Efficient Dependency Graph Matching with the IMS Open Corpus Workbench

Thomas Proisl, Peter Uhrig

University of Erlangen-Nürnberg
Interdisciplinary Centre for Research on Lexicography, Valency and Collocation
Bismarckstr. 1, 91054 Erlangen
thomas.proisl@linguistik.uni-erlangen.de, peter.uhrig@angl.phil.uni-erlangen.de

Abstract

State-of-the-art dependency representations such as the Stanford Typed Dependencies may represent the grammatical relations in a sentence as directed, possibly cyclic graphs. Querying a syntactically annotated corpus for grammatical structures that are represented as graphs requires graph matching, which is a non-trivial task. In this paper, we present an algorithm for graph matching that is tailored to the properties of large, syntactically annotated corpora. The implementation of the algorithm is built on top of the popular IMS Open Corpus Workbench, allowing corpus linguists to re-use existing infrastructure. An evaluation of the resulting software, CWB-treebank, shows that its performance in real world applications, such as a web query interface, compares favourably to implementations that rely on a relational database or a dedicated graph database while at the same time offering a greater expressive power for queries. An intuitive graphical interface for building the query graphs is available via the Treebank.info project.

Keywords: treebank, graph matching, dependency parsing

1. Introduction

Over the past few years, dependency parsers (or dependency annotation schemes) have become more and more popular in the NLP community. There is a wide range of parsers and converters available (see for instance Cer et al. (2010)) that allow the annotation of large quantities of text which then in turn can be used in information retrieval and similar tasks (e. g. biomedical text mining, c. f. Schneider et al. (2009)). There is however a considerable lack of easy-to-use, freely available interfaces that make the wealth of information provided by the parsers accessible to (non-computational) linguists. One notable exception, though not tailored specifically towards dependency graphs, is ANNIS2 (Zeldes et al., 2009), which however still needs some specialist's manual work to set up in conjunction with freely available parsers for English. The Treebank.info project (see next section) sets out to remedy this issue and to provide access to parsed corpora for everyone via an easy-to-use web interface. In order to be able to find subgraphs in the dependency graphs within a time-span acceptable to most users, CWB-treebank – the piece of software described in the present paper – was developed. It works as an add-on to the freely available IMS Open Corpus Workbench and has been made freely available via Launchpad.[1]

1.1. Treebank.info

The Treebank.info project[2] (Uhrig and Proisl, 2011a; Uhrig and Proisl, 2011b) allows users to upload their own corpora in plain text format and then processes the uploaded corpora in a pipeline consisting of sentence splitting, tokenization, pos-tagging, lemmatization, phrase structure parsing, and dependency parsing. All results are stored in a document database and then the dependency graphs (so-called *stemmata* (Tesnière, 1966, 15)) and all item-specific information (see listing 1) are exported to the Open Corpus Workbench, on which CWB-treebank operates as the back-end to the web-based query interface.

1.2. The IMS Open Corpus Workbench

The original IMS Corpus Workbench (Christ, 1994; Christ and Schulze, 1996) was developed at the Institute for Natural Language Processing in Stuttgart (IMS) in the mid-1990s. The software is still under active development and has been released under the GPL as the IMS Open Corpus Workbench.[3] One of the key components of CWB is CQP, the Corpus Query Processor, and its *de facto* standard query language that is also supported by other systems, e. g. Manatee (Rychlý, 2007). Besides its high speed, its efficient indexing and powerful query language render the IMS Open Corpus Workbench a promising basis for efficient dependency graph matching, even though it was not originally conceived for such a use case.

1.3. The problem

Although it is in principle usable with any dependency-type representation, the current version of our software makes use of Stanford Typed Dependencies (de Marneffe et al., 2006; de Marneffe and Manning, 2008b) in a version in which conjunct dependencies are propagated. Thus, the resulting graphs are not necessarily trees and may even be cyclic (de Marneffe and Manning, 2008a, 18). Accordingly, it is not possible to use established, efficient (i. e. requiring only polynomial time) methods for matching trees (see Shamir and Tsur (1999)

[1] `https://launchpad.net/cwb-treebank`
[2] `http://treebank.info`
[3] http://cwb.sourceforge.net/

```
<s id="HHB_2310">
The      DT   the      DET        |det(1, 0)|       |
thought  NN   thought  SUBST      |nsubj(9, 0)|  |det(0, -1)|prep_of(0, 3)|prep_of(0, 5)|
of       IN   of       CONJ/PREP  |                  |
soiled   JJ   soiled   ADJ        |amod(1, 0)|       |
nappies  NNS  nappy    SUBST      |prep_of(-3, 0)| |amod(0, -1)|conj_and(0, 2)|
and      CC   and      CONJ/PREP  |                  |
vomit    NN   vomit    SUBST      |prep_of(-5, 0)|conj_and(-2, 0)|    |prep_on(0, 3)|
on       IN   on       CONJ/PREP  |                  |
my       PRP$ my       PRON       |poss(1, 0)|       |
clothes  NNS  clothes  SUBST      |prep_on(-3, 0)| |poss(0, -1)|
gives    VBZ  give     VERB       |                  |nsubj(0, -9)|iobj(0, 1)|dobj(0, 3)|
me       PRP  me       PRON       |iobj(-1, 0)|     |
the      DT   the      DET        |det(1, 0)|        |
horrors  NNS  horror   SUBST      |dobj(-3, 0)|     |det(0, -1)|
.        .    .        PUNC       |                  |
</s>
```

Listing 1: Vertical format for HHB 2310

for an overview). Since exactly matching a graph (or tree) against a larger graph is an NP-complete problem known as the subgraph isomorphism problem (Garey and Johnson, 1979, 202) and thus becomes quickly unfeasible for large graphs,[4] the implementation presented here relies on known properties of dependency graphs to reduce the search space and thus achieve higher performance.

Earlier prototypes of the system behind the Treebank.info project made use of relational databases (first MySQL, later PostgreSQL) and then moved on to a graph database (neo4j). However, as the evaluation (section 3.) will show, the performance of both earlier versions was not appropriate for interactive use over a web interface so that a dedicated and specialised solution had to be created. As pointed out above, the IMS Open Corpus Workbench appeared to be a suitable basis for such an application, even though it does not natively support syntactically annotated corpora.

2. Graph matching with the IMS Open Corpus Workbench

2.1. Corpus format

The Open Corpus Workbench requires a corpus to be available as verticalized text, i. e. in a format with one token per line, "with the surface form in the first column and token-level annotations specified as additional `TAB`-separated columns" (Evert and The OCWB Development Team, 2010a, 2). Let us illustrate this using the following example:[5]

(1) The thought of soiled nappies and vomit on my clothes gives me the horrors. [BNC: HHB 2310]

The Stanford Typed Dependencies analysis of example sentence (1) is shown in figure 1. Listing 1 shows

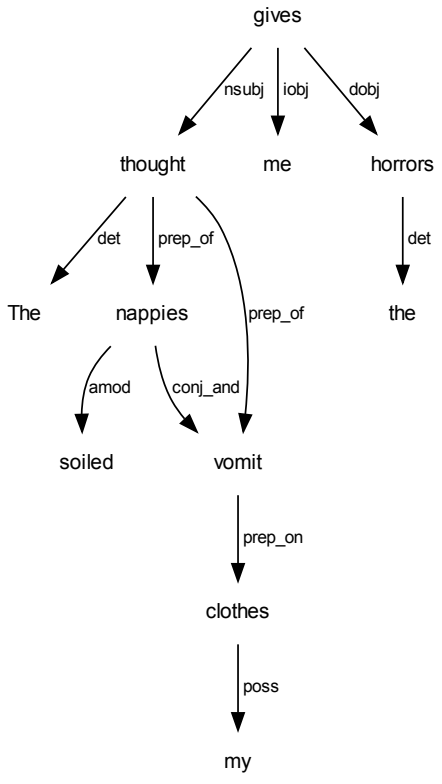


Figure 1: Stemma for HHB 2310

how the dependency relations between the words are encoded using feature set attributes (Evert and The OCWB Development Team, 2010b, 33–35). Incoming dependencies are encoded in the fifth column, outgo-

[4] A large graph is a graph that contains "hundreds or thousands of nodes" (Cordella et al., 2004, 1367), and sentences or units analysed as sentences in authentic natural language can easily exceed that number of tokens. Currently, the Treebank.info project imposes a 200 word limit due to the high memory requirements by parsers for longer sentences.

[5] The example has been taken from the British National Corpus (2007), distributed by Oxford University Computing Services on behalf of the BNC Consortium. All rights in the texts cited are reserved.

	0	1	2	3
0	lemma = “give”	relation = “iobj”	relation = “dobj”	
1		pos = “PRP”		
2			pos = “NNS”	relation = “det”
3				word = “the”

Table 1: Query as adjacency matrix

ing dependencies in the sixth column. Every dependency relation consists of a relation type, e. g. “det” or “prep_of”, and two relative offsets indicating the tokens functioning as governor and dependent. The feature `amod(0, -1)` in the sixth column of the token “nappies”, for example, indicates that the current token (0) governs an adjectival modifier (amod) that is immediately preceding (-1) it.

2.2. Query format

Dependency stemmata are represented as graphs. So in order to be able to reap the full benefits of a treebank, it should be possible to formulate queries as directed graphs that describe a particular linguistic structure, e. g. the lemma *give* in a ditransitive construction with a personal pronoun as indirect object and a plural noun with the determiner *the* as direct object. Using the query interface of Treebank.info, such a query can be formulated in an intuitive graphical way (cf. figure 2). Internally, a query graph is represented as an adjacency matrix (Cormen et al., 2009, 589–592).[6] Table 1 shows the matrix for our query. The nodes are represented on the diagonal, outgoing dependencies are in the same row as the node, incoming dependencies in the same column. To send it across the network to the CWB-treebank server, the matrix is serialized as a JSON[7] object.

In the current implementation, the query graphs are tailored to meet the needs of linguistic dependency graphs: Between any two nodes, there can only be one incoming and one outgoing dependency relation, i. e. any field in the matrix can only hold one dependency relation. This behavior is motivated by the fact that for example a noun cannot be direct object and indirect object of the same verb at the same time.[8]

To further increase the expressive power of the queries, every restriction can contain regular expressions (although the current frontend only makes use of regular expressions on the wordform/lemma) and various features can be negated, namely the wordform/lemma, the

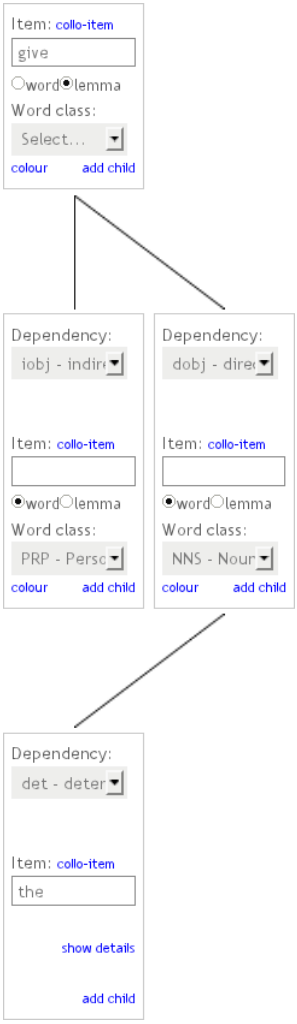


Figure 2: Query as dependency graph (screenshot from Treebank.info)

pos/word class, and outgoing dependency relations. It is thus possible with one single query to find all monotransitive uses of a given verb, i. e. sentences where the verb governs a direct object but neither an indirect object nor a prepositional phrase with “to” nor a phrasal verb particle. In the case of verbs such as *give*, this feature greatly improves precision since without the possibility of negation, a lot of noise, i. e. the much more frequent ditransitive uses, would also be retrieved.

2.3. The matching algorithm

In order to find the sentences that match the query and all corpus positions within the sentences that cor-

[6] Given that our query graphs are typically quite small, we can follow Cormen et al. (2009, 591), who argue that although the alternative representation as an adjacency list “is asymptotically at least as space-efficient as the adjacency-matrix representation, adjacency matrices are simpler, and so we may prefer them when graphs are reasonably small.”

[7] JavaScript Object Notation (JSON) is a text format for the interchange of structured data described in RFC 4627 (Crockford, 2006).

[8] Of course, this restriction can easily be lifted if it is not adequate for a given use case.

respond to nodes in the query graph, the following strategy is pursued:

1. Only look at constraints on nodes (e. g. word form, lemma, part-of-speech, word class, incoming/outgoing dependency relation) and find matching sentences.
2. For every sentence, create a look-up table that contains all the corpus positions that might correspond to the nodes in the query.
3. Recursively remove all unconnected items from the candidate lists and break if there are no candidates left for a given node in the query.
4. Output all possible mappings of query nodes to corpus positions.

2.3.1. Retrieving candidates

To keep the search space as small as possible and to avoid having to match a query unnecessarily against a large graph, we retrieve all candidate corpus positions that might correspond to a node in the query graph before we perform the actual matching. To do this, we have to find all sentences that match the node restrictions and remember the matching corpus positions. This can be done straightforwardly:

First, the frequencies of all the nodes in the query graph are determined independently by using only the restrictions on the nodes themselves, i. e. restrictions on word form, lemma, part of speech, word class, type of incoming or outgoing dependency relations. Then the nodes are sorted by their frequencies in ascending order. For the least frequent node, all matching corpus positions and sentence IDs are determined using a CQP query. The sentence IDs and corpus positions are stored in a data structure, then the active corpus in the Corpus Workbench is limited to the sentences that matched the query. The procedure is repeated on the restricted subcorpus for all remaining nodes in the query graph. By always limiting the corpus to the results of the previous query, the sentence IDs of the last query represent all sentences in the corpus that match all the node restrictions in the query graph. By beginning this process with the least frequent node, subcorpus size is minimized from the start.

Now, to find out whether these sentences also match the whole query graph and to determine the possible mappings of query nodes to corpus positions, the edges connecting the nodes have to be taken into account. So, for every sentence matching the node restrictions, the subroutine *match* (algorithm 1) is called with three arguments: the query graph, the index of the query node with which to start, i. e. 0,[9] and the candidate corpus positions for each query node.

[9] For simplicity, we assume that the first node has index 0, the second index 1, and so on. In practice, we follow Ullman (1976, 34–35) who suggests that it might be advantageous to order the nodes by decreasing degree, even though we were unable to measure a significant improvement in response time.

2.3.2. Filtering the corpus positions

Algorithm 1 Subroutine *match*

Input: *queryGraph*, *index*, *candidates*
Output: mappings of node indices from *queryGraph* to corpus positions from *candidates*

```
 1: if index > getLastIndex(queryGraph) then
 2:   output candidates
 3:   return
 4: end if
 5: queryDeps ← getDeps(queryGraph, index)
 6: for each cpos in candidates[index] do
 7:   localCands ← candidates \ cpos
 8:   localCands[index] ← cpos
 9:   corpusDeps ← getCorpusDeps(cpos)
10:   corpusCands ← [ ]
11:   for each queryDep in queryDeps do
12:     nodeIdx ← getGov(queryDep)
13:     for each corpusDep in corpusDeps do
14:       if queryDep == corpusDep then
15:         nodeCpos ← getNodeCpos(corpusDep)
16:         corpusCands[nodeIdx] ←
              corpusCands[nodeIdx] ∪ nodeCpos
17:       end if
18:     end for
19:   end for
20:   for idx = 0 to getLastIndex(queryGraph) do
21:     if size(corpusCands[idx]) > 0 then
22:       localCands[idx] ← localCands[idx]∩
            corpusCands[idx]
23:       next cpos if size(localCands[idx]) == 0
24:     end if
25:   end for
26:   match(queryGraph, index + 1, localCands)
27: end for
```

The *match* algorithm (algorithm 1) is similar to the classic algorithm by Ullman (1976) and the VF2 algorithm described in Cordella et al. (2004). A prominent difference to these algorithms is the fact that part of the work happening in the "refinement procedure" (Ullman, 1976, 33–35) or the "feasibility rules" (Cordella et al., 2004, 1368–1369) has already been done during candidate retrieval which benefits from efficient indexing on node properties.

In the *match* subroutine, the incoming and outgoing dependency relations of the query node specified by the value of *index* are retrieved from the query graph (5). The algorithm loops over all corpus positions (*cpos*) that are stored as candidates for the current query node (6). Within the loop, a local copy of *candidates* is created, in which the current *cpos* is removed from all nodes (7). The only candidate for the current query node is the current *cpos* (8). The incoming and outgoing dependency relations of the node at the current corpus position are stored in *corpusDeps* (9).

In line 11–18, all dependency relations in *queryDeps* are compared with all dependency relations in *corpusDeps*. If the two dependency relations are of the same type, the corpus position of the node connected via this

query	PostgreSQL		neo4j		CWB-treebank	
	time	sd	time	sd	time	sd
bachelor	0.41 s	0.20 s	4.98 s	2.81 s	5.95 s	3.11 s
confuse	4.77 s	0.27 s	19.95 s	8.20 s	7.41 s	3.69 s
creeps	195.36 s	5.19 s	934.33 s	100.51 s	11.20 s	1.83 s
way	927.63 s	14.87 s	343.39 s	276.90 s	31.79 s	4.36 s

Table 2: Average response times[10]

relation is added to the corpus-based list of candidates (*corpusCands*).

The corpus positions in *localCands* and *corpusCands* are then intersected (20–25). This way, corpus positions that match the node restrictions but are not connected to the current node are removed from the candidate list. The updated list of candidates is then used in the next recursion (26).

Once all query nodes have been processed, there remains only one candidate per query node. These candidates consitute one possible mapping of query nodes to corpus positions and are output before the subroutine returns (1–4).

```
["22",[["404","405","409","408"],
       ["404","405","414","413"]]]
```

Listing 2: Serialized output for a sentence with two mappings

The serialized output (listing 2) for a sentence consists of its sentence ID followed by a list of lists of corpus positions. Within these lists of corpus positions, the first corpus position corresponds to the first query node, the second position to the second node and so on. This data is sufficient for retrieving any kind of information that might have been requested, e. g. word forms, lemmata or the whole sentence.

3. Evaluation

In order to evaluate the performance of CWB-treebank, it was compared to earlier implementations using PostgreSQL as a relational database system and neo4j as a native graph database.[11] The hardware used was an AMD Opteron 875@2.2 GHz Linux machine with 16 GB of RAM and an U320 SCSI hard disk and should thus be relatively close to real world applications, although all systems will of course run faster on a modern CPU.[12] The corpus used for the evaluation was the 100 million word British National Corpus parsed with the Stanford Parser.

The data model in the relational database was highly normalized with one table per typed dependency, each containing columns with sentence ID, governor ID and dependent ID, plus a vocabulary lookup table. Relevant indexes were added to speed up the queries.

In the graph database, two data models were evaluated in an earlier stage of development and it turned out that a model in which word form, lemma, and pos were not coded as properties of every occurrence but as separate nodes with named edges to every occurrence was more suitable for the graph matching algorithm provided by neo4j.

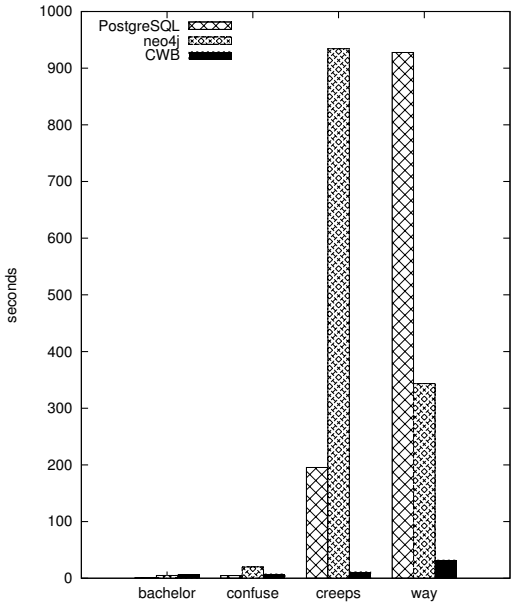


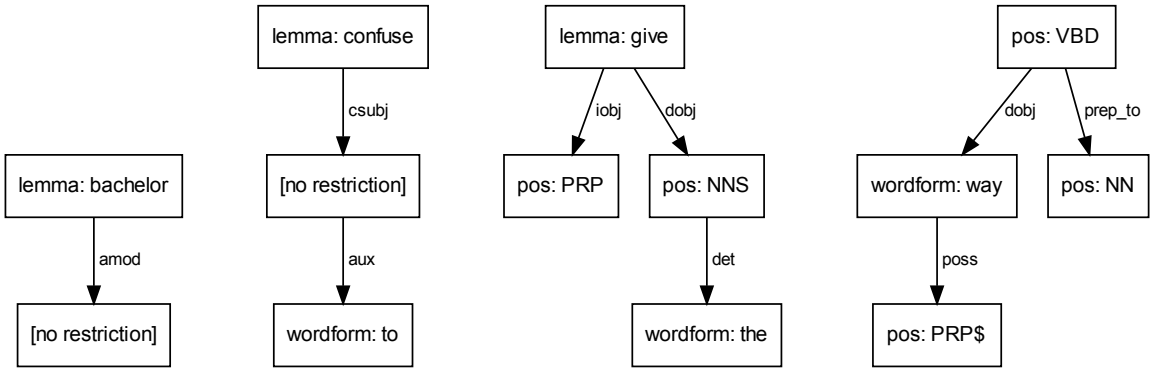
Figure 3: Average response times

As we can see in table 2 (visualized in figure 3), PostgreSQL is significantly faster than both CWB-

[10] In order to account for effects of memory mapping and file system caching (and thus obtain measurements closer to real world use cases), the four queries were executed in all possible orders and server was shut down and the caches were cleared after each round only.

[11] All measurements were taken with cold application caches since a cached query returns almost instantaneously in any of the systems so that differences are not noticeable to users of a web frontend as long as traffic on the site is not extremely high.

[12] Our first tests were carried out on an Intel Core2 Quad Q9550 with 4 GB of RAM but both PostgreSQL and neo4j ran out of memory for the more complex queries. It turns out that neo4j profits more from even larger RAM than the other two solutions. Thus storing the entire database in the RAM disk of a machine with 128 GB of RAM can improve its response time for complex queries by almost two orders of magnitude whereas the performance gain was by far not as drastic for the other two systems. However, RAM disks do not currently represent affordable storage for larger databases such as the ones required for large scale corpora.



(a) bachelor (b) confuse (c) creeps (d) way

Figure 4: Query graphs

treebank[13] and neo4j[14] for simple queries where only one join on word ID fields is required, such as a query for premodifying adjectives of *bachelor* (figure 4a). It should be noted that even for such simple queries, neo4j is not significantly faster than CWB-treebank.[15] Where three nodes are involved in the query, such as for *to*-infinitive subjects of *confuse* (figure 4b), PostgreSQL is faster than CWB-treebank but not significantly,[16] whereas both are significantly faster than neo4j.[17] In both queries, even the "slower" systems respond within acceptable timespans for a web interface. For larger query graphs which include higher frequency items, such as the query for sentences with ditransitive *give*, a personal pronoun as indirect object and a plural noun with a determiner *the* as direct object (as discussed in 2.2.; figure 4c), CWB treebank outperforms the other two solutions significantly.[18] The same is true of the query which looks for instances of *VERBed one's way PP* (figure 4d).[19] Thus we can say that the performance of CWB-treebank is much more consistent and that the system can handle larger query graphs more easily than the earlier implementations. The standard deviation is still within acceptable limits for CWB-treebank and PostgreSQL, but neo4j shows huge variation in its response times since these are highly dependent on the queries that had run before and are thus almost unpredictable in a production environment.[20]

A comparison of the size of the data on disk also reflects favourably on CWB-treebank, since it only consumes around 1.8 GB due to the efficient indexing and compression of the IMS Open Corpus Workbench while the PostgreSQL database takes up 18 GB (including indexes) and the graph database 24 GB.

4. Conclusion

As we have seen above, CWB-treebank, while not the fastet solution for trivial queries, significantly outperforms the alternative solutions for complex queries, allowing for reasonable response times. Its support for regular expressions and negation are additional features that make it the system of choice over the other solutions tested. While both other implementations have to treat the corpus as one big unit due to the data models enforced by the software used,[21] the main advantages of our system are that graphs are stored as individual sentences and that the powerful indexing mechanism and query engine provided by the IMS Open Corpus Workbench are adapted to linguistic data and thus enable CWB-treebank to efficiently reduce the search space through restrictions applied on individual nodes.

5. Acknowledgements

We would like to thank KONWIHR[22] for a small grant to work on our processing pipeline and the HPC group at the University of Erlangen-Nürnberg for their continuous support and for allowing us to use all sorts of extremely powerful hardware.
We are also grateful to the three anonymous reviewers for their constructive comments.

6. References

The British National Corpus. 2007. Version 3 (BNC XML edition). Distributed by Oxford University Computing Services on behalf of the BNC Consortium. `http://www.natcorp.ox.ac.uk`.

[13] Exact Wilcoxon Mann-Whitney Rank Sum Test: $Z = -5.9399, p_{\text{one-sided}} = 3.101e^{-14}$

[14] $Z = -5.9397, p_{\text{one-sided}} = 3.101e^{-14}$

[15] $Z = -1.3403, p_{\text{one-sided}} = 0.09176$

[16] $Z = 0, p_{\text{one-sided}} = 0.5041$

[17] $Z = -5.0518, p_{\text{one-sided}} = 1.129e^{-08}$ for a comparison of CWB-treebank with neo4j.

[18] $Z = -5.9385, p_{\text{one-sided}} = 3.101e^{-14}$ for a comparison of CWB-treebank with PostgreSQL.

[19] $Z = -5.6085, p_{\text{one-sided}} = 2.837e^{-11}$ for a comparison of CWB-treebank with neo4j.

[20] This is possibly due to the memory mapping architecture employed by neo4j, given that queries with cold file system caches after a restart of the system took particularly long.

[21] Thus the graph matching algorithm of neo4j only supported one large connected graph and had to be supplied with a starting node. In PostgreSQL, a faster data model may be possible through denormalization, but such steps usually come at the cost of increasing the size of the tables.

[22] Kompetenznetzwerk für Wissenschaftliches Höchstleistungsrechnen in Bayern

Daniel Cer, Marie-Catherine de Marneffe, Dan Jurafsky, and Christopher D. Manning. 2010. Parsing to stanford dependencies: Trade-offs between speed and accuracy. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC 2010)*, pages 1628–1632, Valletta.

Oliver Christ and Bruno M. Schulze. 1996. Ein flexibles und modulares Anfragesystem für Textcorpora. In H. Feldweg and E. W. Hinrichs, editors, *Lexikon und Text: Wiederverwendbare Methoden und Ressourcen zur linguistischen Erschließung des Deutschen*, pages 121–133, Tübingen. Niemeyer.

Oliver Christ. 1994. A modular and flexible architecture for an integrated corpus query system. In *Proceedings of COMPLEX'94: 3rd Conference on Computational Lexicography and Text Research*, pages 23–32, Budapest.

Luigi P. Cordella, Pasquale Foggia, Carlos Sansone, and Mario Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition.

Douglas Crockford. 2006. RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON). Technical report, IETF. `http://tools.ietf.org/html/rfc4627`.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008a. Stanford typed dependencies manual. `http://nlp.stanford.edu/software/dependencies_manual.pdf`. Revised in 2011.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008b. The stanford typed dependencies representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, pages 1–8, Manchester.

Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC 2006)*, pages 449–454, Genoa.

Stefan Evert and The OCWB Development Team. 2010a. The IMS Open Corpus Workbench (CWB). Corpus encoding tutorial. `http://cwb.sourceforge.net/files/CWB_Encoding_Tutorial.pdf`.

Stefan Evert and The OCWB Development Team. 2010b. The IMS Open Corpus Workbench (CWB). CQP query language tutorial. `http://cwb.sourceforge.net/files/CQP_Tutorial.pdf`.

Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York.

Pavel Rychlý. 2007. Manatee/Bonito – a modular corpus manager. In *1st Workshop on Recent Advances in Slavonic Natural Language Processing*, pages 65–70, Brno. Masarykova univerzita.

Gerold Schneider, Kaarel Kaljurand, and Fabio Rinaldi. 2009. Detecting protein/protein interactions using a parser and linguistic resources. In *CICLing 2009, 10th International Conference on Intelligent Text Processing and Computational Linguistics*, Mexico City.

Ron Shamir and Dekel Tsur. 1999. Faster subtree isomorphism. *Journal of Algorithms*, 33:267–280.

Lucien Tesnière. 1966. *Éléments de Syntaxe Structurale*. Klincksieck, Paris, 2nd edition.

Peter Uhrig and Thomas Proisl. 2011a. A fast and user-friendly interface for large treebanks. Presented at *Corpus Linguistics 2011* in Birmingham on July 20.

Peter Uhrig and Thomas Proisl. 2011b. The treebank.info project. Presented at *ICAME 32* in Oslo on June 4.

Jeffrey D. Ullman. 1976. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42.

Amir Zeldes, Julia Ritz, Anke Lüdeling, and Christian Chiarcos. 2009. ANNIS: A search tool for multilayer annotated corpora. In *Proceedings of Corpus Linguistics 2009*, pages 1–8, Liverpool.