An Ambiguity Aware Treebank Search Tool

Marcin Woliński and Andrzej Zaborowski

Institute of Computer Science, Polish Academy of Sciences, Warsaw

Abstract. We present a search tool for constituency treebanks with some interesting new features. The tool has been designed for a treebank containing several alternative trees for any given sentence, with one tree marked as the correct one. The tool allows to compare the selected tree with other candidates.
The query language is modelled after TIGER Search, but we extend the use of the negation operator to be able to use a class of universally quantified conditions in queries.
The tool is built on top of an SQL engine, whose indexing facilities provide for efficient searches.

Keywords: treebanks query/search tools, constituency trees, syntactic ambiguity

1 Introduction

The primary goal of the work being reported is to facilitate the access to a treebank we are building. This treebank — Składnica [10] currently comprises constituency trees for about 8200 Polish sentences. The treebank is under active development, so we are interested in a search tool that would not only be useful for "end users" but also which would assist in finding errors and inconsistencies in the currently available trees.

The present treebank was built semi-automatically. Sets of candidate parse trees were generated automatically using a parser and then disambiguated by human annotators. In such a setup it is desirable to be able to confront the whole set of solutions proposed by the parser with the single one selected by annotators. Unfortunately, available treebank search tools [4, 6, 7] assume a single tree for each sentence.

We decided to build our own tool that would be aware of ambiguous syntactic structures present in Składnica. The tool is web based, which provides for an easy access to the treebank from a standard web-browser and without installing specialised programs. We also decided to build our query language on the TIGER Search syntax [4], which looks familiar for the users of Poliqarp [2] commonly used for searching in lower levels of annotation in our corpus.

2 Quick Overview of the TIGER Query Language

A query in the TIGER language [3, Chapter III] describes a set of matching nodes. The construct `[]` matches any node in a tree. It can be limited by specifying values of features combined with logical operators: negation `!`, conjunction `&`, and alternative `|`. For example, the following query matches nodes of category `NP` with feature `fa` equal `val1` or feature `fb` different from `val2`:

```
[cat="NP" & (fa = "val1" | fb != "val2")]
```

The feature `cat` is commonly used to denote category, i.e., the name of the nonterminal unit. In general, however, feature names and values depend on a given treebank.

Alternatives can also be used on the right hand side of an equation, which provides for compact queries like:

```
[cat=("NP"|"PP")]
```

Nodes in a query can be linked using several node relations. The following query

```
[cat="S"] > [cat="NP"]
```

uses the direct dominance relation `>`, so the first node is to be the parent of the second.

Other node relations include indirect dominance `>*` (transitive closure of `>`), labelled dominance `>lbl` (if edges in the treebank are labelled), direct precedence `.`, indirect precedence `.*`, sibling (common parent) relation `$`, and some more variations.

The language uses unification variables denoted with `#name`. A variable can bind the right hand side of an equation, so in the following query we require the two adjacent nodes to be of the same category irrespective which of the two allowed categories it is:

```
[cat=#c:("NP"|"PP")] . [cat=#c]
```

A variable can also bind a single node, so the following query requires the node `#n` to be the parent both to the `NP` and `VP` node:

```
#n:[cat="S"] & #n > [cat="NP"] & #n > [cat="VP"]
```

The third type of variables are “variables for feature constraints”, which bind all features of a particular node, e.g., the query:

```
[#f:(pos="prep")] .* [#f]
```

matches trees where a node with exactly the same features appears twice. Assuming `pos="prep"` selects a terminal being a preposition, both matching nodes must be prepositions with the same orthographic form and the same tag. We don’t find this behaviour very useful, since it can be used only when all features match. Therefore this type of binding was skipped in our implementation. We prefer to state the features that should unify explicitly:

```
[pos="prep" & orth=#o] .* [pos="prep" & orth=#o]
```

The last element we are going to mention are node predicates. For example the following requires node `#n` to have at least 2 and at most 4 children:

```
#n:[cat="S"] & arity(#n,2,4)
```

3 Searching in Parse Forests

Składnica uses shared parse forests [1] as a compact representation of all trees generated by the parser for a given sentence. In the shared parse forest each subtree occurring in multiple complete parse trees is represented only once. The forest consists of nodes, each having the name of the nonterminal unit of the grammar and its morpho-syntactic features as attributes. Each combination of a particular span of text, a nonterminal unit, and its attributes corresponds to a separate node.

Moreover, each node carries a set of lists of its possible immediate constituents. Each list represents children of the given node in one of possible trees.

A subset of nodes is marked as comprising the tree selected by annotators.

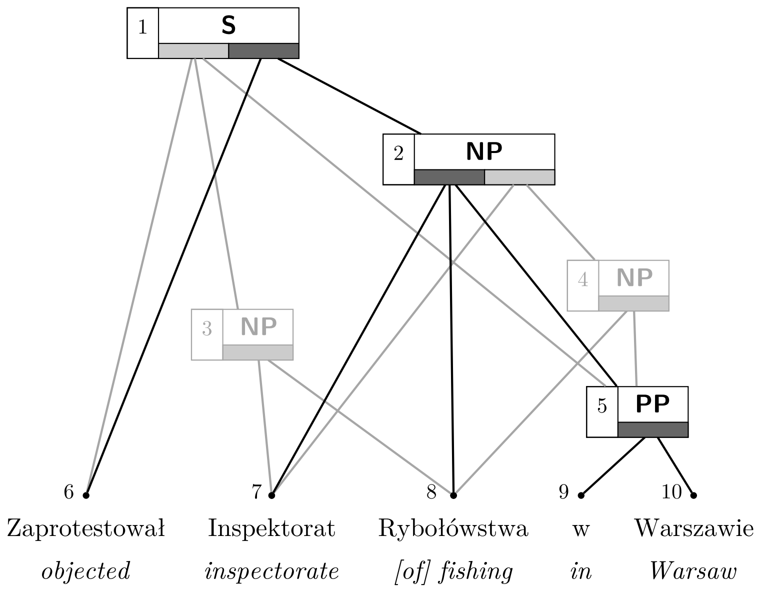


Fig. 1. A simplified shared parse forest for the sentence 'The Inspectorate of Fishing in Warsaw has objected.' The tree selected by the annotators is shown in a darker colour.

For example, Fig. 1 shows the forest for a sentence in which a prepositional phrase 'in Warsaw' can be attached in three places. Node 1, representing the sentence, has two possible lists of children. The first consists of nodes 6 (the verb), 3 (nominal phrase 'Inspectorate of Fishing'), and 5 (prepositional phrase 'in Warsaw'). The second list contains nodes 6 and 2 (nominal phrase 'Inspectorate of Fishing in Warsaw'). This node is part of the selected tree together with its second list of children. Observe that node 2 represents all possible structures of the phrase in question, in particular it represents two attachments of the prepositional phrase: either the Inspectorate is in Warsaw or the Inspectorate deals with fishing in Warsaw. The annotators have selected the first attachment.

The compactness of the representation comes from the fact that each of nodes 5, 7, and 8 is shared by three possible parents and node 6 is shared by two parents. It is worth reminding that a shared parse forest can represent an exponential number of trees in polynomial amount of memory.

When searching in shared forests we need to be able to address particular nodes and check whether they have been selected by the annotators. For that purpose we have equipped the nodes with an extra-grammatical boolean feature `sel`.

Although searching in all possible trees is important to us as treebank developers, we expect users to mainly search in the disambiguated trees. For that reason we reserve the TIGER notation `[ ]` for a node selected by the annotators and denote any node in the forest by `[[ ]]` (which means `[...]` is a shorthand for `[[sel=true & ...]]`).

For the forest in Fig. 1 the query `[cat="NP"]` would return only node 2. On the other hand the query `[[cat="NP"]]` has three answers: nodes 2, 3, and 4.

When searching among all nodes of a forest it may happen that the matching nodes do not all belong to a single tree. For example the answer to the query

```
#n: [[cat="NP"]] > [[cat="PP"]] & #n > [[cat="NP"]]
```

could consist of nodes 2, 5, and 4. This may be exactly what we are interested in, for example when searching for structures which were disambiguated in some way by the annotators while the parser proposed simultaneously a particular different structure. Sometimes, however, we need to request some nodes with `sel=false` to come from a single consistent tree. This can be achieved with the use of the predicate `same_tree(...)`, which is true if all nodes enumerated as its arguments appear in one tree in the forest.

Obviously, these considerations do not apply if all nodes in the query are specified with `[]`, since the nodes with `sel=true` are guaranteed to form a single complete tree.

The following example illustrates a real query we have used to check to what extent the annotators were consistent in their decisions. In Składnica some complements to verbs are labelled as filling the adverbial position while being realised by prepositional-nominal phrases with various prepositions (cf. [8]). The following query matches complements labelled as prepositional by the annotators, while the parser proposed also the adverbial interpretation for the same span of text:

```
[cat="fw" & tfw=/prepnp.*/ & from=#f & to=#t]
 & [[cat="fw" & tfw="advp" & from=#f & to=#t]]
```

In this query we show actual feature values as defined in Składnica. Nonterminal `fw` denotes a complement. The attribute `tfw` denotes structural type of the complement. We use the regular expression `/prepnp.*/` to allow for prepositional phrases with any preposition. The attributes `from=` and `to=` denote positions in the text where the given phrase begins and ends. Their repetition forces the two phrases to span the same text fragment.

4 Searching for the Nonexistent

TIGER Search has a well known limitation with regard to quantification. All node specifications in queries are implicitly existentially quantified. There exists no feature of the query language that would allow to form questions of the form "all nodes such that…" or "there exists no node such that…".

In particular, the authors claim in the manual [3] that the use of the universal quantifier causes computational overhead and so they do not introduce it for the sake of computational simplicity and tractability. However, the TIGER language has subsequently been successfully extended with universal quantification [5].

In the scope of our project it is crucial to be able not only to search for elements present in the trees but also to check for absent elements. For the present version of the search tool we have decided just to extend the possibility of introducing negation in queries. Namely the negation operator `!` when applied to nodes has the meaning "there exists no node". For example the query

```
#n: [cat="zdanie"] > [cat="ff"] & (! #n > [tfw="subj"])
```

finds all finite sentences without a subject. Literally it searches for all nodes of category 'zdanie' (sentence) which have a finite verbal phrase `ff` as a direct descendant and among such direct descendants there is no phrase marked as the subject.

Note that `#n` in this query is defined outside the scope of the `!` operator, so the "there exists no" applies only to the node `[tfw="subj"]`.

We can combine this mechanism with facilities described in the previous section and search, e.g., for sentences which do not have a subject according to the annotators, although a subject is present in some interpretations generated by the parser:

```
#n: [cat="zdanie"] > [cat="ff"] &
(! #n > [tfw="subj"]) & #n > [[tfw="subj"]]
```

5 Searching Trees with SQL

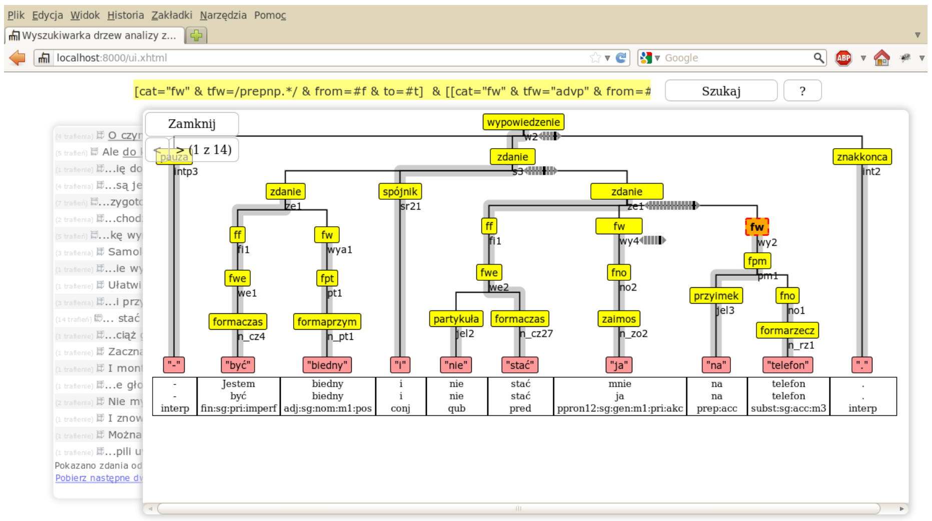
In contrast to the original TIGER Search our tool is built on top of an SQL engine, whose indexing facilities provide for efficient searches.

The query entered by the user is translated into an SQL query using joins in case of multiple node specifications (each node is represented by one row in the `node` table). Feature specifications are mapped to `where` clauses in SQL. The most fundamental node attributes like the category and the span of the node are represented with dedicated columns in the node table. Other features, which can differ from category to category, are conveniently stored in a single field of type `hstore`. In PostgreSQL databases this type can be used to store arbitrary sets of key-value pairs. What is important, PostgreSQL server provides for indexing values of this type, which results in efficient queries.

```
SELECT filename, tree.id, n0.id, n1.id
FROM node AS n0, node AS n1, tree
WHERE n0.treeid = n1.treeid
  AND (n0.sel AND n0.category = 'zdanie')
  AND (n1.sel AND n1.f::hstore -> 'tfw'='subst')
  AND n1.id = ANY(n0.children)
  AND n0.treeid = tree.id
```

Fig. 2. The result of converting the query `[cat="zdanie"] > [tfw="subst"]` to SQL. The feature `cat` is represented with a designated column, the feature `tfw` is stored in a `hstore` container.

The user interacts with the search tool via a standard web browser (currently we support Mozilla Firefox and WebKit based browsers including Google Chrome and Safari). The trees are displayed using a JavaScript library developed in the project and used also in our treebank development tool [9].



[cat="fw" & tfw=/prepnp.*/ & from=#f & to=#t] & [[cat="fw" & tfw="advp" & from=#

-	Jestem	biedny	i	nie	stać	mnie	na	telefon	.
-	być	biedny	i	nie	stać	ja	na	telefon	.
interp	fin:sg:pri:imperf	adj:sg:nom:m1:pos	conj	qub	pred	ppron12:sg:gen:m1:pri:akc	prep:acc	subst:sg:acc:m3	interp

Fig. 3. Results of a query displayed in the web interface

Current implementation covers a large subset of the TIGER query language including conditions on node features expressed with equality and regular expressions, direct and indirect dominance and precedence (including labelled dominance), node and feature variables and their unification, boolean expressions over node and feature specifications.

We haven't implemented template definitions nor type definitions. In fact, our tool does not consider features typed. All values are treated as simple strings. A system of query templates would help in stating complex queries, so we will probably consider it in the future. Note, however, that neither of these elements increases the expressive power of the language.

Our implementation strategy is admittedly simple: we push the problem of effective finding of matching nodes to the specialised software namely to the database engine. It seems that PostgreSQL performs very well in this context. This is probably due to its sophisticated query optimiser.

We have run a few tests comparing performance of the original implementation of TIGER Search and our tool. The treebank consists of about 8200 sentences with varying numbers of trees. The total number of nodes in all parse forests is over 2,500,000. In the experiment, our tool explored the complete set of nodes, while TIGER Search used only about 300,000 nodes selected by annotators. For queries using 3–5 node specifications and conditions including (indirect) dominance and precedence our tool gives answers in 5–15 seconds. TIGER Search gives analogous answers in 3–8 seconds. Answer times of our tool are longer, but the data set used is significantly larger. We do not observe any influence of the "not exists" operator on execution time, as queries involving this operator compute in similar times.

6 Conclusions

The tool presented in this paper has been created using a seemingly simple technique, and yet it has proved stable and effective enough to be used as the main search interface of our treebank. Current implementation covers all important features of the TIGER query language. Moreover, our language already has a larger expressive power with respect to addressing ambiguous structures and negation. We extend the language with new operators as need arises, one example being the `same_tree` predicate. We expect the tool to evolve together with the Składnica treebank.

The program is free software. It is available for download from the Składnica webpage `http://zil.ipipan.waw.pl/Składnica`. A running installation ready to query our treebank is also present there.

References

1. Billot, S., Lang, B.: The structure of shared forests in ambiguous parsing. In: Meeting of the Association for Computational Linguistics. pp. 143–151 (1989)
2. Janus, D., Przepiórkowski, A.: Poliqarp 1.0: Some technical aspects of a linguistic search engine for large corpora. In: Waliński, J., Kredens, K., Goźdź-Roszkowski, S. (eds.) The proceedings of Practical Applications of Linguistic Corpora 2005. Peter Lang (2006)

3. König, E., Lezius, W., Voormann, H.: TIGERSearch 2.1 user's manual. Tech. rep., IMS, Universität Stuttgart, Germany (2003)
4. Lezius, W.: TIGERSearch — ein Suchwerkzeug für Baumbanken. In: Busemann, S. (ed.) Proceedings der 6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002). Saarbrücken (2002)
5. Marek, T., Lundborg, J., Volk, M.: Extending the TIGER query language with universal quantification. In: KONVENS 2008: 9. Konferenz zur Verarbeitung natürlicher Sprache. pp. 5–17 (2008)
6. Maryns, H., Kepser, S.: MonaSearch — a tool for querying linguistic treebanks. In: Van Eynde, F., Frank, A., De Smedt, K. (eds.) Treebanks and Linguistic Theories 2009. pp. 29–40 (2009)
7. Rohde, D.L.T.: Tgrep2 user manual (2005), `http://tedlab.mit.edu/~dr/Tgrep2/`
8. Świdziński, M., Woliński, M.: Towards a bank of constituent parse trees for Polish. In: Sojka, P. (ed.) Text, Speech and Dialogue, 13th International Conference, TSD 2010, Brno, September 2010, Proceedings. LNAI, vol. 6231, pp. 197–204. Springer, Heidelberg (2010)
9. Woliński, M.: Dendrarium — an open source tool for treebank building. In: Kłopotek, M.A., Marciniak, M., Mykowiecka, A., Penczek, W., Wierzchoń, S.T. (eds.) Intelligent Information Systems, pp. 193–204. Siedlce, Poland (2010)
10. Woliński, M., Głowińska, K., Świdziński, M.: A preliminary version of Składnica — a treebank of Polish. In: Vetulani, Z. (ed.) Proceedings of the 5th Language & Technology Conference. pp. 299–303. Poznań (2011)