Dep_search: Efficient Search Tool for Large Dependency Parsebanks

Juhani Luotolahti[1,2], **Jenna Kanerva**[1,2], **and Filip Ginter**[1]
[1]Turku NLP Group
[2]University of Turku Graduate School (UTUGS)
University of Turku, Finland
mjluot@utu.fi, jmnybl@utu.fi, figint@utu.fi

Abstract

We present an updated and improved version of our syntactic analysis query toolkit, *dep_search*, geared towards morphologically rich languages and large parsebanks. The query language supports complex searches on dependency graphs, including for example boolean logic and nested queries. Improvements we present here include better data indexing, especially better database backend and document metadata support, API access and improved web user interface. All contributions are available under open licences.

1 Introduction

Huge text collections crawled from the Internet have become a popular resource for natural language processing and linguistic studies. Recently, massive corpora with automatically analyzed full syntactic structure became available for 45 languages (Ginter et al., 2017). To efficiently find and access specific types of sentences or syntactic structures from corpora with billions of tokens, powerful search systems are needed for both text and tree based queries, and combinations thereof.

The SETS dependency tree search tool (Luotolahti et al., 2015) is developed for efficient and scalable tree search in dependency treebanks and parsebanks, supporting complex searches on words, lemmas, detailed morphological analyses and dependency graphs. It is implemented using efficient data indexing and turning search expressions into compiled code.

In this paper we present the *Turku dep_search* tool, an improved version of SETS. In addition to changing the name, our main contributions are: 1) major speed-up and reduction of stored index size by changing backend from SQLite to Solr[1] and LMDB[2]; 2) API access and improved web user interface; 3) supporting arbitrary sentence/-document metadata enabling for example subcorpora and multilingual searches; 4) Universal Dependencies[3] treebanks and automatically analyzed UD parsebank data for 45 languages publicly available through dep_search API and online interface.

2 Query Language

The query language is in many ways inspired by languages used in existing tree query software such as TGrep (Rohde, 2004). The query language defines criteria for tokens and their relations in dependency graphs to be queried. It allows the user to specify graph structures in the syntactic trees, tokens by their properties, linear order of the tokens, and combinations thereof. Boolean logic is supported and queries can be combined and nested. The query itself is parsed into a tree structure, reflecting the nature of syntactic trees, and it is transformed into executable code with Cython[4]. The query parser is able to differentiate between token word forms and tags present in the queried corpora.

The query language is best presented by examples. Arguably the most simple query is querying for all tokens. This is achieved by the query _, underscore representing any token. To restrict the token by a word form or a tag the query is the word form itself, for example, to query the word *cat* the expression is `cat`. The lemma *cat* can be queried with the expression `L=cat`. Similarly to find tags, a query to find *nouns* is `NOUN`. Token queries can easily be combined using boolean operators. For example to search for a token with the lemma *cat* or *dog*, that is not in genitive case, one can query: `(L=cat | L=dog) & !Case=Gen`.

[1]http://lucene.apache.org/solr/

[2]http://www.lmdb.tech/doc/

[3]http://universaldependencies.org/

[4]http://cython.org

Proceedings of the 21st Nordic Conference of Computational Linguistics, pages 255–258,
Gothenburg, Sweden, 23-24 May 2017.

We can add dependency restrictions to our query to search for syntactic structures in the dependency graphs. A simple query like this, looks like `cat > _`, which finds tokens with word form *cat* with a dependent token. In a similar vein, we can search for a token with a typed dependent, for example to find the token *cat* with an *amod* dependent, query is: `cat >amod _`. The query for *cat* with two separate *amod* dependents query is `cat >amod _ >amod _` and to look for chains of *amod* dependencies from the token *cat*, one needs to use parenthesis in their query: `cat >amod (_ amod> _)`.

Boolean operators can be used with dependencies similarly as they can be used with the tokens. To negate a dependency relation, two options are offered: The query `cat !>amod _` looks for token *cat* without an *amod* dependent, where as the query `cat >!amod _` searches for the token *cat* with a dependent not the type of *amod*. Since all parts of the query support boolean logic, subtrees can be negated, for example querying `cat >amod !pretty` would find token *cat* if it has an *amod* dependent which is anything but the word *pretty*. Besides using negation, one can use OR operator to query dependency relations. For example, the query `cat >amod|>nmod _` finds *cat* with *amod* or *nmod* dependents.

The third class of query operators has to do with the sentence; linear order of the tokens and set operations on subqueries. To query for tokens next to each other in the sentence, query syntax is: `first . second`, query to search for tokens in a window is `cat <lin_2:3 NOUN`, which finds *cat* - tokens with a *noun* within two to three tokens from it. The operation can be limited to a particular direction by adding @R/@L - operator. The previous query limited to only tokens to the right is: `cat <lin_2:3@R NOUN`. The @R/@L -operator can also be applied to all dependency types. The universal quantifier operator (->) allows searching for sentences in which all tokens of certain type have a property. For example query: `(_ <nsubj _) -> (Person=3 <nsubj _)`, finds sentences in which all subjects are in third person. The plus operator allows us to find sentences with multiple properties, eg. to find sentences with both tokens *cat* and *dog*, where *dog* is a subject, query is: `(dog <nsubj _) + cat`. In addition to these, the size of the result can be set by adding {`len_set=limit`} after the query. For example: `Clitic=Han {len_set=2}`.

Operator	Meaning
`<, >`	governed by, governs
`<@L, <@R`	governed by on the left, right
`>@L, >@R`	has dependent on the left, right
`.`	tokens are next to each other in linear order
`<lin_s:e`	tokens are in s:e distance from each other
`!, &, |`	negation, and, or
`+`	match if both sets not empty
`->`	universal quantification

Table 1: Query language operators.

3 Design

The search is executed in two main steps: 1) fetching candidate sentences from the indexed data, so that in a returned candidate sentence, all restrictions must be individually met, and 2) evaluating these candidates to check whether the configuration of the sentence fully matches the query. As the full configuration evaluation (part 2) stays mostly untouched compared to earlier version of the search tool, it is only briefly discussed here, and more detailed information can be found from Luotolahti et al. (2015).

For fast retrieval of candidate sentences, we use the Solr search engine to return a list of sentence ids, where a sentence has to match all query restrictions individually (for example, sentence has a specific word, morphological tag, and/or relation type), but no relations of these individual restrictions are evaluated at this point. For fast retrieval of candidate sentences, an index is build individually for all possible attributes (e.g. words, lemmas, morphological features and dependency relations). The actual sentence data is stored separately in a fast memory-mapped database, LMDB, where for each sentence the data is already stored in the binary form used when the full sentence configuration is evaluated. The sentences can be fetched from the LMDB using sentence ids given by the search engine.

Together with the different attribute indices, the search engine index can be used to store any necessary metadata related to sentences and documents and further restrict the search also on metadata level. Metadata can be used for example to restrict the search to a specific language, sub corpora or time span, naturally depending on the metadata available for a corpus in use. This way we can keep all data from different corpora and languages in one database, giving us also the possibility to search similar structures across languages.

Especially in the case of Universal Dependencies, cross-linguistically consistent treebank annotation, this gives a great opportunity to study similar structures across languages without compromising on speed or ability to limit the set of interesting languages.

After fetching the candidate sentences, their full configuration is evaluated against the actual search expression. The search expression is turned into a sequence of set operations, which can be compiled into native binary code. This compilation is done only once in the beginning of the search, taking typically less than a second. All sentences passing this step are then returned to the user.

4 Benchmarks

Generally, by changing the database backend we are able to gain in terms of speed and disk space. We are also able to remove two major bottleneck queries.

In the previous version of the search tool, one major bottleneck was queries involving rare lexical restrictions. When the SQLite backend was used, the data index needed to be divided into multiple small databases in order to keep retrieving fast. This however resulted slower queries when rare lexical items were queried because it was needed to iterate through many of these small databases with very few hits in each in order to find enough hits for the user. The new Solr backend is able to hold the whole data in one index, giving us a major speed-up when rare lexical items are searched.

Even bigger speed-up in the new version of dep_search is noticed when the query involves OR statements. Solr handles alternative restrictions much faster than SQLite is able to do, removing the most computationally heavy part of these queries.

In addition to faster queries, dep_search needs now much less disk space for the data index. In the index of 20M trees the disk usage is ∼45G, which is about half of the size compared to what the old version was using.

5 Web User Interface and API

In addition to the search tool, we also provide a graphical web interface, which can be set to talk to dep_search API, and render results in browser using Python Flask[5] library, Ajax and BRAT annotation tool (Stenetorp et al., 2012). Dep_search API is accessed through http, and receives the query and the database name, and returns the matching trees as a response.

We maintain a public server for online searches at `http://bionlp-www.utu.fi/dep_search/`, where we have indexed all 70 Universal Dependencies v2.0 treebanks (Nivre et al., 2017), as well as automatically analyzed parsebank data for all the 45 languages present in UD parsebank collection. For each of these 45 languages, 1M sentences are made available in the dep_search web interface. Additionally, dep_search is used through its API to provide automatic content validation in the Universal Depedencies project,[6] automatically reindexed upon any update of a UD treebank development repository. In the past 6 weeks, the server processed over 11,000 requests — mostly related to the automated UD validation.

6 Conclusions

In this paper we presented the Turku dep_search dependency search tool with expressive query language developed to support queries involving rich morphological annotation and complex graph structures. The search tool is made scalable to parsebanks with billions of words using efficient data indexing to retrieve candidate sentences and generating algorithmic implementation of the actual search expression compiled into native binary code. Dep_search tool supports indexing varying sentence and document metadata making it possible e.g. to focus the search to a specific time span or a set of languages. Source code for the search backend and the web user interface is publicly available at `https://github.com/fginter/dep_search` and `https://github.com/fginter/dep_search_serve`, respectively.

Additionally, we provide a public, online search interface, where we host all Universal Dependencies version 2.0 treebanks, together with UD parsebank data for 45 different languages.

Acknowledgements

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

[5] `http://flask.pocoo.org/`

[6] `http://universaldependencies.org/svalidation.html`

References

Filip Ginter, Jan Hajič, Juhani Luotolahti, Milan Straka, and Daniel Zeman. 2017. CoNLL 2017 shared task - automatically annotated raw texts and word embeddings. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University in Prague.

Juhani Luotolahti, Jenna Kanerva, Sampo Pyysalo, and Filip Ginter. 2015. Sets: Scalable and efficient tree search in dependency graphs. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 51–55. Association for Computational Linguistics.

Joakim Nivre, Željko Agić, Lars Ahrenberg, Maria Jesus Aranzabe, Masayuki Asahara, Aitziber Atutxa, Miguel Ballesteros, John Bauer, Kepa Bengoetxea, Riyaz Ahmad Bhat, Eckhard Bick, Cristina Bosco, Gosse Bouma, Sam Bowman, Marie Candito, Gülşen Cebirolu Eryiit, Giuseppe G. A. Celano, Fabricio Chalub, Jinho Choi, Çar Çöltekin, Miriam Connor, Elizabeth Davidson, Marie-Catherine de Marneffe, Valeria de Paiva, Arantza Diaz de Ilarraza, Kaja Dobrovoljc, Timothy Dozat, Kira Droganova, Puneet Dwivedi, Marhaba Eli, Tomaž Erjavec, Richárd Farkas, Jennifer Foster, Cláudia Freitas, Katarína Gajdošová, Daniel Galbraith, Marcos Garcia, Filip Ginter, Iakes Goenaga, Koldo Gojenola, Memduh Gökrmak, Yoav Goldberg, Xavier Gómez Guinovart, Berta Gonzáles Saavedra, Matias Grioni, Normunds Grūzītis, Bruno Guillaume, Nizar Habash, Jan Hajič, Linh Hà M, Dag Haug, Barbora Hladká, Petter Hohle, Radu Ion, Elena Irimia, Anders Johannsen, Fredrik Jørgensen, Hüner Kaşkara, Hiroshi Kanayama, Jenna Kanerva, Natalia Kotsyba, Simon Krek, Veronika Laippala, Phng Lê Hng, Alessandro Lenci, Nikola Ljubešić, Olga Lyashevskaya, Teresa Lynn, Aibek Makazhanov, Christopher Manning, Cătălina Mărănduc, David Mareček, Héctor Martínez Alonso, André Martins, Jan Mašek, Yuji Matsumoto, Ryan McDonald, Anna Missilä, Verginica Mititelu, Yusuke Miyao, Simonetta Montemagni, Amir More, Shunsuke Mori, Bohdan Moskalevskyi, Kadri Muischnek, Nina Mustafina, Kaili Müürisep, Lng Nguyn Th, Huyn Nguyn Th Minh, Vitaly Nikolaev, Hanna Nurmi, Stina Ojala, Petya Osenova, Lilja Øvrelid, Elena Pascual, Marco Passarotti, Cenel-Augusto Perez, Guy Perrier, Slav Petrov, Jussi Piitulainen, Barbara Plank, Martin Popel, Lauma Pretkalnia, Prokopis Prokopidis, Tiina Puolakainen, Sampo Pyysalo, Alexandre Rademaker, Loganathan Ramasamy, Livy Real, Laura Rituma, Rudolf Rosa, Shadi Saleh, Manuela Sanguinetti, Baiba Saulīte, Sebastian Schuster, Djamé Seddah, Wolfgang Seeker, Mojgan Seraji, Lena Shakurova, Mo Shen, Dmitry Sichinava, Natalia Silveira, Maria Simi, Radu Simionescu, Katalin Simkó, Mária Šimková, Kiril Simov, Aaron Smith, Alane Suhr, Umut Sulubacak, Zsolt Szántó, Dima Taji, Takaaki Tanaka, Reut Tsarfaty, Francis Tyers, Sumire Uematsu, Larraitz Uria, Gertjan van Noord, Viktor Varga, Veronika Vincze, Jonathan North Washington, Zdeněk Žabokrtský, Amir Zeldes, Daniel Zeman, and Hanzhi Zhu. 2017. Universal dependencies 2.0. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University in Prague.

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

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 the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 102–107. Association for Computational Linguistics.