J. Yaghob (Ed.): ITAT 2015 pp. 100–105
Charles University in Prague, Prague, 2015



# Implementation of a search engine for DeriNet

Jonáš Vidra

Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics, Charles University in Prague
vidra.jonas@seznam.cz,
WWW home page: http://jonys.cz/

*Abstract:* DeriNet, a new database of lexical derivates, is being developed at the Institute of Formal and Applied Linguistics. Since it is a wordnet containing large amounts of trees, it is hard to visualize, search and extend without specialized tools. This paper describes a program which has been developed to display the trees and enable finding certain types of errors, and its query language. The application is web-based for ease of use and access. The language has been inspired by CQL, the language used for searching the Czech National Corpus, and uses similar keywords and syntax extended to facilitate searching in tree structures.

## 1 Introduction

### 1.1 DeriNet

DeriNet[1] is a database of lexical derivates.[5] It contains lexemes structured into oriented trees that express derivational relations – the nodes are lexemes and edges are child→parent relations. Each lexeme represents one lemma with its meaning (word sense) partially disambiguated – several lexemes may represent the same lemma in different senses, if they differ in their derivational history – that is, they're derived from different words[2] or they have different descendants[3].

The database is stored in a tab-separated-values file using a simple format; see table 1 for an excerpt. Each line contains one lexeme. The meanings of the columns are: ID is a nonnegative integer which serves as a unique identifier; lemma is the base form of a word; techlemma is the lemma plus additional annotation in the style of the m-layer of the Prague Dependency Treebank[2]. POS is part of speech – A for adjectives, D for adverbs, N for nouns, V for verbs. Parent is the ID of the direct derivational parent. Every lexeme has at most one parent, which simplifies the structure, although it doesn't allow representing compounding. Version 0.9, released in December 2014, contains 305,781 lexemes sampled from the Czech National Corpus and 117,327 oriented edges.

[1] http://ufal.mff.cuni.cz/derinet

[2] such as "sladit" (to sweeten / to harmonize) from "sladký" (sweet) or "ladit" (to tune)

[3] as in the case of "hnát" (vulgar form of limb / to drive) whose child is either "hnátek" (diminutive from hnát) or "hnaný" (the chased one)

Table 1: Excerpt from the DeriNet database showing four lexemes.

| ID | lemma | techlemma | POS | parent |
|---|---|---|---|---|
| 177751 | nosní | nosní | A | 177725 |
| 177752 | nosník | nosník | N | 177753 |
| 177753 | nosný | nosný | A | |
| 177754 | nosně | nosně_^(*1ý) | D | 177751 |

### 1.2 Requirements

Since DeriNet is a graph database contained in a TSV file, it's very hard to visualize the structure without a specialized tool. Therefore, we've decided to create a specialized application which would use a simple language for searching. Ideally, this application should be accessible and usable by both skilled linguists and random visitors. The requirements were to support:

- Searching for lemmas, techlemmas and parts-of-speech using regular expressions.
- Querying of ancestor-descendant and sibling-sibling relations.
- Displaying the trees. The largest trees in DeriNet 0.9 contain 31 lexemes, the deepest are 7 levels deep, but most of them are smaller. This means that the visualization can be straightforward.

## 2 Related work

We've decided to survey the available query languages and either choose a suitable one or create a custom domain-specific language while drawing inspiration from existing sources. Our criteria were

1. support for searching within tree structures: specifying criteria for ancestors, descendants and siblings,
2. familiarity within our target group of Czech linguists – nonprogrammers; or at least a shallow learning curve,
3. easy and effective writing of new queries, and
4. simple implementation.

The first two points ruled out relational database query languages such as SQL, since they have no provisions for searching in trees. We've examined XPath, PML-TQ and CQL in more detail.

XPath[1] is a long-established language designed specifically for tree querying and it has an existing implementation in Javascript[6]. DeriNet has an XML version which could be used as a backing database with just a few changes. Unfortunately, testing has shown that existing web browsers cannot handle XML files the size of DeriNet. Loading the database easily took several minutes on weaker machines and any XPath query took tens of seconds to process. The latter could be overcome by using an external XPath library such as Wicked Good XPath, but the problem with representing the XML in a browser remains.

PML-TQ (Prague Markup Language – Tree Query[4]) is a language used for querying the Prague Dependency Treebank. It has many options, but it is also very complex and hard to implement. Therefore, it could only be reasonably used with the existing implementation as a server-side backend. Writing queries by hand is not very convenient, they are mostly edited using an offline application which we wouldn't be able to replicate in a web application. There is a web interface[4], but it doesn't offer the interactive editor. Also, many of its features would be unused in our simple system.

CQL (Corpus Query Language[3]) is a language used for searching the Czech National Corpus[5]. It only supports searching in sentences, i.e. linearly ordered sequences of nodes, but the syntax is simple and easy to modify and expand. It could be easily extended to handle basic tree queries as well. Also, its syntax supports using bare keywords as a query, while XPath and PML-TQ require a richer construction even for the simplest cases.

## 3 Query language

We've decided to go for maximum simplicity and familiarity and base our language on CQL. We wrote our own implementation using Jacob[6], an alternative of Flex and Bison for JavaScript, which creates an LALR parser from an EBNF grammar.

The simplest query is the attribute expression, which describes attributes of a single node in the database. Its results are all nodes that match the attributes. It has the following general form:

```
[attribute="value" …]
```

Multiple attributes can be specified at once – the query then matches iff all of them match. We currently support the following comparison operators: "=" for RegExp comparison, "==" for string comparison and "!=" and "!==" as their negations. The following attributes may be used: "lemma", "techlemma", "pos", "id" and "parent". There is a special syntax for matching nodes without a parent: `parent=="-1"` matches all nodes *without* a parent. This special case is guaranteed not to clash with any actual IDs, since normal IDs are nonnegative integers.

Examples:

```
[lemma="stroj"]
```

Matches all nodes whose lemma contains "stroj".

```
[lemma="^stroj$"]
```

Matches all nodes whose lemma is exactly "stroj".

```
[lemma=="stroj"]
```

The same as the previous one.

```
[]
```

Matches any node.

```
[id=="12345"]
```

Matches nodes whose ID is exactly "12345".

```
[pos=="A" lemma="ký$"]
```

Matches adjectives ending in "ký".

```
[parent=="-1"]
```

Matches all nodes without a parent.

Attribute expressions also have shorthand forms: By selecting the appropriate "default attribute", you can write just `"stroj"` instead of `[defaultattribute="stroj"]` and `stroj` instead of `[defaultattribute=="stroj"]`.

Attribute expressions can be combined to form a tree expression. The standard CQL was designed for querying sentences, which contain totally ordered nodes and so the only higher-order expression it supports is a linear sequence. However, nodes in our database are only partially ordered, so we had to define an extension which would allow querying of non-linearly ordered sets of nodes. To do this, we've introduced parentheses. The sequence of attribute expressions encodes parent→child relation, with multiple children enclosed in a parenthesis and separated by commas: `car carský (carsky, carismus, carství)`

## 4 Implementation

The application is a web-based program written in JavaScript and runs entirely client-side; the server is only used for storing the databases and the program itself. After the page loads, a background task downloads and parses the selected database. You can change the database used for searching by selecting a new one from a drop-down menu.

After you enter a search query into the text field, your query is processed and the results are displayed as SVG trees. Possible larger amounts of results are split into several pages. You can view the techlemma and part-of-speech of a single node by hovering

[4] https://lindat.mff.cuni.cz/services/pmltq/

[5] https://kontext.korpus.cz/first_form?queryselector=cqlrow

[6] https://github.com/Canna71/Jacob

over it with the pointer, or, alternatively, you can enable the detailed display for all nodes by selecting the „Show additional information“ checkbox. The results can be exported to the same tab-separated-values format that DeriNet uses for storage and stored to a file.

A screenshot of the application is included in figure 1 and a running version of the search engine is available at `http://jonys.cz/derinet/search/`.

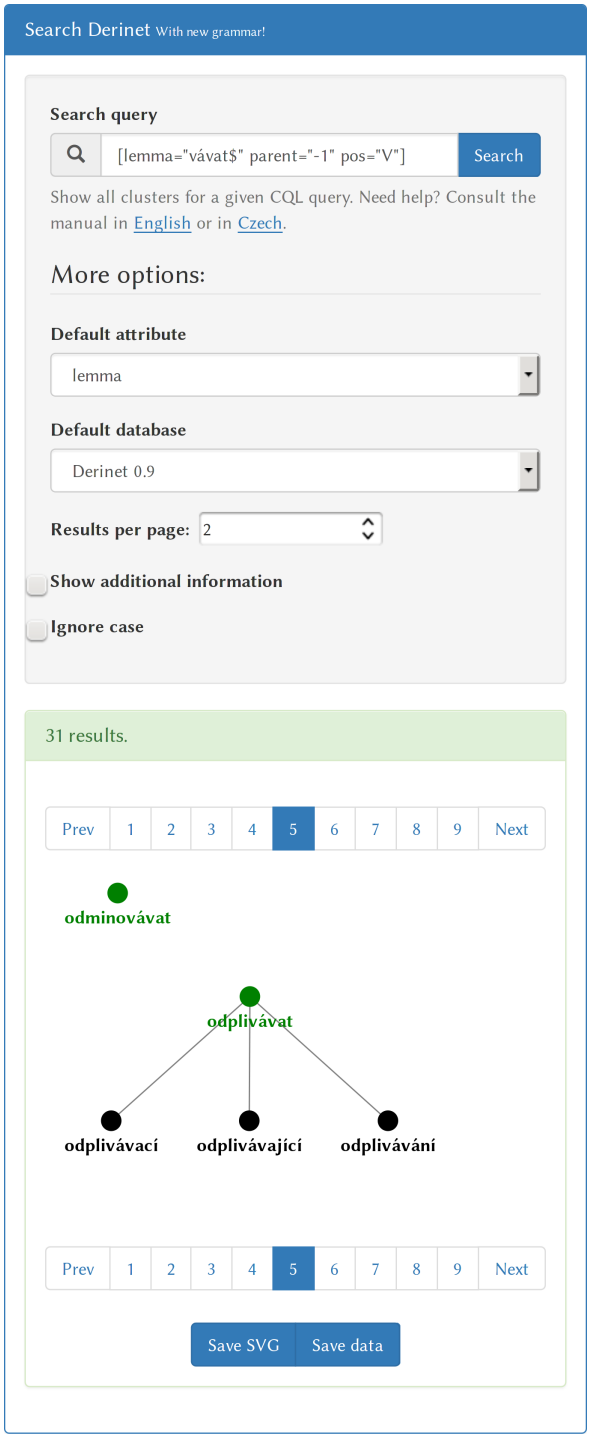


Figure 1: Screenshot of the application.

## 5 Detecting errors

The main reason for creating this application was enabling a quick way of searching for certain common errors that were present in the then-current version of DeriNet. DeriNet is built from the source lexicon using a series of scripts based on handwritten rules. Many of the rules have the form of pattern+exceptions, for example the pattern “A-ačný N-ačnost” which connects (among others) the noun “lačnost” to the adjective “lačný”, or “jednoznačnost” to “jednoznačný”, has an exception of “senzačnost”, which is derived from “senzační” instead. These rules often have overlooked exceptions.

In addition, there are several systematic errors in 0.9 that have been discovered by this search tool. For example:

- `[techlemma="_\^.*\(.*\*[0-9].*\)" parent="-1"]`
  Many techlemmas contain derivational information, which is not parsed correctly in version 0.9.
- `"á$" "ý$"`
  Matches female and male variants of names that have the derivation order reversed. The female variant ending in “á” should be derived from the male one.
- `[lemma="(ovat|it)$" pos=="A"]`
  These words are not adjectives, but errors in the source lexicon. These endings are both typical for verbs.
- `[lemma!="(í|ý|ost)$"] "ost$"`
  Bad derivations. Words ending in “ost” that aren’t derived by prefixation should be nearly always derived from adjectives ending in “í” or “ý”.

## 6 Problems

### 6.1 Performance

An important issue is performance. A loaded webpage with DeriNet Search can easily consume over 100 MiB of RAM and complex queries (especially queries with many siblings with few constraints on a single level, such as `[] ([],[],[],[],[])`) may take over a minute to complete. However, although the searching isn’t instantaneous, for simpler queries it is reasonably fast. Even a dated laptop (CPU: Intel Atom Z520, 1.33 GHz) is able to complete simple searches in under a second.

The design of the application is well suited to parallelization, but JavaScript has only limited provisions for parallel execution. Its threading model is based on message-passing with no shared data structures,

so speedup can only be achieved at the expense of using more memory, because each thread either needs its own copy of the whole database, or we have to split the DB into clusters (connected components) and allocate these clusters to search threads. The latter design would highly complicate displaying the results. This means that if we want to avoid complexity and needless copying of data, we are limited to single threaded execution only.

However, if there are multiple search windows open, each one runs in its own thread. It would be possible to create a global execution thread or a pool with user-determined amount of threads, delegate all work to it and keep only a single database copy in memory, but not all browsers support these techniques and we've decided not to complicate the design for small gains.

### 6.2 Compatibility

Apart from large memory requirements, which may be a problem on mobile devices, the application requires support for ECMAScript 5.1, HTML5, CSS3 and SVG. All modern desktop browsers (less than three years old) should pass these requirements, with the exception of Internet Explorer before version 11, but support on browsers for mobile devices is poor. Tested browsers include Firefox 31, Konqueror 4.14 and Chrome 36. The application is only partially usable in Opera 12.16, which doesn't support the history manipulation method needed to switch between pages of results and so only the first page is ever shown.

### 6.3 Scalability

Another set of problems was uncovered when testing the application with the newest version of DeriNet, 0.10, which is as of 2015-06-14 not yet publicly available. Version 0.10 is four times larger than 0.9 (45 MiB uncompressed and 9.6 MiB when compressed with gzip, as opposed to 12 MiB uncompressed and 3 MiB compressed) and downloading a database of this size is problematic for users without a broadband network connection.

Search times have also increased, because DeriNet 0.10 contains 968,929 lexemes – over three times more than 0.9. To test the scalability of our application with increasing database size, we've prepared six progressively smaller datasets by cutting 500,000, 250,000, 100,000, 50,000 and 10,000 lexemes with the lowest IDs out of DeriNet 0.10. We've then measured the time needed to parse a query, search the database and draw the resulting SVG. Each measurement was repeated five times, the results averaged and the error expressed using standard deviation.

Three queries were selected for testing: `Absolón`, which produces a single result; `[]`, which matches all nodes; and `[pos!=="A"] ([pos="[ADNV]$"], [] [], [])` as a test of more complex queries. Figure 2 shows that the time needed for parsing and display is approximately constant and that the searching time quickly dominates. Figure 3 indicates that the query `[pos!=="A"] ([pos="[ADNV]$"], [] [], [])` scales supralinearly, but figure 4 shows that for the other queries the algorithm is, in fact, linear. The explanation for why the complex query behaves differently lies in the structure of the database, as evidenced by the number of results. The database is scanned from the beginning to the end without using any acceleration technique such as an index, and the beginning (low IDs) contains lemmas that start with capital letters. These are typically names which contain fewer derivational links than the rest of the database. This causes the complex query to abort quickly, just after discovering that the node doesn't have enough children to satisfy the requirements.

## 7 Future work

Due to the problems encountered with DeriNet 0.10, we plan on switching to server-side searching. This is feasible with few changes thanks to Node.js, which allows running arbitrary JavaScript code on the server. Doing so would allow us to support even primitive browsers on devices with little processing power – the demands for modern JavaScript features could be dropped; support for SVG would be sufficient.

Another needed change is optimization of the search routines. Queries such as `[] ([], [], [], [], [])` (find all trees with at least five siblings on a single level) take a very large amount of time, sometimes even minutes. The current algorithm goes through all the possible combinations, which is not needed, since many of the results will be duplicates of each other. Stopping at the first match is not a solution, though, because advanced features of CQL such as "within" and "containing" clauses rely on this behavior to work.

The third possible area of improvement is introducing new language features.

- A way of marking "no more nodes here". We have the `[parent=="-1"]` syntax for finding nodes that don't have a parent, but there should be a better way of specifying this, that would also allow searching for trees that are smaller than a certain limit, as opposed to larger than a limit. The latter can be done using `[] ([pos="A"], [], [])` (find all adjectives with more than two siblings).
- Support for iteration: "*", "+", "?", "{x,y}". The last type partially satisfies (and depends on) the previous point.
- Support for "within" and "containing".

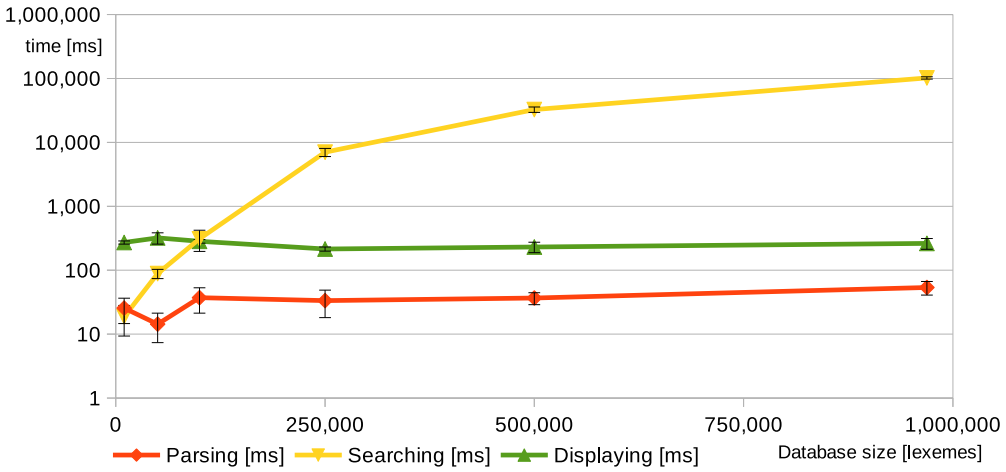


Figure 2: A graph showing the increase of time needed to process query `[pos!=="A"] ([pos="[ADNV]$"], [] [], [])` with increasing database size. The error bars represent one standard deviation over five trials.

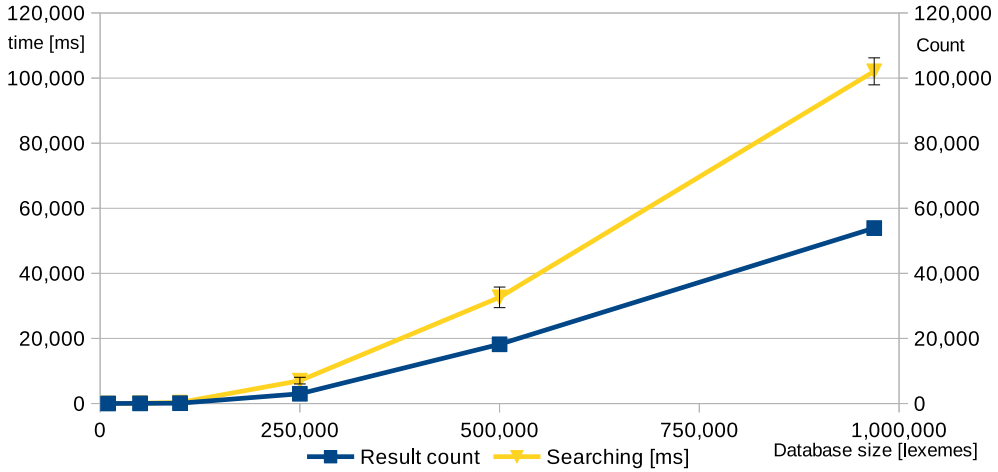


Figure 3: Graph showing that when processing `[pos!=="A"] ([pos="[ADNV]$"], [] [], [])`, the time needed increases supralinearly. The number of results has been included for comparison. The error bars represent one standard deviation over five trials.

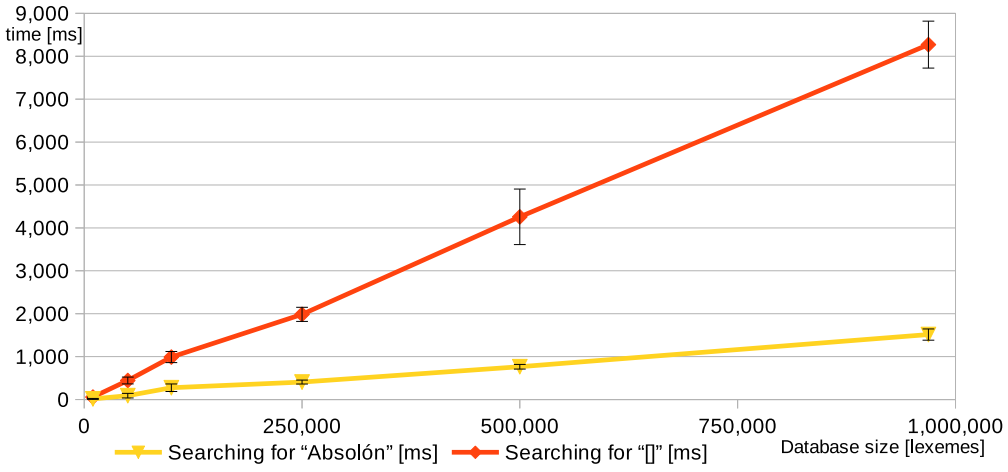


Figure 4: Graph showing that processing `Absolón` and `[]` scales linearly with increasing database size. The error bars represent one standard deviation over five trials.

The application itself could be improved, too. The following would be useful:

- User-friendlier handling of erroneous input. When you enter `[lemma!="ický$" [lemma="ičnost$"]`, instead of "Unexpected token lsbra "[" at (0:16)" the parser should probably report "Missing close-bracket before position 16" and show the query with the offending part highlighted in red. Unfortunately, Jacob, the tool we use for building the parser, doesn't have any support for returning structured error information and printing useful messages.
- A method of exporting the SVG to a file that produces a useful image. Currently, the SVG images are styled using CSS, which gives us a flexible way of coloring the nodes and showing/hiding technical information. This method doesn't work outside of browsers, though, which means that the exported SVG is black-and-white only and, unless the user had the "Show additional information" checkbox checked, contains overlapping text.
- A more compact style of drawing trees. Even in the current database there are some trees that are too wide to fit on screen without horizontal scrolling. Trees in DeriNet 0.10 are even larger, which makes this a big problem.

## 8 Conclusion

The stated goals were to build an application that would enable users to easily search DeriNet and help its creators discover both systematic and random errors. The utility fulfilled our expectations with the second part – error searching – since several mistakes and faults have been detected with its help. The query language is only very basic so far, but extensions are possible.

## Acknowledgements

This work has been using language resources developed and/or stored and/or distributed by the LINDAT/CLARIN project of the Ministry of Education of the Czech Republic (project LM2010013).

## References

[1] James Clark, Steve DeRose, et al. *XML path language (XPath) version 1.0*. 1999.

[2] Jan Hajič. *Disambiguation of rich inflection: computational morphology of Czech*. Karolinum, 2004.

[3] Miloš Jakubíček et al. "Fast Syntactic Searching in Very Large Corpora for Many Languages." In: *PACLIC*. Vol. 24. 2010, pp. 741–747.

[4] Petr Pajas and Jan Štěpánek. "System for querying syntactically annotated corpora". In: *Proceedings of the ACL-IJCNLP 2009 Software Demonstrations*. Association for Computational Linguistics. 2009, pp. 33–36.

[5] Magda Ševčíková and Zdeněk Žabokrtský. "Word-Formation Network for Czech". In: *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*. Ed. by Nicoletta Calzolari (Conference Chair) et al. Reykjavik, Iceland: European Language Resources Association (ELRA), May 26–31, 2014. ISBN: 978-2-9517408-8-4.

[6] Ray Whitmer et al. "Document Object Model (DOM) Level 3 XPath Specification". In: *W3C, http://www.w3.org/TR/DOM-Level-3-XPath* (2004).