SETS: Scalable and Efficient Tree Search in Dependency Graphs

Juhani Luotolahti[1], **Jenna Kanerva**[1,2], **Sampo Pyysalo**[1] **and Filip Ginter**[1]
[1]Department of Information Technology
[2]University of Turku Graduate School (UTUGS)
University of Turku, Finland
first.last@utu.fi

Abstract

We present a syntactic analysis query toolkit geared specifically towards massive dependency parsebanks and morphologically rich languages. The query language allows arbitrary tree queries, including negated branches, and is suitable for querying analyses with rich morphological annotation. Treebanks of over a million words can be comfortably queried on a low-end netbook, and a parsebank with over 100M words on a single consumer-grade server. We also introduce a web-based interface for interactive querying. All contributions are available under open licenses.

1 Introduction

Syntactic search is one of the basic tools necessary to work with syntactically annotated corpora, both manually annotated treebanks of modest size and massive automatically analyzed parsebanks, which may go into hundreds of millions of sentences and billions of words. Traditionally, tools such as *TGrep2* (Rohde, 2004) and *TRegex* (Levy and Andrew, 2006) have been used for tree search. However, these tools are focused on constituency trees annotated with simple part-of-speech tags, and have not been designed to deal with dependency graphs and rich morphologies. Existing search systems are traditionally designed for searching from treebanks rarely going beyond million tokens. However, treebank sized corpora may not be sufficient enough for searching rare linguistic phenomena, and therefore ability to cover billion-word parsebanks is essential. Addressing these limitations in existing tools, we present SETS, a toolkit for search in dependency treebanks and parsebanks that specifically emphasizes expressive search of dependency graphs including detailed morphological analyses, simplicity of querying, speed, and scalability.

Operator	Meaning
<	governed by
>	governs
<@L	governed by on the left
<@R	governed by on the right
>@L	has dependent on the left
>@R	has dependent on the right
!	negation
& |	and / or
+	match if both sets not empty
->	universal quantification

Table 1: Query language operators.

2 Demonstration outline

We demonstrate the query system on the set of all available Universal Dependencies[1] treebanks, currently covering 10 languages with the largest treebank (Czech) consisting of nearly 90K trees with 1.5M words. We demonstrate both the command line functionality as well as an openly accessible web-based interface for the graph search and visualization on multiple languages. We also demonstrate how new treebanks in the CoNLL formats are added to the system.

3 Query language

The query language is loosely inspired by TRegex, modified extensively for dependency structures. Each query specifies the words together with any restrictions on their tags or lemmas, and then connects them with operators that specify the dependency structure. Table 1 shows the operators defined in the query language, and Table 2 illustrates a range of queries from the basic to the moderately complex.

[1] universaldependencies.github.io/docs. Note that while the SETS system is completely generic, we here use UD tagsets and dependency relations in examples throughout.

Proceedings of NAACL-HLT 2015, pages 51–55,
Denver, Colorado, May 31 – June 5, 2015.

Target	Query
The word *dog* as subject	`dog <nsubj _`
A verb with *dog* as the object	`VERB >dobj dog`
A word with two nominal modifiers	`_ >nmod _ >nmod _`
A word with a nominal modifier that has a nominal modifier	`_ >nmod (_ >nmod _)`
An active verb without a subject	`VERB&Voice=Act !>nsubj _`
A word which is a nominal modifier but has no adposition	`_ <nmod _ !>case _`
A word governed by *case* whose POS tag is not an adposition	`!ADP <case _`

Table 2: Example queries.

The query language is explained in detail in the following.

3.1 Words

Word positions in queries can be either unspecified, matching any token, or restricted for one or more properties. Unspecified words are marked with the underscore character. Lexical token restrictions include wordform and lemma. Wordforms can appear either as-is (`word`) or in quotation marks (`"word"`). Quotation marks are required to disambiguate queries where the wordform matches a feature name, such as a query for the literal word *NOUN* instead of tokens with the `NOUN` POS tag. Words can be searched by lemma using the `L=` prefix: for example, the query `L=be` matches all tokens with the lemma *(to) be*.

Words can also be restricted based on any tags, including POS and detailed morphological features. These tags can be included in the query as-is: for example, the query for searching all pronouns is simply `PRON`. All word restrictions can also be negated, combined arbitrarily using the *and* and *or* logic operators, and grouped using parentheses. For example, `(L=climb|L=scale)&VERB&!Tense=Past` searches for tokens with either *climb* or *scale* as lemma whose POS is verb and that are not in the past tense.

3.2 Dependency relations

Dependency relations between words are queried with the dependency operators (< and >), optionally combined with the dependency relation name. For example, the query to find tokens governed by an *nsubj* relation is `_ <nsubj _`, and tokens governing an *nsubj* relation can be searched with `_ >nsubj _`. The left-most word in the search expression is always the target, and is identified in the results. While the two preceding *nsubj* queries match the same graphs, they thus differ in the target token. To constrain the linear direction of the dependency relation, the operators `@R` and `@L` can be used, where e.g. `_ >nsubj@R _` means that the token must have a *nsubj* dependent to the right.

Negations and logical operators can be applied to the dependency relations in the same manner as to words. There are two different ways to negate relations; the whole relation can be negated, as in `_ !>nsubj _`, which means that the tokens may not have an *nsubj* dependent (not having any dependent is allowed), or only the type can be negated, as in `_ >!nsubj _`, where the token must have a dependent but it cannot be *nsubj*. Tokens which have either a nominal or clausal subject dependent can be queried for with the logical *or* operator: `_ >nsubj|>csubj _`.

Subtrees can be identified in the search expression by delimiting them with parentheses. For example, in `_ >nmod (_ >nmod _)`, the target token must have a nominal modifier which also has a nominal modifier (i.e a chain of two modifiers), whereas in `_ >nmod _ >nmod _` the token must have two (different) nominal modifiers. Note that queries such as `_ >nmod _ >nmod _` are interpreted so that all sibling nodes in the query must be unique in the match to guarantee that the restriction is not satisfied twice by the same token in the target tree.

There is no restriction on the complexity of subtrees, which may also include any number of negations and logical operators. It is also possible to negate entire subtrees by placing the negation operator `!` before the opening parenthesis.

3.3 Sentence

The more general properties of the sentence instead of just the properties of certain token, can be queried using the operators +, match a sentence if both sets are not empty and ->, universal quantification – operators. For example, if we wanted to find a sentence where all subject dependents are in the third person, we could query `(_ <nsubj _) -> (Person=3 <nsubj _)`. And to find sentences where we have a token with two nmod dependents and a word `dog` somewhere in the sentence we could query `(_ >nmod _ >nmod _) + "dog"`.

4 Design and implementation

The scalability and speed of the system stem from several key design features, the most important of which is the that every query is used to generate an algorithmic implementation that is then compiled into native binary code, a process which takes typically less than a second. Search involves the following steps:

1) The user query is translated into a sequence of set operations (intersection, complement, etc.) over tokens. For example, a query for tokens that are in the partitive case and dependents of a subject relation is translated into an intersection of the set of partitive case tokens and the set of subject dependents. Similarly, negation can in most cases be implemented as the set complement. The code implementing these operations is generated separately for each query, making it possible to only include the exact operations needed to execute each specific query.

2) The code implementing this sequence of operations is translated into C by the Cython compiler. The set operations are implemented as bit operations on integers (bitwise and, or, etc.) and can thus be executed extremely fast.

3) An SQL statement is generated and used to fetch from a database the token sets that are needed to evaluate the query. The query retrieves the token sets only for those trees containing at least one token meeting each of the restrictions (dependency relations, morphological tags, etc.).

4) The sequence of set operations implementing the query is used to check whether their configuration matches the query. For each match, the whole tree is retrieved from the database, reformatted and output in the CoNLL-U format.

The data is stored in an embedded database as precomputed token sets, with separate sets for all different lemmas, wordforms, and morphological features. These sets are stored as native integers with each bit corresponding to a single token position in a sentence. Since the vast majority of sentences are shorter than 64 words, these sets typically fit into a single integer. However, the system imposes no upper limit on the sentence length, using several integers when necessary.

The system uses SQLite as its database backend and the software is written as a combination of Python, Cython and C++. Cython enables easy integration of Python code with fast C-extensions, vital to assure the efficiency of the system. As it uses the embedded SQLite database, the system is fully self-contained and requires no server applications.

In addition to the primary search system, we created a simple browser-based frontend to the query system that provides a dynamic visualization of the retrieved trees and the matched sections (Figure 1). This interface was implemented using the Python Flask[2] framework and the BRAT annotation tool (Stenetorp et al., 2012).

5 Benchmarks

Our graph-search tool is tested and timed on three different machines and two datasets. Evaluation platforms include a server-grade machine with good resources, a standard laptop computer and a small netbook with limited performance. To compare the efficiency of our system to the state-of-the-art treebank searching solutions, we employ ICARUS (Gärtner et al., 2013) search and visualization tool which also focuses on querying dependency trees. ICARUS system loads the data into the computer's main memory, while our system uses a database, which is optimized by caching. The comparison of our graph-search tool and the ICARUS baseline is run on server machine with a dataset of roughly 90K trees.

Three test queries are chosen so that both systems support the functionality needed in or-

[2] http://flask.pocoo.org/

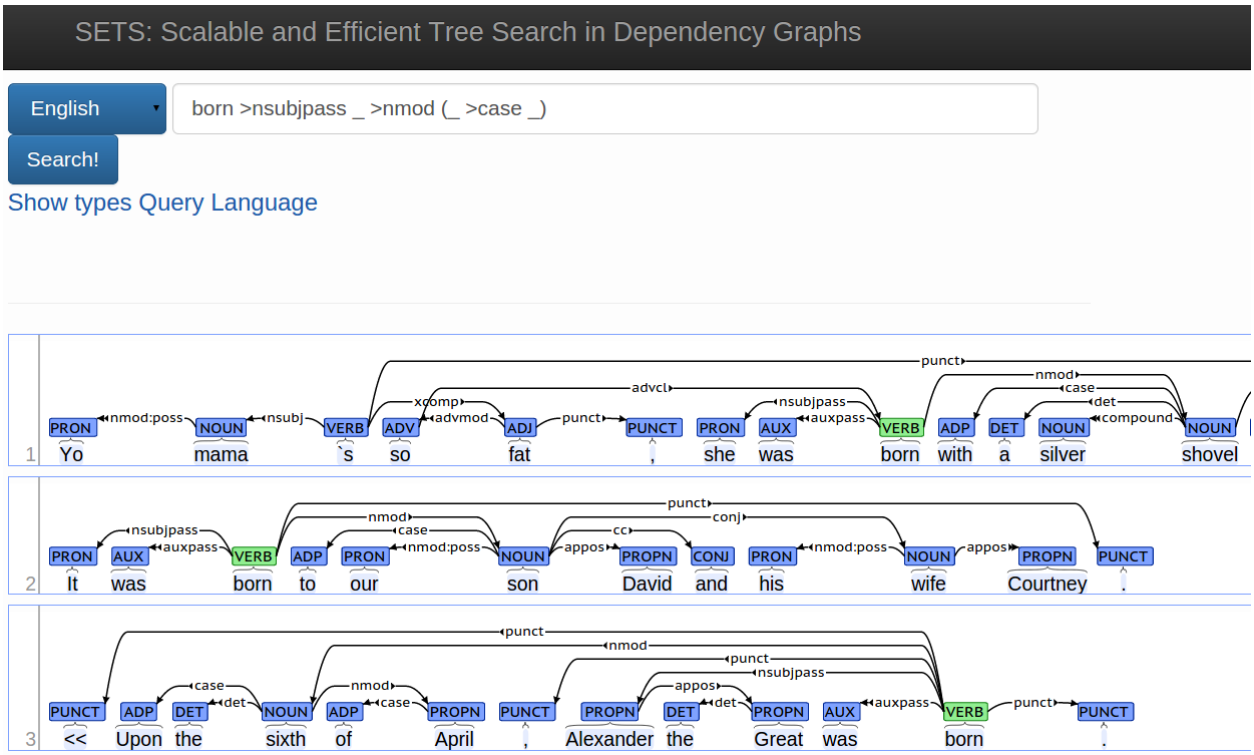

Figure 1: Web interface showing trees in Finnish.

der to run the tests. The first query is a straightforward search for all subject dependents (`_ <nsubj _`) and the second query adds a lexical restraint to it and requires the lemma to be I (`L=I <nsubj _`). The third query is much more complex and is inspired by an actual linguistic use case to find examples of an exceptionally rare transitive verb usage in Finnish. The query includes chaining of dependencies and a negation (`_ >nsubj (Case=Gen > _) >dobj ... _ !<xcomp _`).

	Query 1	Query 2	Query 3
ICARUS	2m30s	2m30s	2m30s
SETS	1.61s	1.2s	2.18s

Table 3: The speed of our system compared to the baseline on the three different test queries when a treebank of about 90K sentences is used.

As can be seen from Table 3, when our system and the baseline system are tested on the server machine using the three example queries our system clearly outperforms the baseline. The speed of the baseline seems to be relatively unaffected by the complexity of the query, suggesting a bottle-neck somewhere else than tree-verification. It should be noted that these measurements are only to illustrate the relative speed and performance differences, and are subject to change depending on system cache. Due to their architecture, neither system has a major advantage in the use of memory and the results are broadly comparable.

Our system is also tested on a standard laptop, and a netbook using the same three queries and the same input corpus. The first test query was finished by a netbook in 37 seconds, the third query, most complex of them, was finished in 13.5 seconds. The laptop finished the first query in 16 seconds, the second in 7 seconds and the third in 16 seconds.

As our system is meant for searching from very large corpora, we test it with a parsebank of 10 million trees and over 120 million tokens. A variant of the test query number 3, the most complex of the queries, was executed in time between 1m52s and 48s (depending the system cache). The test query 1 took from 5m10s to 4m30s and the lexicalized version (query 2) from 12s to 9s. The test queries were performed on the same server-machine as the runs shown in Table 3.

Since our system uses pre-indexed databases the disk space needed for holding the data slightly increases. Indexing the 90K sentence treebank used in our tests requires about 550M of free disk space, whereas indexing the 10 million sentence parsebank uses 35G of space.

6 Conclusion

We have presented a syntax query system especially geared towards very large treebanks and parsebanks. In the future, we will implement support for graph queries, e.g. coindexing of the tokens, since many treebanks have multiple layers of dependency structures. Related to this goal, we aim to include support for properties of the tokens and dependencies, for example the annotation layer of the dependency, word sense labels, etc.

The full source code of the system is available under an open license at `https://github.com/fginter/dep_search`. Additionally, we maintain a server for public online search in all available Universal Dependencies treebanks (Nivre et al., 2015) at `http://bionlp-www.utu.fi/dep_search`.

Acknowledgments

This work was supported by the Kone Foundation and the Emil Aaltonen Foundation. Computational resources were provided by CSC – IT Center for Science.

References

[Gärtner et al.2013] Markus Gärtner, Gregor Thiele, Wolfgang Seeker, Anders Björkelund, and Jonas Kuhn. 2013. Icarus – an extensible graphical search tool for dependency treebanks. In *Proceedings of Demonstrations at ACL'13*, pages 55–60.

[Levy and Andrew2006] Roger Levy and Galen Andrew. 2006. Tregex and Tsurgeon: tools for querying and manipulating tree data structures. In *Proceedings of LREC'06)*.

[Nivre et al.2015] Joakim Nivre, Cristina Bosco, Jinho Choi, Marie-Catherine de Marneffe, Timothy Dozat, Richárd Farkas, Jennifer Foster, Filip Ginter, Yoav Goldberg, Jan Hajič, Jenna Kanerva, Veronika Laippala, Alessandro Lenci, Teresa Lynn, Christopher Manning, Ryan McDonald, Anna Missilä, Simonetta Montemagni, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Maria Simi, Aaron Smith, Reut Tsarfaty, Veronika Vincze, and Daniel Zeman. 2015. Universal dependencies 1.0.

[Rohde2004] Douglas L. T. Rohde, 2004. *TGrep2 User Manual*. Available at http://tedlab.mit.edu/~dr/Tgrep2.

[Stenetorp et al.2012] Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. 2012. Brat: a web-based tool for nlp-assisted text annotation. In *Proceedings of Demonstrations at EACL'12*, pages 102–107.