Token Gazetteer and Character Gazetteer for Named Entity Recognition

Giang Nguyen, Štefan Dlugolinský, Michal Laclavík, Martin Šeleng

Institute of Informatics, Slovak Academy of Sciences
Dúbravská cesta 9, 84507 Bratislava, Slovakia
{giang.ui, stefan.dlugolinsky, laclavik.ui, martin.seleng}@savba.sk

Abstract. Named entity recognition (NER) in information extraction (IE) systems is usually based on large gazetteers — datasets of well-known and classified entities. NER is also often performed by independent look-up piece of code, which is considered as a bottleneck of many NER systems. In this paper, we present two approaches for building tree gazetteers for NER; i.e. lookup by token and by character.

Keywords: gazetteer, information extraction, named entity recognition, tokenization

1 Introduction

Nowadays, named entity recognition (NER) is a very important area in computer science. A large amount of information is produced every day through numerous media around us. When some entity is classified as a well-known entity, the next step is to recognize its frequency in the incoming text and provide its references for next processing steps. In NER research area, word *gazetteer* is often used interchangeably for both the set of entity lists and for the processing resource that uses those lists to find occurrences of named entities in texts. There is a number of existing gazetteer implementations available; e.g. Ontotext's [2] contributions to GATE [1]: *Hash Gazetteer* – based on hash tables instead of FSM (Finite State Machine) with average four times less memory use and three times faster than an optimized FSM implementation, *Stand-Alone Gazetteer* – Java library, which can be used without GATE, *Large Knowledge Base Gazetteer* – provides support for ontology-aware NLP and *Linked Data Gazetteer* – experimentally uses Linked Open Data for lookups.

2 Token Gazetteer

Gazetteers usually do not depend on any other annotation and matching patterns are often based on the textual content of the documents. A gazetteer is usually a standalone lookup tool that allows occurrences of strings from predefined lists to be

adfa, p. 1, 2011.

found in texts. In his paper, we present two gazetteer approaches; i.e. token-based and character-based. Our token gazetteer is based on a red-black tree data structure, a special kind of a binary search tree. We have used *java.util.TreeMap<K, V>* implementation, which provides guaranteed *log(n)* time cost for the *containsKey, get, put and remove* operations, where *n* is a number of entries in the tree map. The tree structure is efficient for in-order traversal; i.e. left–parent–right (**Fig. 1**). Each node of the token gazetteer represents a *token* with features; e.g. property *NE type* denoting the named entity class. Tokens are loaded into the tree structure from gazetteer lists — simple text files, where each line contains one named entity (NE) and its features like references, type, note, etc.

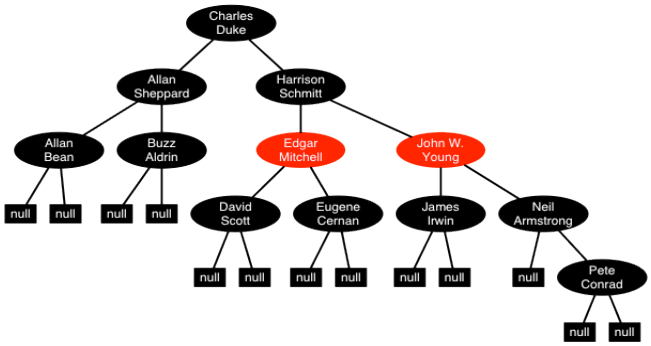

Fig. 1. A small example of filled red-black tree structure of token gazetteer

One of the most important parts of the token gazetteer is its *tokenizer*. It is because the *tokenizer* should properly split input text it into tokens, which are consequently searched in the tree structure. Tokenization is performed in two steps. In the first step, *tokenizer* splits the input text into tokens on whitespaces. Resulting tokens are further tokenized in the second step using a set of regular expressions (**Table 1**). These patterns can be adjusted as needed.

Regex	**Matches**	**Example**
\p{L}+	One or more code points in the category "letter"	(Slovenský)
[\p{L}\d]+	One or more letters or digits	#Apollo13.
[-\p{L}&]+	Letter sequences divided by – or &	Cartoon:Tom&Jerry
\p{S}+	One or more math symbols, currency signs, dingbats, box-drawing chars…	€45.00
\d+(?:[.,]\d+)*	Decimal or floating point numbers	$199.00-$249.00

Table 1. Tokenizer patterns

For instance, text "WIKT2013 (to be held in Herlany)" would be split into following six tokens: "WIKT2013", "(to", "be", "held", "in", "Herlany)" and in the second step, they would be further split into nine tokens: "WIKT", "2013", "(", "to", "be", "held", "in", "Herlany", ")". Tokens from the second step are then searched in the

gazetteer tree structure. Results are saved and the second-step tokens are abandoned. Search in the tree structure continues with searching for the first-step tokens and their sequences. This procedure is outlined in pseudo code below:

```
set results to empty
WHILE tokens on input
   set token ← next token from input
   set class ← value mapped on token in the gazetteer tree
   IF class not NIL THEN
      add class/token pair to results
   END IF
   set ceil ← ceiling key for value token + " " in the gazetteer tree
   WHILE tokens on input and ceil starts with token + " "
      set token ← " " + next token from input
      set class ← value mapped on token in the gazetteer tree
      IF class not NIL THEN
         add class/token pair to results
      END IF
   END WHILE
END WHILE
return results
```

Token gazetteer is able to run in case-less mode. In this mode, all list entities are stored in lowercase format and the input is also converted to lowercase. Furthermore, gazetteer supports custom identifiers to be specified for records in the lists; e.g. FreeBase MIDs type (name-of-entity MID), where MID is an ID of an object in Freebase. Token gazetteer has also a prefix search feature, which enables it to treat list records as prefixes and search for these prefixes in input tokens. This feature is usable for special cases like searching for words, which could have inflected forms. The advantage of the token gazetteer is in memory consumption, which grows linearly with the size of records in a gazetteer list. There is a new node for each record created in the tree. The memory consumption could be further utilized, because storing of records in the tree is not optimized; e.g. "Slovak" and "Slovak republic" – the string "Slovak" would be stored twice in the memory, which is not very effective.

3 Character Gazetteer

Tokenization in the token level deals with multi-travel through the input text, token boundaries, complexity of regular expressions, non-trivial entities with non-trivial characters. Due to these reasons, we have tried to construct gazetteer from characters, which would provide more precise results also for non-trivial cases. Our first idea came from exercise terms in the Information Retrieval course[1] at FIIT STU but unfortunately the first implementation contained impurities that made the code unusable. The implementation presented in this paper has been completely restructured and

[1] http://vi.ikt.ui.sav.sk/User:adamec?view=home

improved for right functionality, better performance and more effective memory usage. The character tree consists of a number of nodes, where each node contains:

- Character representing the node
- Reference to the parent node
- List of children nodes
- List of references (e.g. *MDI*), which also indicates if the node is a tree list node

There is a small example of filled character gazetteer tree structure in **Fig. 2**. Gray color indicates last characters of known entities. The tree structure enables fast and straightforward searching for all the possible entities in input text. In general, human languages are limited and therefore also the whole space of the tree structure, so it is possible to load whole tree into machine memory with current hardware possibilities.

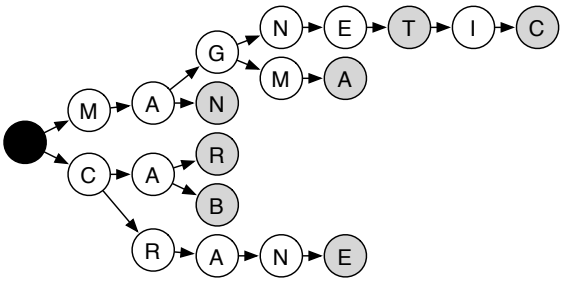

Fig. 2. A small example of filled tree structure of character gazetteer

The tree is implemented using *java.util.HashMap<K,V>* and the lookup time has an average-case complexity *O(1)*. Complexity of the tokenization algorithm is *O(n)*, where *n* is a number of characters in input text. It means, that we need to traverse the input text nearly one time to obtain results. The matching algorithm can be described briefly and in very simplified way as follows:

```
FOR each character on input buffer stream
   IF current node has a child node mapped on character THEN
      set current node ← child node mapped on character (go deeper in the tree)
      IF current node is a tree list THEN
         record the matched case for later use
      END IF
   ENF IF
END FOR
```

Our origin idea was to provide one-time-traverse matching algorithm, but the realization has shown that it is not possible to check all occurrences of all entities without the *"carry back"* part, which realizes jumps to previous positions of the first occurrence of white character in the last matched named entity; i.e. possible word start of other named entity. Therefore, the complexity of the tokenization algorithm is increased but not too much due to the fact that entity lengths are usually short.

4 Experiments and Evaluations

Both implementations of token and character gazetteers were tested on datasets acquired from FreeBase[2] – an online collection of structured data harvested from many sources with the aim of creating a global resource, which allows people and machines to access common information more effectively. Harvested data is lightly structured into triples (MDI, type, entity). FreeBase datasets contain millions of entities such as famous people names, well-known organizations or locations in the world. There were 3 171 393 person records, 1 498 862 location records and 846 602 organization records in the dataset, which was used for our experiments.

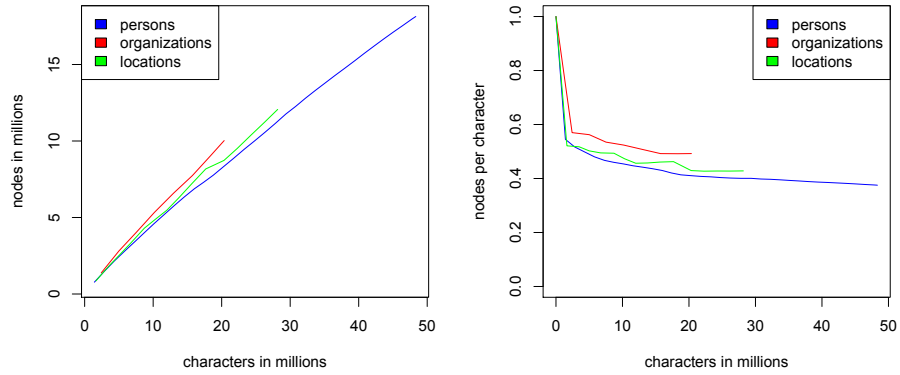

Fig. 3. Memory usage of the character gazetteer

Memory consumption of the character gazetteer is depicted in **Fig. 3**. It is affected by character divergence of strings in input datasets. Theoretically, if Unicode character representation is used, each node in the gazetteer tree can have as many children as the number of Unicode characters. Actually, there are 109 384 ($\approx 2^{17}$) code points assigned in Unicode 6.0. Fortunately, human language is limited and the tree of character gazetteer will not rise to its theoretical size. Our measures have shown, that growth ratio of the tree; i.e. number of nodes per character tends to decrease with the number of inserted entities or characters respectively (**Fig. 3**).

Character gazetteer provides linear complexity matching solution and fast entity recognition with precise results. Entities can contain various special characters; e.g. quotation marks, dash, dot, ampersand, copyright sign. They can have also overlapping parts.

We have compared our two gazetteer implementations with Ontotext's Hash Gazetteer [2]. The comparison was aimed on processing time and memory consumption. We have used a list of person names (3 171 393 instances) from FreeBase and populated all the tree gazetteer instances. The highest memory consumption was measured for character gazetteer (≈ 3 260 MB), then for hash gazetteer (≈ 900 MB) and the least for token gazetteer (≈ 865 MB). We have evaluated the processing time on a set of

[2] https://developers.google.com/freebase/data

1 390 text documents from CoNLL-2003 dataset. The test corpus was built by merging train, test A and test B CoNLL datasets. There were 10 measures made for each gazetteer over the test dataset. Results are depicted in **Fig. 4**. The box and bold line represent interquartile range (IQR) and median respectively. The whiskers stands for minimum and maximum defined as Q1 – 1.5 IQR and Q3 + 1.5 IQR respectively. The individual point indicates an outlier, which is out of the range of min/max whiskers. As we can see, character gazetteer has slightly outperformed hash gazetteer and significantly token gazetteer.

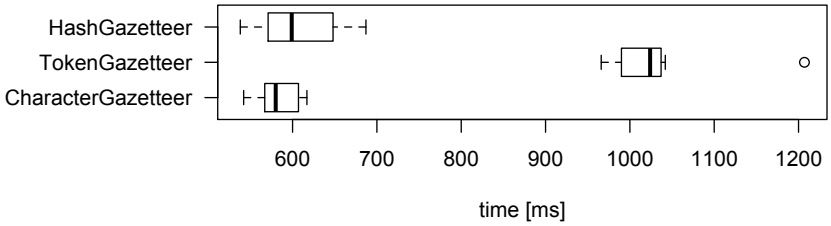

Fig. 4 Box plot of processing time for three different gazetteers

5 Conclusion

We use token gazetteer and character gazetteers in several of our projects. Character gazetteer is nearly 1.8 times faster in matching than token gazetteer. It also works better for non-trivial cases but consumes more memory than token gazetteer. Therefore at the moment we spent more efforts on character gazetteer development – we are also working on tree structure improvement with the aim to reduce memory consumption. Our gazetteers work with real big datasets with millions of entities on input text of arbitrary length.

Acknowledgement: This work is supported by projects VEGA 2/0185/13, CRISIS (ESF ITMS 26240220060) and TRADICE (APVV-0208-10).

6 References

1. GATE general architecture for text engineering http://gate.ac.uk/sale/tao/splitch13.html#sec:gazetteers:lkb-gazetteer
2. Ontotext: Hybrid Semantics and Metadata Management Solutions - GATE Components and Applications http://www.ontotext.com/collaborations/gate
3. Michal Laclavik, Martin Seleng, Marek Ciglan, Ladislav Hluchy: Ontea: Platform for Pattern based Automated Semantic Annotation. Computing and Informatics, Vol. 28, 2009, 555-579, ISSN 1335-9150
4. Erik F. Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 shared task: language-independent named entity recognition. In Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003 - Volume 4 (CONLL '03), Vol. 4. Association for Computational Linguistics, Stroudsburg, PA, USA, 142-147. DOI=10.3115/1119176.1119195 http://dx.doi.org/10.3115/1119176.1119195