Combining Czech Dependency Parsers*

Tomáš Holan and Zdeněk Žabokrtský

Faculty of Mathematics and Physics, Charles University
Malostranské nám. 25, CZ-11800 Prague, Czech Republic
{tomas.holan,zdenek.zabokrtsky}@mff.cuni.cz

Abstract. In this paper we describe in detail two dependency parsing techniques developed and evaluated using the Prague Dependency Treebank 2.0. Then we propose two approaches for combining various existing parsers in order to obtain better accuracy. The highest parsing accuracy reported in this paper is 85.84 %, which represents 1.86 % improvement compared to the best single state-of-the-art parser. To our knowledge, no better result achieved on the same data has been published yet.

1 Introduction

Within the domain of NLP, dependency parsing is nowadays a well-established discipline. One of the most popular benchmarks for evaluating parser quality is the set of analytical (surface-syntactic) trees provided in the Prague Dependency Treebank (PDT). In the present paper we use the beta (pre-release) version of PDT 2.0,[1] which contains 87,980 Czech sentences (1,504,847 words and punctuation marks in 5,338 Czech documents) manually annotated at least to the analytical layer (a-layer for short).

In order to make the results reported in this paper comparable to other works, we use the PDT 2.0 division of the a-layer data into training set, development-test set (d-test), and evaluation-test set (e-test). Since all the parsers (and parser combinations) presented in this paper produce full dependency parses (rooted trees), it is possible to evaluate parser quality simply by measuring its accuracy: the number of correctly attached nodes divided by the number of all nodes (not including the technical roots, as used in the PDT 2.0). More information about evaluation of dependency parsing can be found e.g. in [1].

Following the recommendation from the PDT 2.0 documentation for the developers of dependency parsers, in order to achieve more realistic results we use morphological tags assigned by an automatic tagger (instead of the human annotated tags) as parser input in all our experiments.

The rest of the paper is organized as follows: in Sections 2 and 3, we describe in detail two types of our new parsers. In Section 4, two different approaches to parser combination are discussed and evaluated. Concluding remarks are in Section 5.

* The research reported on in this paper has been carried out under the 1ET101120503, GAČR 207-13/201125, 1ET100300517.

[1] For a detailed information and references see http://ufal.mff.cuni.cz/pdt2.0/

2 Rule-based Dependency Parser

In this section we will describe a rule-based dependency parser created by one of the authors. Although the first version of the parser was implemented already in 2002 and its results have been used in several works (e.g. [2]), no more detailed description of the parser itself has been published yet.

The parser in question is not based on any grammar formalism (however, it has been partially inspired by several well-known formal frameworks, especially by unification grammars and restarting automata). Instead, the grammar is 'hardwired' directly in Perl code. The parser uses tred/btred/ntred[2] tree processing environment developed by Petr Pajas. The design decisions important for the parser are described in the following paragraphs.

One tree per sentence. The parser outputs exactly one dependency tree for any sentence, even if the sentence is ambiguous or incorrect. As illustrated in Figure 1 step 1, the parser starts with a flat tree – a sequence of nodes attached below the auxiliary root, each of them containing the respective word form, lemma, and morphological tag. Then the linguistically relevant oriented edges are gradually added by various techniques. The structure is connected and acyclic at any parsing phase.

No backtracking. We prefer greedy parsing (allowing subsequent corrections, however) to backtracking. If the parser makes a bad decision (e.g. due to insufficient local information) and it is detected only much later, then the parser can 'rehang' the already attached node (rehanging becomes necessary especially in the case of coordinations, see steps 3 and 6 in Figure 1). Thus there is no danger of exponential expansion which often burdens symbolic parsers.

Bottom-up parsing (reduction rules). When applying reduction rules, we use the idea of a 'sliding window' (a short array), which moves along the sequence of 'parentless' nodes (the artificial root's children) from right to left.[3] On each position, we try to apply simple hand-written grammar rules (each implemented as an independent Perl subroutine) on the window elements. For instance, the rule for reducing prepositional groups works as follows: if the first element in the window is an unsaturated preposition and the second one is a noun or a pronoun agreeing in morphological case, then the parser 'hangs' the second node below the first node, as shown in the code fragment below (compare steps 9 and 10 in Figure 1):

```
sub rule_prep_noun($) {
  my $win = shift;
  if (preposition($win->[0])
    and nominal($win->[1])
    and not $win->[0]->{p_saturated}){
      $win->[0]->{p_saturated}=1;
      return hang($win->[1],$win->[0]);
  } else {  return 0 }
}
```

```
sub rule_adj_noun($) {
  my $win = shift;
  if (adjectival($win->[0]) and noun($win->[1])
       and ($win->[0]->{p_ordinal} or
         (agr_case($win->[0],$win->[1]) and
          agr_number($win->[0],$win->[1]) and
          agr_gender($win->[0],$win->[1])))) {
    return hang($win->[0],$win->[1]);
    } else {  return 0 }
}
```

[2] http://ufal.mff.cuni.cz/~pajas/tred/index.html

[3] Our observations show that the direction choice is important, at least for Czech.

The rules are tried out according to their pre-specified ordering; only the first applicable rule is always chosen. Then the sliding window is shifted several positions to the right (outside the area influenced by the last reduction, or to the right-most position), and slides again on the shortened sequence (the node attached by the last applied rule is not the root's child any more). Presently, we have around 40 reduction rules and – measured by the number of edges – they constitute the most productive component of the parser.

Interface to the tagset. Morphological information stored in the morphological tags is obviously extremely important for syntactic analysis. However, the reduction rules never access the morphological tags directly, but exclusively via a predefined set of 'interface' routines, as it is apparent also in the above rule samples. This routines are not always straightforward, e.g. the subroutine `adjectival` recognizes not only adjectives, but also possessive pronouns, some of the negative, relative and interrogative pronouns, some numerals etc.

Auxiliary attributes. Besides the attributes already included in the node (word form, lemma, tag, as mentioned above), the parser introduces many new auxiliary node attributes. For instance, the attribute `p_saturated` used above specifies whether the given preposition or subordinating conjunction is already 'saturated' (with a noun or a clause, respectively), or special attributes for coordination. In these attributes, a coordination conjunction which coordinates e.g. two nouns pretends itself to be a noun too (we call it the effective part of speech), so that e.g. a shared attribute modifier can be attached directly below this conjunction.

External lexical lists. Some reduction rules are lexically specific. For this purpose, various simple lexicons (containing e.g. certain types of named entities or basic information about surface valency) have been automatically extracted either from the Czech National Corpus or from the training part of the PDT, and are used by the parser.

Clause segmentation. In any phase of the parsing process, the sequence of parentless nodes is divided into segments separated by punctuation marks or coordination conjunctions; the presence of a finite verb form is tested in every segment, which is extremely important for distinguishing interclausal and intraclausal coordination.[4]

Top-down parsing. The application of the reduction rules can be viewed as bottom-up parsing. However, in some situations it is advantageous to switch to the top-down direction, namely in the cases when we know that a certain sequence of nodes (which we are not able to further reduce by the reduction rules) is of certain syntactic type, e.g. a clause delimited on one side by a subordinating conjunctions, or a complex sentence in a direct speech delimited from both sides by quotes. It is important especially for the application of fallback rules.

Fallback rules. We are not able to describe all language phenomena by the reduction rules, and thus we have to use also heuristic fallback rules in some situations. For instance, if we are to parse something what is probably a single

[4] In our opinion, it is especially coordination (and similar phenomena of non-dependency nature) what makes parsing of natural languages so difficult.

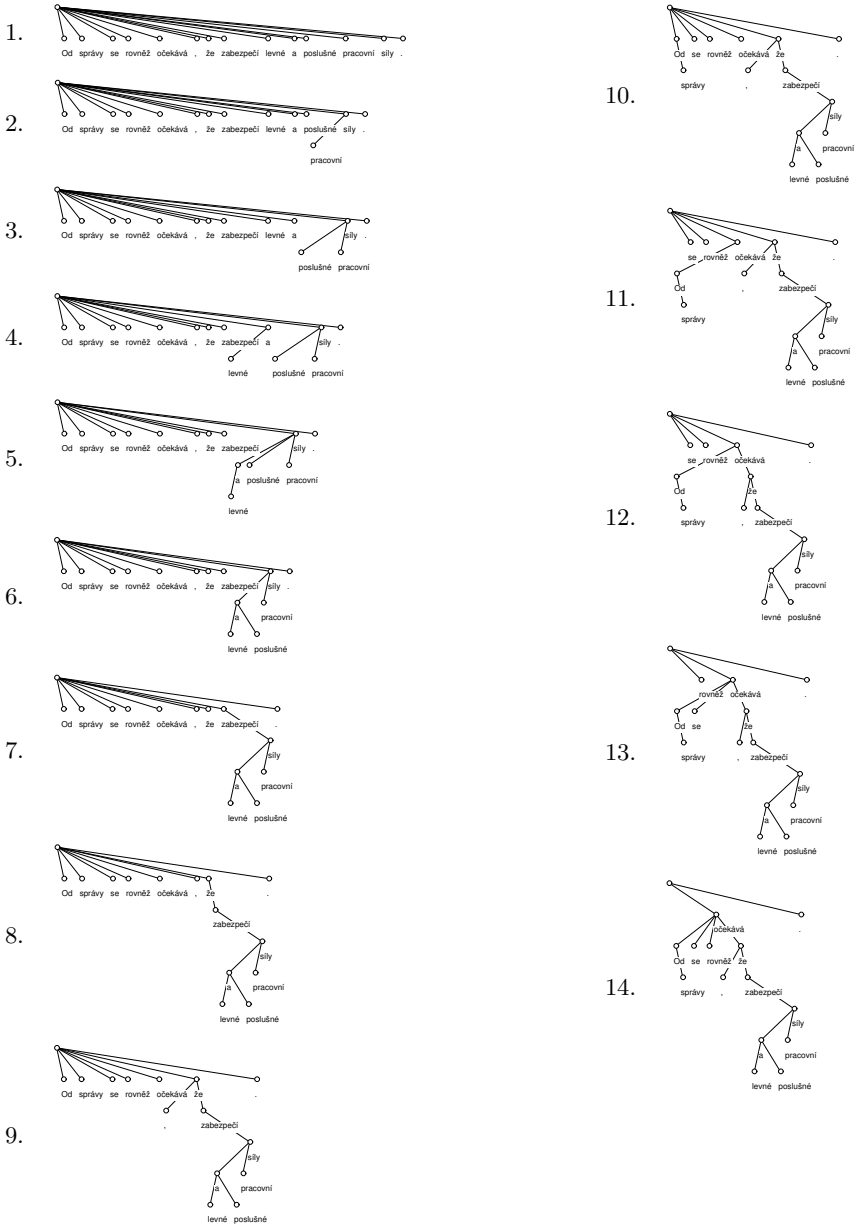

Fig. 1. Step-by-step processing of the sentence *'Od správy se rověž očekává, že zabezpečí levné a poslušné pracovní síly.'* (The administration is also supposed to ensure cheap and obedient manpower.) by the rule-based parser.

clause and no reduction rules are no longer applicable, then the finite verb is selected as the clause head and all the remaining parentless nodes are attached below it (steps 11-14 in Figure 1).

Similar attempts to parsing based on hand-coded rules are often claimed to be hard to develop and maintain because of the intricate interplay of various language phenomena. In our experience and contrary to this expectation, it is possible to reach a reasonable performance (see Table 1), speed and robustness within one or two weeks of development time (less than 2500 lines of Perl code). We have also verified that the idea of our parser can be easily applied on other languages – the preliminarily estimated accuracy of our Slovene, German, and Romanian rule-based dependency parsers is 65-70 % (however, the discussion about porting the parser to other languages goes beyond the scope of this paper).

As for the parsing speed, it can be evaluated as follows: if the parser is executed in the parallelized ntred environment employing 15 Linux servers, it takes around 90 seconds to parse all the PDT 2.0 a-layer development data (9270 sentences), which gives roughly 6.9 sentences per second per server.

3 Pushdown Dependency Parsers

The presented pushdown parser is similar to those described in [3] or [4]. During the training phase, the parser creates a set of premise-action rules, and applies it during the parsing phase. Let us suppose a stack represented as a sequence $n_1..n_j$, where n_1 is the top element; stack elements are ordered triplets $< form, lemma, tag >$. The parser uses four types of actions:

- read a token from the input, and push it into the stack,
- attach the top item n_1 of the stack below the artificial root (i.e., create a new edge between these two), and pop it from the stack,
- attach the top item n_1 below some other (non-top) item n_i, and pop the former from the stack,
- attach a non-top item n_i below the top item n_1, and remove the former from the stack.[5]

The forms of the rule premises are limited to several templates with various degree of specificity. The different templates condition different parts of the stack and of the unread input, and previously performed actions.

In the training phase, the parser determines the sequence of actions which leads to the correct tree for each sentence (in case of ambiguity we use a pre-specified preference ordering of the actions). For each performed action, the counters for the respective premise-action pairs are increased.

During the parsing phase, in each situation the parser chooses the premise-action pair with the highest score; the score is calculated as a product of the value of the counter of the given pair and of the weight of the template used in

[5] Note that the possibility of creating edges from or to the items in the middle of the stack enables the parser to analyze also non-projective constructions.

the premise (see [5] for the discussion about template weights), divided by the exponentially growing penalty for the stack distance between the two nodes to be connected.

In the following section we use four versions of the pushdown parser: L2R – the basic pushdown parser (left to right), R2L – the parser processing the sentences in reverse order, L23 and R23 – the parsers using 3-letter suffices of the word forms instead of the morphological tags.

The parsers work very quickly; it takes about 10 seconds to parse 9270 sentences from PDT 2.0 d-test on PC with one AMD Athlon 2500+. Learning phase takes around 100 seconds.

4 Experiments with Parser Combinations

This section describes our experiments with combining eight parsers. They are referred to using the following abbreviations: McD (McDonnald's maximum spanning tree parser, [6]),[6] COL (Collins's parser adapted for PDT, [7]), ZZ (rule-based dependency parser described in Section 2), AN (Holan's parser ANALOG which has no training phase and in the parsing phase it searches for the local tree configuration most similar to the training data, [5]), L2R, R2L, L23 and R32 (pushdown parsers introduced in Section 3). For the accuracy of the individual parsers see Table 1.

We present two approaches to the combination of the parsers: (1) Simply Weighted Parsers, and (2) Weighted Evaluation Classes.

Simply Weighted Parsers (SWP). The simplest way to combine the parsers is to select each node's parent out of the set of all suggested parents by simple parser voting. But as the accuracy of the individual parsers significantly differ (as well as the correlation in parser pairs), it seems natural to give different parsers different weights, and to select the eventual parent according to the weighted sum of votes. However, this approach based on local decisions does not guarantee cycle-free and connected resulting structure. To guarantee its 'treeness', we decided to build the final structure by the Maximum Spanning Tree algorithm (see [6] for references). Its input is a graph containing the union of all edges suggested by the parsers; each edge is weighted by the sum of weights of the parsers supporting the given edge. We limited the range of weights to small natural numbers; the best weight vector has been found using a a simple hill-climbing heuristic search.

We evaluated this approach using 10-fold cross evaluation applied on the PDT 2.0 a-layer d-test data. In each of the ten iterations, we found the set of weights which gave the best accuracy on 90 % of d-test sentences, and evaluated the accuracy of the resulting parser combination on the unseen 10 %. The average accuracy was 86.22 %, which gives 1.98 percent point improvement compared to McD. It should be noted that all iterations resulted in the same weight vector:

[6] We would like to thank Václav Novák for providing us with the results of McD on PDT 2.0.

Table 1. Percent accuracy of the individual parsers when applied (separately) on the PDT 2.0 d-test and e-test data.

	McD	COL	ZZ	AN	R2L	L2R	R23	L23
d-test	84.24	81.55	76.06	71.45	73.98	71.38	61.06	54.88
e-test	83.98	80.91	75.93	71.08	73.85	71.32	61.65	53.28

(10, 10, 9, 2, 3, 2, 1, 1) for the same parser ordering as in Table 1. Figure 2 shows that the improvement with respect to McD is significant and relatively stable.

When the weights were 'trained' on the whole d-test data and the parser combination was evaluated on the e-test data, the resulting accuracy was 85.84 % (1.86 % improvement compared to McD), which is the best e-test result reported in this paper.[7]

Weighted Equivalence Classes (WEC). The second approach is based on the idea of partitioning the set of parsers into equivalence classes. At any node, the pairwise agreement among the parsers can be understood as an equivalence relation and thus implies partitioning on the set of parsers. Given 8 parsers, there are theoretically 4133 possible partitionings (in fact, there are only 3,719 of them present in the d-test data), and thus it is computationally tractable.

In the training phase, each class in each partitioning obtains a weight which represents the conditional probability that the class corresponds to the correct result, conditioned by the given partitioning. Technically, the weight is estimated as the number of nodes where the given class corresponds to the correct answer divided by the number of nodes where the given partitioning appeared.

In the evaluation phase, at any node the agreement of results of the individual parsers implies the partitioning. Each of the edges suggested by the parsers then corresponds to one equivalence class in this partitioning, and thus the edge obtains the weight of the class. Similarly to the former approach to parser combination, the Maximum Spanning Tree algorithm is applied on the resulting graph in order to obtain a tree structure.

Again, we performed 10-fold cross validation using the d-test data. The resulting average accuracy is 85.41 %, which is 1.17 percentage point improvement compared to McD. If the whole d-test is used for weight extraction and the resulting parser is evaluated on the whole e-test, the accuracy is 85.14 %.

The interesting property of this approach to parser combination is that if we use the same set of data both for the training and evaluation phase, the resulting accuracy is the upper bound for of all similar parser combinations based only on the information about local agreement/disagreement among the parsers. If this experiment is performed on the whole d-test data, the obtained upper bound is 87.15 %.

[7] Of course, in all our experiments we respect the rule that the e-test data should not be touched until the developed parsers (or parser combinations) are 'frozen'.

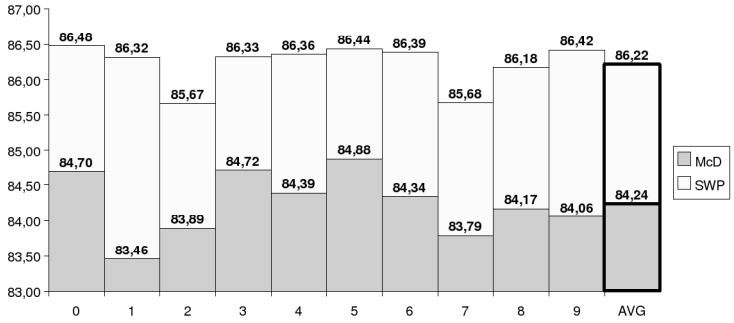

Fig. 2. Accuracy of the SWP parser combination compared to the best single McD parser in 10-fold evaluation on the d-test data.

5 Conclusion

In our opinion, the contribution of this paper is threefold. First, the paper introduces two (types of) Czech dependency parsers, the detailed description of which has not been published yet. Second, we present two different approaches to combining the results of different dependency parsers; when choosing the dependency edges suggested by the individual parsers, we use the Maximum Spanning Tree algorithm to assure that the output structures are still trees. Third, using the PDT 2.0 data, we show that both parser combinations outperform the best existing single parser. The best reported result 85.84 % corresponds to 11.6 % relative error reduction, compared to 83.98 % of the single McDonald's parser.

References

1. Zeman, D.: Parsing with a Statistical Dependency. PhD thesis, Charles University, MFF (2004)
2. Zeman, D., Žabokrtský, Z.: Improving Parsing Accuracy by Combining Diverse Dependency Parsers. In: Proceedings of the 9th International Workshop on Parsing Technologies, Vancouver, B.C., Canada (2005)
3. Holan, T.: Tvorba závislostního syntaktického analyzátoru. In: Sborník semináře MIS 2004. Matfyzpress, Prague, Czech Republic (2004)
4. Nivre, J., Nilsson, J.: Pseudo-Projective Dependency Parsing. In: Proceedings of ACL'05, Ann Arbor, Michigan (2005)
5. Holan, T.: Genetické učení závislostních analyzátorů. In: Sborník semináře ITAT 2005. UPJŠ, Košice (2005)
6. McDonald, R., Pereira, F., Ribarov, K., Hajič, J.: Non-Projective Dependency Parsing using Spanning Tree Algorithms. In: Proceedings of HTL/EMNLP'05, Vancouver, BC, Canada (2005)
7. Hajič, J., Collins, M., Ramshaw, L., Tillmann, C.: A Statistical Parser for Czech. In: Proceedings ACL'99, Maryland, USA (1999)