A Grammar and Dependency Aware Search System for Japanese Sentences

Arseny Tolmachev Hajime Morita Sadao Kurohashi
Kyoto University, Graduate School of Informatics
{arseny, morita}@nlp.ist.i.kyoto-u.ac.jp, kuro@i.kyoto-u.ac.jp

Abstract

Example sentences are useful both for language learners and linguists. Often users have a need to find the usage of a grammatical structure while having parts of speech tags instead of some words in the query. We have implemented a high-performance search engine that is able to process grammatical dependencies and parts of speech information in queries from a large scale corpus.

1 Introduction

General search systems like Google or Microsoft Bing are designed for searching documents relevant to a specific query. Such documents are usually long pieces of text, for example web pages. Language learners and teachers use such systems for acquiring example usages or contexts for words they learn. However, general search systems are not well-suited for this task. Firstly, users doing such search are not looking for documents, they are looking for sentences. Additionally, conventional search engines ignore grammatical information when indexing text, although this data is extremely useful for finding example usages of words.

Searching sentences for educational and linguistic usage often requires more features than just querying on terms as general systems do. Sentence level search should support queries not only on lexical level, but on lexical dependencies, part of speech (POS) tags, conjugation forms and **grammatic words** like case markers (が, を) or auxiliary verbs (いる after テ form) as well. Usually, users want to find usages of a word in some context.

Understanding onomatopoeia is difficult for many Japanese language learners. Example sentences like 肌がピリピリする are mostly for learners, because they give no idea about what ピリピリ is. Having sentences like 肌がピリピリ痛く感じる are substantially better in this case. It can be said that a system should give users an ability to choose *"a form"* of context for the word. Additionally, a system should be able to work with huge amount of data, because context patterns are usually sparse. At the same time, to be useful the system should give replies quickly.

We have implemented a distributed high-performance sentence level search system for Japanese language that is able to process queries with not only lexical information, but grammatic words and lexical dependency information as well. The system achieves less than 300 ms query times for 90% of queries when 700M sentences are indexed.

The proposed system architecture consists of two main parts: compressed database and search core.

Search core is based on Apache Lucene[1] with additional components for dependency and POS tag support for indexing and querying. Furthermore, in order to support fast queries on huge corpora, the system is implemented in a distributed master-slave like manner. Distribution was done using actor programming model and Akka library[2].

2 Related work

Sentence search tools are related the most to corpus management and exploration tools. However, there are not many tools which use structural information. Jakubíek et al. implements a syntactic corpus search system [1]. This system focused on searching using constituency instead of dependency syntactic structures. Also the system was not a search engine and query times on sentences structure were in orders of tens of minutes which renders working with huge corpora impossible.

For Japanese language, dependencies are used in search as well [2, 3]. TSUBAKI search system [2] uses dependency trees for indexing and querying and it is distributed. However it is a document-level search system and does not allow doing queries using POS information. A system proposed by Takeuchi and Tsujii [3] uses dependency information, however does not allow to use grammatical and dependency information. It has more focus on handling paraphrases. Recent versions of Chaki (茶器) corpus management

[1] https://lucene.apache.org/core/
[2] http://akka.io/

tool[3] support queries using dependencies, lexical and part of speech information, however it is not a search engine – it was not designed for a scale of several hundred millions sentences.

3 Query language & examples

Japanese language has no natural word boundaries symbols and the definition of a morpheme varies from one analyzer to another. In this sense, writing queries using sub-bunsetsu units is going to be difficult and error-prone, therefore another approach was taken. Search system query language is designed by adding special symbols to plain Japanese. The list of special symbols is [-, →*~]. Minus (-) has its usual meaning as in typical search system: absence of a term in search results.

A query is divided into parts by commas. Each part is completely independent of others. Usually, the whole query is a disjunction of its parts. General search systems like Google use whitespace instead of commas.

Dependency between two terms is specified by an arrow symbol: "A → B", meaning B as a parent of A. Arrows can be chained "A → B → C", however in this case both A and B would be treated as sibling children of C. Sentence head can be specified as "A → EOS", where EOS is literal.

A star after a word means that you want to ignore its conjugation information. For example, "食べる*" would match any conjugation form of 食べる and even 食べない, however query "食べる" would match *only current (dictionary) form* of the term. Only the last conjugation is going to be ignored: "食べたい*" will match 食べたくない, but won't match 食べない. Ignoring conjugations inside words is not supported yet. For non-conjugatable POS this symbol is a no-op.

A tilde (~) before a content word means to replace it with its own POS tag. This means "~食べて" is going to match a テ *form of any verb*. For grammatic words this replacement is not supported presently.

Here are some example queries[4] the system supports. A query "~物が→ぴりぴり→~動く*,-する*" finds usages of onomatopoeia ぴりぴり in context, but usages with する are difficult to understand and are therefore ignored. Search results include sentences like 寒さより肌がピリピリと痛く感じます。頬がピリピリ凍る、京都の冬です。全身の神経がピリピリ緊張する。which are good examples for the ピリピリ.

In the query "~書いた→ため,-ため→ EOS" the objective is to find out usages of ため after past form of verbs, so that it is not a head of the sentence. Search results are sentences like 陰謀を暴いたために脅迫された.

For a query "いい加減→~やらない", the objective was to find unusual imperative-like usages of verbs with いい加減 as a modifier. Results contained sentences like その妄想いい加減やめない？ or 良い子はいい加減止めない？ which are exactly the usages we seek.

These types of queries are impossible to search with a regular search engine.

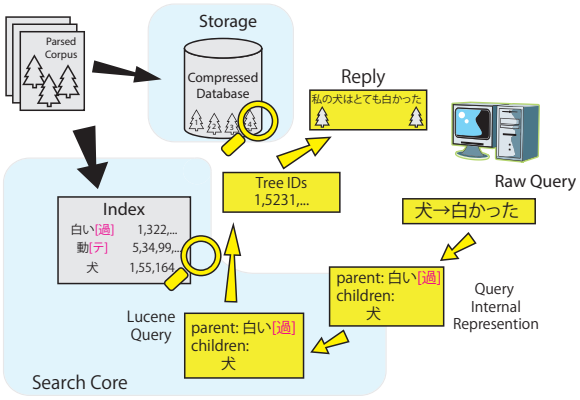

Figure 1: System structure and querying workflow

4 System structure

Proposed system consists of two main architectural parts, as shown on Figure 1: a compressed tree database and a search core. The tree database stores dependency parse trees in compressed form. They are used to build replies to search queries. The search core is built on Apache Lucene with custom tokenizer and querying. Instead of content word morphemes, tokens are created from parse trees in a way to make it possible to issue queries on conjugation forms of POS with grammatic words attached.

Current implementation uses KNP[5] dependency parser trees as input. KNP groups morphemes to bunsetsu units which are useful units for human interpretation. The search core uses KNP bunsetsu as a core unit for further processing.

4.1 Compressed Database

The system operates on dependency trees as input. Those trees should be extractable to create replies, and because output of dependency parsers is rather verbose, the data should be stored on disk in a compressed form. However, usual compression tools do not support random access to the compressed files. Building a specialized compressed database overcomes this limitation.

The compressed database consists of an index stored as a B-tree using MapDB[6] and data files. The

[3] https://osdn.jp/projects/chaki/
[4] They are hyperlinks and clickable
[5] http://nlp.ist.i.kyoto-u.ac.jp/EN/?KNP
[6] http://www.mapdb.org/

database index is a mapping from a tree id to a tree compressed pointer and tree size pair. Data files consist of 64 kilobyte blocks. Each block is archived independently of others and saved to disk. There are no inter-block dependencies, meaning that blocks can be read in the arbitrary order. To extract a tree from a data file, the system needs to read a block from disk, decompress it, and get a tree from that block. The information about the block and the position inside the block can be stored in a single compressed pointer.

Compressed pointer is a trick taken from the bioinformatics BAM/BGZF[7] storage format used for storing DNA sequences in compressed files. Compressed pointer consists of two parts: beginning of a compressed block in a data file and offset of needed data inside the decompressed block. Block sizes are fixed to 64k and the 64-bit pointer can be formed by making lower 16 bits to store the uncompressed offset and the remaining 48 bits to store block start address in the compressed file.

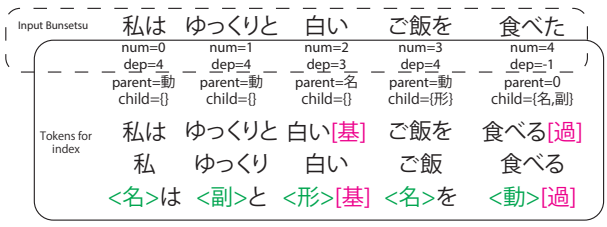

Figure 2: Bunsetsu to token conversion for indexing sentence "私はゆっくりと白いご飯を食べた". Tokens contain lexical information (black), POS tags (green) and conjugation forms (magenta). Dependency information is common for a set of tokens spawned from a single bunsetsu. This information consists of bunsetsu number, dependency number, POS of parent and set of children POS.

4.2 Search: Indexing

Speed of search engines comes from a special structure – **inverse index** – that is created from original documents. Index is created from tokens which are produced by analyzing input. For the proposed system tokens are created from bunsetsu of input trees. Each bunsetsu is numbered and has a **dependency number** (number of a parent). Both of these numbers with POS information of bunsetsu children and parent are transferred to tokens. Tokens are created for each occurrence of the bunsetsu in the tree. When stored to index they are inverted and become **token postings** – information about documents which contain a certain token.

Search systems work by fully matching query tokens with indexed ones and processing posting information linked to the tokens. The main objective behind the token design was to combine lexical and grammatic information in a single place, meaning that it could be both stored in index and at the same time easily constructed from a search query. Adding dependency information to these tokens enables an implementation of a system to meet all the requirements stated in the introduction.

Tokens are generated from a bunsetsu in two steps. The first step generates a seed token from the bunsetsu. Token content is a concatenation of bunsetsu morphemes. Morphemes with conjugatable POS are represented by a lemma form with the conjugation tag. For example, the verb 帰った would be represented as "帰る [過]". Non-conjugatable POS morphemes are represented by themselves.

The next step generates rewritten tokens from the seed until no more new tokens can be created using rewriting rules. Rewriting is done by replacing content word lexical information with part of speech information or removing some parts of tokens. For example, case markers of nouns are removed for some rules. Actual rules are not presented because of space limitations.

This representation allows to easily match same forms of different words while getting the benefits of reverse index in terms of performance. A list of created tokens for raw sentence "私はゆっくりと白いご飯を食べた" is shown on Fig. 2.

Querying for single POS tag is not very useful – it is going to match any document for most of POS tags. In addition to that storing such tokens in the index will consume much of the index space. However, search in a case when POS is a child or parent of something is useful. Storing the information about parent and children POS for each bunsetsu allows answering POS dependency queries efficiently. By limiting such information only to nouns, verbs, adverbs and adjectives it is possible to store the POS dependencies **packed** – using only one additional byte per posting.

4.3 Search: Querying

Input search queries undergo two transformations. The first transformation is to analyze input query with a morphological analyzer and build an internal query representation. This representation is transferred to actual working nodes, where internal query representation is finally transformed to low level Lucene queries.

Analysis of input queries is performed in two steps. The first step removes special symbols from query replacing some of them with commas. This is done so that special symbols do not interfere with the morphological analyzer. This step is followed by morphological analysis using JUMAN. Results of the morphological analysis are merged with the information from special symbols forming **internal query rep-**

[7] http://samtools.github.io/hts-specs/SAMv1.pdf

resentation. Internal representation consists of one or more **parts**, separated in a raw query by commas. Each part consists of a **parent** with zero or more **children**.

There are five Lucene queries used. Two are built-in core queries and three are ad-hoc for using the postings information. Core Lucene queries are **term** and **boolean** queries. Boolean query is used to bind multiple query parts together. A term queries is used for parts without any children, because there is no need to use dependencies.

Ad-hoc Lucene queries use information in postings to select documents. The first one – **packed term query** – is also an extension of the core term query. It is used when a part has a POS-only parent or child. Its specific work is to compare both child and parent packed POS dependency from postings with a reference created from the query part.

For query parts when at least one of the children is not just a POS tag, the second one – **dependency query** – is used. The main idea behind it is to find trees which contain a conjunction of parent and all the children from the query part. Also, for each found tree, the child dependency numbers should be equal to the parent bunsetsu number. POS-only children are handled in the same way as in the packed term query.

The last one is used for supporting queries of type "A → EOS" which checks dependency numbers of postings.

By default Lucene uses tf-idf similarity for result ranking. Idf measure does not make any sense in case of pos-like tokens, so only tf is used in the system. Also, the length is normalized in a way so that moderately long (4-5 bunsetsu) sentences gain the highest score.

5 System state

The system is currently accessible online[8]. It is deployed on 55 slave nodes and 1 master node of our cluster system. The corpus used for search is a part of web corpus, of 700M sentences (about 1TB of compressed results of syntactic parse). Each node has approximately 20GB of compressed trees stored in the compressed database. The index size is 3GB per node on average.

We have measured search response times in this setting. In the experiment we used a list of 1600 queries that contains frequent verbs, nouns and adjectives with dependency queries like "~私を→分かる" with different frequent POS and words in parent and child place. Queries from that list were run once per 500ms and the response time was measured. For each search, trees for top 100 results were extracted from the database and sent as a response. 152179 response times were collected. 99% were less than 424 ms, 90% were less than 285 ms; median and average response times were 26 ms and and 179 ms respectively. We believe that current search speed is reasonably good for general usage. We should note however, that the system is presently deployed on a shared cluster which runs programs by other users at the same time, implying that dedicated installation is going to have more stable performance.

The system uses results of an automatic morphological analyzer and parser, so there are errors in analysis. Sentences with different words do appear in search results because of it. With the improvement of the analyzers this problem is going to disappear.

Source code of the system is going to be distributed under open-source license. It is also possible to match arbitrary tree structure without modifying indexing and tokenization, only on query level, however this has not been implemented yet.

[8] `http://lotus.kuee.kyoto-u.ac.jp/depfinder/search`

6 Conclusion

We have developed a distributed large scale sentence search engine that should be useful for linguists, researchers, teachers and students studying Japanese. It supports queries not only on lexical information, but also on POS, grammatic and dependency information as well. An installation that uses 700M sentences from web is available on the Internet. 90% of simple dependency queries get a response in 300 milliseconds.

A design of tokens which contained lexical, POS and grammatic information at a single place allows to use general search technology. Dependency trees were stored in a specialized compressed database.

Acknowledgements

The first author thanks Japanese Government for the support in Mongakubusho MEXT program for the foreign students.

References

[1] Milo Jakubíek, Adam Kilgarriff, Diana McCarthy, and Pavel Rychlý. Fast Syntactic Searching in Very Large Corpora for Many Languages. In *Proceedings of the 24th PACLIC*, pages 741–747, Tohoku University, Sendai, Japan, 2010. Workshop on Advanced Corpus Solutions.

[2] Keiji Shinzato, Tomohide Shibata, Daisuke Kawahara, and Sadao Kurohashi. TSUBAKI: An Open Search Engine Infrastructure for Developing Information Access Methodology. *Journal of Information Processing*, 52(12):216–227, 2011.

[3] 竹内淳平 and 辻井潤一. 係り受け関係と言い換え関係を用いた柔軟な日本語検索. In 言語処理学会第 *11* 回年次大会発表論文集, pages 568–571, 2005.