DGrep: A Pattern-matching Tool for Dependency Trees

Eric Nichols
Honda Research Institute Japan Co.Ltd.
e.nichols@jp.honda-ri.com

Paul Reisert
Tohoku University, Japan
preisert@ecei.tohoku.ac.jp

1 Introduction

Dependency parsing has become a keystone to much natural language processing, however, there is still a lack of tools designed for pattern matching on dependency trees, leading to much duplication of effort by programmers who want to search or manipulate them. In this paper, we introduce DGrep, a TGrep-inspired query language and tool suite designed specifically for dependency trees. DGrep's key features include: query syntax that aims to be isomorphic to the target dependency tree; consistent query composition semantics that allows intuitive query chaining; default surface form matching without special syntax to accommodate the lexically-driven nature of dependency trees; and full query support over the chunks and tokens of Cabocha's Japanese dependency trees - including parts-of-speech and other features. In this paper, we describe DGrep's query language and outline plans for its future development. We plan to release DGrep to the research community in the near future.

2 Related Research

One of the earliest tools developed for querying treebanks is TGrep [13]. It provided a simple, grep-like command line interface for searching Penn Treebank-style bracketed phrase structure trees, and its query language supported specification of parent, child, and sibling relations and searching with regular expression. Many of the techniques that have been adopted by modern treebank query languages originated in TGrep: it provided a mechanism for marking nodes to print in search results and converted treebanks to a cacheable form to improve search efficiency. TGrep is now defunct, but many of its features were reimplemented in TGrep2 [15] and Tregex [8].

The simple CLI and robust feature set of TGrep and its successors provide the blueprint for DGrep's development: our goal is to adopt the elements of TGrep that made it successful and adapt them to searching dependency trees.

Another paradigm for searching phrase structure trees is the search engine. This approach dates back to TIGERSearch [5] The Linguist's Search Engine [14], and LPath [7], with recent instantiations in Fangorn [4]. Typically, these systems will index treebanks and allow users to search for patterns by manually entering queries or by constructing queries with a GUI-based editor.

MonaSearch [9] offers another approach to phrase structure tree search, with a query language implemented in Monadic Second Order logic that permits some queries that are not permitted in first-order-logic-based language, such search for arbitrarily nested prepositional phrase structures. By converting queries into tree automata, MonaSearch claims linear time search is possible.

Recently, several tools that support querying over multiple treebank formats have been developed. PML-TQ [16] converts phrase structure trees and dependency trees to a single, XML-based markup language. INESS-Search [11] extends TIGERSearch [5] to support general directed graphs for querying the various structures in LFG treebanks. GrETEL [1] supports treebank search without requiring queries to be entered in a domain-specific language. Instead, users search for examples using a tree browser GUI and edit the tree into a query.

Finally, a few tools that focus exclusively on searching dependency trees have been developed. ChaKi [10] is a GUI-driven corpus annotation tool for the Japanese language. It is the only tool we know of besides DGrep that supports full querying over Japanese morphological analysis results and syntactic dependencies, however, our goals are divergent: ChaKi aims to provide an easy to use annotation environment to non-programmers and does not include a CLI, while DGrep aims to provide a lightweight and efficient CLI for large-scale text processing. Semgrex [2] is a query language for semantic graphs produced by the Stanford CoreNLP parser. ICARUS [3] is a dependency treebank search tool with both a GUI and CLI. It supports the CoNLL dependency parsing shared task format, and aims to strike a balance between query language expressiveness and tree visualization.

3 The DGrep Query Language

Before introducing DGrep's query language, let us consider its desiderata. Our motivation in developing DGrep is to create a lightweight tool that can be used to preprocess large collections of text for tasks like knowledge acquisition and information retrieval. Existing query languages for phrase structure trees could be used to search dependency trees in principle, however, because dependency trees tend to gather more information into lexical nodes and represent syntactic relations through edges, applying existing query languages can be lead to verbose and unintuitive queries. We want an intuitive query language that resembles the trees it is querying over. Furthermore, we want the ability to print the dependency paths between nodes in a match in addition to subtrees. At the same time, because dependency trees are highly lexical by nature, we want to make

it easy to search over the surface forms of words. Finally, taking inspiration from TGrep, we want to develop a clean command line interface with minimal dependencies and installation process.

3.1 Querying over Nodes

DGrep currently supports searching Japanese dependency trees produced by the dependency parser, Cabocha[6]. Cabocha's dependency trees consist of node which are *chunks*, or sequences of tokens. Each token is set of feature-value pairs. These features are explained in detail in Section 3.4. For the examples in this section, we use the dependency tree below. See Table 1 for the example queries referenced throughout this section. The first chunk in the tree, 食堂で consists of the two tokens, 食堂 and で.

I ate delicious tsukemen at the dining hall.

DGrep defines the following operators over nodes:

Operator	Meaning
`A -> B`	`A` is a child of `B`
`A <- B`	`A` is a parent of `B`
`A ->> B`	`A` is a descendant of `B`
`A <<- B`	`A` is an ancestor of `B`
`A >: B`	`A` is only child of `B`
`A <: B`	`B` is only child of `A`
`A & B`	Both conditions `A` and `B` are met
`A | B`	Either condition `A` or `B` is met
`A - B`	A parent or child relation exists between `A` and `B`
`A -- B`	An ancestor or descendant relation exists between `A` and `B`
`!A`	All nodes that do not match condition `A`

The following special node symbols are also supported:

Symbol	Name	Meaning
`^`	Root	Matches the root node
`_`	Wildcard[1]	Matches any node

The wildcard is demonstrated in Table 1 Example 3.

3.2 Regular Expressions

DGrep also supports regular expression matching on chunk surface form and token feature values. DGrep supports POSIX-compatible Java regular expressions with the following syntax: `/<regex>/`. Table 1 Examples 1 and 8 demonstrate the use of regular expressions at the chunk and token levels. By default, a chunk-level query is evaluated as a regular expression search over the surface form of the entire chunk that is represented by the node. Thus, the two queries shown in Example 1 are equivalent.

3.3 Compound Queries

DGrep evaluates queries from left to right and handles compound queries by using the root nodes of intermediate query results to continue the query. Thus, the query `A -> B -> C` is read as `A` *depends on* `B` *and* `B` *depends on* `C`, is evaluated in the order of `((A -> B) -> C)`, and can be thought of as being decomposed into `(A -> B) & (B -> C)`. As shown in Table 1 Example 5, the query `(`美味しい `->` つけ麺を`)` `->` 食べ is equivalent to 美味しい `->` つけ麺を `->` 食べ.

Parentheses can be used to change the order of evaluation of a given query. Thus, the query `A -> (B -> C)` will be read as `A` *depends on* `C` *and* `B` *depends on* `C` and can be thought of as equivalent to `(A -> C) & (B -> C)`. Table 1 Example 6 gives an example of how parentheses can be used to specify multiple children with the same parent:

3.4 Querying over Tokens

DGrep supports full search over the token features in the chunks composing Cabocha's dependency trees. Features can be accessed using the following feature aliases:

Feature	Japanese	English
`pos`	品詞	*part-of-speech*
`pos1`	品詞細分類 1	*POS1*
`pos2`	品詞細分類 2	*POS2*
`pos3`	品詞細分類 3	*POS3*
`ctype`	活用形	*conjugation type*
`cform`	活用型	*conjugation form*
`baseform`	原形	*base form*
`orth`	読み	*orthography*
`pron`	発音	*pronunciation*

In addition to features, each token has a `surface` value which may be searched. For example, queries of the form `{surface:`麺`}` will result in searching at the token level for a surface of 麺. This differs from a query of 麺 only which is a query across the concatenation of all token surface forms in a chunk.

For more information on the above features, consult the Mecab Reference Manual[2] and the Ipadic User Manual[3].

Like chunk-level queries, token-level queries are also evaluated from left to right. Parentheses may be used for compound queries at the token level. DGrep offers the ability to search on multiple tokens per chunk in order to match specific conditions that could not be found with a simple query alone. In order to achieve this, we introduce the following syntax:

```
[{<feature>:<value>,...}
 {<feature>:<value>,...} ...]
```

[1] Using _ as a wildcard is a convention of the Scala programming language.

[2] http://mecab.googlecode.com/svn/trunk/mecab/doc/index.html

[3] http://sourceforge.jp/projects/ipadic/docs/ipadic-2.7.0-manual-en.pdf/en/1/

Examples of feature-level queries are given in Table 1 Examples 7 and 8.

3.5 Boolean Operators

Boolean operators `&` and `|` can be used on both chunk-level queries and token-level queries. `&` indicates that multiple conditions must hold for a match to be found, while `|` indicates one or more of a set of conditions is satisfactory. In the event that multiple matches are found for a query using `|`, each match is returned independently.

3.6 Output Modes

DGrep supports the following output modes:

- *dependency paths*: outputs the minimal dependency paths for a specified query
- *subtrees*: outputs subtrees starting from the root node of a resulting path from a given query

When a DGrep query finds multiple matches, each match is returned independently. Table 1 Examples 3, 4, and 7 show queries with multiple matches.

DGrep output can be displayed as plain text, XML, and LaTeX dependency graphs[4]. All dependency graphs in this paper were generated by DGrep. Matches can be printed in color in the context of the full input sentence, or the matching region alone can be printed.

3.7 Implementation

DGrep is implemented in Scala as a query library with Java bindings and a command line interface. Developing in Scala offered a number of benefits: we were able to rapidly prototype in a type-safe functional programming-friendly language which runs on the JVM, giving us access to a large number of Java libraries and reasonable performance. Scala's support for functional programming allowed us to implement the DGrep syntax in a composable manner using parser combinators, and it opens the door for exploring query optimization techniques like memoization.

4 Conclusion

In this paper, we presented DGrep, a pattern-matching tool for searching dependency trees. DGrep was designed with the goals of creating lightweight, composable language that is intuitive for dependency trees to support efficient querying of Web-scale text. DGrep supports full search over cabocha-formatted Japanese dependency trees – including at the chunk- and token-level – and output of both dependency paths and subtrees in plain text, XML, and LaTex dependency graphs. We plan to release DGrep to the research community in the near future.

In future work, we plan to expand support for other parsers, formats, and languages, and to conduct detailed performance evaluation and explore areas of potential optimization. Implementing support for the CoNLL dependency shared task format would enable comparison between DGrep and existing query tools and facilitate robust profiling. Query rewriting to avoid inefficient formulations and memoization to avoid redundant querying are potential avenues of optimization. Because Web corpora are often too large to index in advance, lazy methods of indexing on demand as queries are made may also be beneficial.

[4] LaTeX dependency graphs use the TikZ-dependency [12] package.

Acknowledgments

This research was supported by Honda Research Institute Japan Co.Ltd. The authors would also like to thank Professor Kentaro Inui (Tohoku University) for his invaluable feedback.

References

[1] Liesbeth Augustinus, Vincent Vandeghinste, and Frank Van Eynde. Example-based treebank querying. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation*, 2012.

[2] Nathanael Chambers, Daniel Cer, Trond Grenager, David Hall, Chloe Kiddon, Bill MacCartney, Marie-Catherine de Marneffe, Daniel Ramage, Eric Yeh, and Christopher D Manning. Learning alignments and leveraging natural logic. In *RTE '07: Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing*, 2007.

[3] Markus Gärtner, Gregor Thiele, Wolfgang Seeker, Anders Björkelund, and Jonas Kuhn. Icarus – an extensible graphical search tool for dependency treebanks. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Sofia, Bulgaria, August 2013.

[4] Sumukh Ghodke and Steven Bird. Fangorn: A system for querying very large treebanks. In *Proceedings of COLING 2012: Demonstration Papers*, pages 175–182, Mumbai, India, 2012.

[5] Esther König, Wolfgang Lezius, and Holger Voormann. TIGERSearch 2.1 user's manual. 2003. http://www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/TIGERSearch/doc/pdf/manual.pdf.

[6] Taku Kudo and Yuji Matsumoto. Japanese dependency analysis using cascaded chunking. In *CoNLL 2002: Proceedings of the 6th Conference on Natural Language Learning 2002 (COLING 2002 Post-Conference Workshops)*, pages 63–69, 2002.

[7] Catherine Lai and Steven Bird. LPath+: A first-order complete language for linguistic tree query. In *Proceedings of the 19th Pacific Asia Conference on Language, Information and Computation*, 2005.

[8] Roger Levy and Galen Andrew. Tregex and Tsurgeon: tools for querying and manipulating tree data structures. In *Proceedings of the fifth international conference on Language Resources and Evaluation*, pages 2231–2234, 2006.

[9] Hendrik Maryns and Stephan Kepser. MonaSearch – querying linguistic treebanks with monadic second order logic. In *The 7th International Workshop on Treebanks and Linguistic Theories (TLT 7)*, Groningen, Holland, January 2009.

[10] Yuji Matsumoto, Masayuki Asahara, Kiyota Hashimoto, Yukio Tono, Akira Ohtani, and Toshio Morita. An annotated corpus management tool: ChaKi. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation*, Genoa, Italy, May 2006.

[11] Paul Meurer. INESS-Search: A search system for LFG (and other) treebanks. In *Proceedings of the LFG12 Conference*, 2012.

[12] Daniele Pighin. The TikZ-dependency package. User manual, version 1.1. 2012. http://ctan.math.utah.edu/ctan/tex-archive/graphics/pgf/contrib/tikz-dependency/tikz-dependency-doc.pdf.

[13] Richard Pito. TGREPDOC. Manual page for TGrep. *Linguistics Data Consortium, University of Pennsylania*, 1994. http://www.stanford.edu/~bresnan/128_2003-4/Docs/tgrepdoc.1.pdf.

[14] Philip Resnik and Aaron Elkiss. The Linguist's Search Engine: An overview. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*, pages 33–36, Ann Arbor, Michigan, June 2005.

[15] Douglas L T Rohde. TGrep2 user manual: Version 1.15. 2005. http://tedlab.mit.edu/dr/Tgrep2.pdf.

[16] Jan Štěpánek and Petr Pajas. Querying diverse treebanks in a uniform way. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation*, Valletta, Malta, may 2010. European Language Resources Association (ELRA).

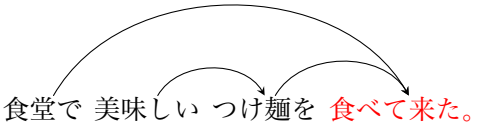
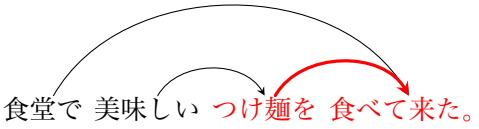
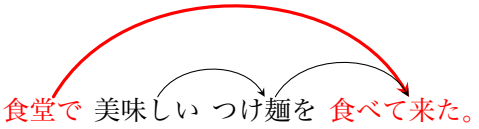
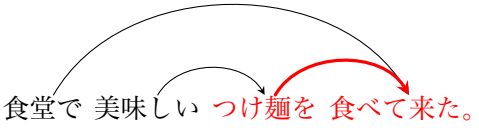
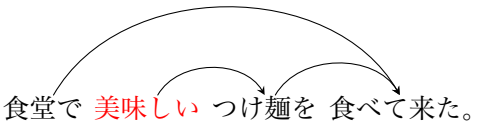
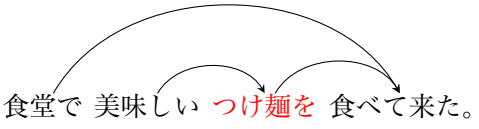
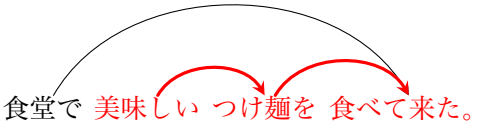
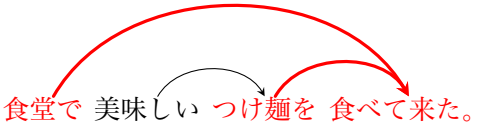
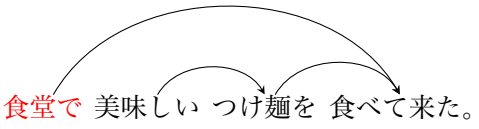
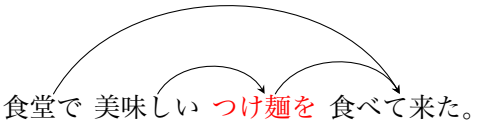
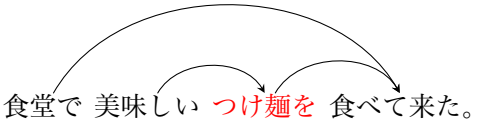
	Query	Meaning	Matches
1	食べ /食べ/	Matches all chunks with a surface form containing 食べ	
2	つけ麺を -> 食べ	Matches all paths in which つけ麺を depends on 食べ	
3	_ -> 食べ	Matches all paths between 食べ and its children	 
4	!食	Matches any chunk whose surface does not contain 食	 
5	(美味しい -> つけ麺を) -> 食べ	Matches all paths where 美味しい depends on つけ麺を and つけ麺を depends on 食べ	
6	食堂で -> (つけ麺を -> 食べ)	Matches all paths where 食堂で and つけ麺を both つけ麺を depend on 食べ	
7	{pos:名詞}	Matches all chunks containing a token with part-of-speech 名詞	 
8	{orth:/ん$/}	Matches all chunks which contain a token whose orthography ends with ん	

Table 1: Example queries for the sentence 食堂で美味しいつけ麺を食べて来た。*I ate delicious tsukemen at the dining hall.*