [1] or ClearParser [3] have better efficiencies. MaltParser trained with linear classifiers can parse up to 1 sentence in 2.5ms, Ensemble in 10 ms, Bohnet's parser - 77ms and ClearParser - 2.29 ms.

All dependency parsers can be split into two approaches: transition-based and graph-based. Whereas the accuracies of these systems are quite similar, the numbers above clearly demonstrate that transition-based systems are more efficient (mate-tools is the only graph-based). We will restrict ourselves to the transition-based.

Transition-based systems start at some initial configuration and perform a sequence of transitions to some final configuration, such that the desired dependency graph is derived in the process. There are plenty of parsing strategies with different sets of possible transitions, which are capable of solving this task. It is usually considered that the number of configurations is the most important property for the efficiency of the algorithm. E.g. Nivre's arc-standard (AS) and arc-eager (AE) algorithms [7] require O(n) transitions, whereas Covington's parsing strategy [4] requires $O(n^2)$ in the

B. Glimm and A. Krüger (Eds.): KI 2012, LNCS 7526, pp. 253–256, 2012.

worst-case. Therefore there are a lot different variations of Nivre's algorithm (including all of the above parsers), whereas Covington's strategy is much less popular.

However, the number of configurations is not the only important property of an algorithm. It is rather important how long it takes to perform a transition from one configuration to another. We have used profiling technology in order to determine which parts of code amount for which percentage of execution time and found out that most of the running time is spent for extracting features (also reported by [1] for their graph-based system), which are used to predict the most probable transition. Even though less complex algorithms require feature extraction less often, we will show that feature extraction costs vary considerably across different strategies.

In this paper we show that Covington's parsing strategy is particularly suitable for efficient feature extraction. Despite the fact that its theoretical complexity is quadratic in the length of the sentence, in practice the worst-case never occurs and thus it can easily outperform linear strategies without efficient feature extraction. In our experiments we could achieve parsing speed of 0.8 ms per sentence.

2 Complexity in Theory

Given a sentence s, consisting out of words w_1 to w_n the objective of a transition-based parsing strategy is to find all dependencies (w_i, l, w_j), i.e. pairs of words w_i and w_j, which stand in a syntactic relation l. The most naive strategy to do that is to examine every possible pair of words and link them if necessary. Covington's strategy proposes an intelligent refinement to this. First, when searching for potential links for a word j it works backward. This way heads and dependents are found earlier, because they are more likely to be near than far away. Second, many pairs are discarded because they violate permissibility, i.e. well-formedness constraints of a dependency tree. Examples of such constrains are that words can have only one head, the whole structure can have only one root, there can be no cycle and if necessary that there are no crossing branches (projectivity). The worst-case complexity remains $O(n^2)$.

Nivre's AE or AS algorithms propose a further restriction of the search space. They use two stacks and allow only top elements from these stacks to be linked, which guarantees that no invalid dependency structure comes into being. Additionally, already processed words are removed from the stacks, such that they are no longer eligible for other words to come. This way the algorithms have O(n) complexity.

3 Complexity in Practice

Let us consider the sentence $Economic_1$ $news_2$ had_3 $little_4$ $effect_5$ on_6 $financial_7$ $markets_{8.9}$. We have used MaltParser, which has all algorithms implemented, with option "-m testdata" in order to analyse how many transitions are necessary to parse the data. With Nivre's arc-eager strategy it took 16 transitions to parse the example sentence, for Nivre's arc-standard 17 transitions were necessary and for Covington's algorithm 33 word pairs are examined, however, 17 of them are not permissible and thus there are only 16 real configurations. By real configurations we mean those for which

feature vectors actually have to be constructed and the correct transition has to be predicted. For non-permissible states it is not necessary and they therefore hardly influence the overall performance. Thus the theoretically more complex Covington's strategy in practice does not re-quire more real configurations than Nivre's linear algorithms. We have performed similar experiments for the whole CoNLL English development data and found out that for these 1337 sentences 63916 real configurations are required with Covington's algorithm, 64137 with AE and 65148 with AS.

4 Feature Extraction

In order to predict what transition should be performed in which parser state, the parser state is trans-formed into a feature vector and according to the previously learned model the best transition is selected. The algorithms presented in this paper require a similar number of feature templates in order to achieve similarly competitive performance. In MaltParser arc-standard default algorithm runs with 21 different templates, arc-eager with 22 and Covington's algorithm also uses 22 feature templates.

In his PhD [7] Nivre differentiates between static and dynamic feature templates. Static templates always return the same value for the same input, e.g POS tags of the words never change. Dynamic feature templates might change their output in course of processing, e.g. the dependency label of a word is null in the beginning and changes to some non-null value as soon as the word gets a head.

The decisive difference between the algorithms is that many other features, which actually are also static can only be reused in Covington's and not in Nivre's algorithms, where the reusability of features is limited, because one never knows what the stacks will look like and it would be too memory intensive to keep all possibilities in memory until it is clear which one of them is correct. The reusability of static features considerably improves the performance of an algorithm, since it is no longer necessary to look up the value of a feature and then its index in a global alphabet (mapping of strings to unique integers constructed during the training of the model; might contain tens of thousands of different values and thus is not so fast) so often. Instead, we consult the global mapping only once for all features which are used many times and store those in a different local (i.e. valid only within the current sentence) data structure from where they can be retrieved much faster.

Additionally, in order to compensate for the lack of a kernel, which creates conjoined features implicitly, one has to add artificial feature combinations manually. In MaltParser's feature models for Liblinear around 40% are feature combinations, which are concatenations of basic features.

String is an immutable basic type in Java, each time you append something a new String is created, the old value is stored the new value is added, and the old String is thrown away. The longer the strings the longer the concatenations take, but even for typical feature lengths of ~10 characters it takes around 0.25 µs. For 780,000 feature combinations (the amount required for 65000 configurations) it would mean around 0.2 seconds, i.e. around 20% of the whole time if one aims to parse a sentence in less than 1ms. Therefore it is even more important that feature combinations are reused whenever possible, since they contain costly string operations.

Even though String operations are expensive in Java, there are no alternatives. Tricks like translating features to integers and substituting concatenation by multiplication do not work better, since they require a mapping from the String values to ints and the look up in such large collections is even more expensive than concatenation.

5 Results and Conclusion

We have implemented a system which is based on Covington's parsing strategy and reuses static features whenever possible. We could achieve a parsing speed of 0.8 ms/sentence for an average English sentence (24.41 words). Despite the worse theoretical complexity, we have shown that in practice other properties are more important. In particular, that most of the execution time is spent on feature extraction and thus the suitability of an algorithm for efficient feature extraction is decisive.

For space reasons we could not discuss the accuracies of different algorithms and models. However, running MaltParser with default models has shown that the accuracy of Covington's algorithm for English is better than the accuracy with Nivre's algorithms. Both the default MaltParser's model and a model where the feature conjunctions are replaced by static ones have very similar accuracies.

The tests were performed on a 2.4 GHz CPU with only one core used.

Acknowledgements. The work presented here was partially supported by a research grant from the German Federal Ministry of Education and Research (BMBF) to the DFKI project Deependance (FKZ. 01IW11003).

References

1. Bohnet, B.: Top Accuracy and Fast Dependency Parsing is not a Contradiction. In: COLING 2010, Beijing, China (2010)
2. Buchholz, S., Marsi, E.: CoNLL-X shared task on multilingual dependency parsing. In: Proceedings of CONLL-X, New York, pp. 149–164 (2006)
3. Choi, J.D., Palmer, M.: Getting the Most out of Transition-based Dependency Parsing. In: ACL: HLT 2011, Portland, Oregon, USA, pp. 687–692 (2011)
4. Covington, M.A.: A Fundamental Algorithm for Dependency Parsing. In: Proceedings of the 39th Annual ACM Southeast Conference (2000)
5. Klein, D., Manning, C.D.: Accurate Unlexicalized Parsing. In: ACL 2003, pp. 423–430 (2003)
6. McDonald, R., Pereira, F., Ribarov, K., Hajič, J.: Non-Projective Dependency Parsing using Spanning Tree Algorithms. In: HLT 2005 (2005)
7. Nivre, J.: Inductive Dependency Parsing (Text, Speech and Language Technology). Springer-Verlag New York, Inc., Secaucus (2006)
8. Surdeanu, M., Manning, C.D.: Ensemble Models for Dependency Parsing: Cheap and Good? In: NAACL 2010 (2010)