Character Gazetteer for Named Entity Recognition with Linear Matching Complexity

Stefan Dlugolinsky, Giang Nguyen, Michal Laclavik, Martin Seleng
Institute of Informatics, Slovak Academy of Sciences
Dubravska cesta 9, 84507 Bratislava, Slovakia
Email: {stefan.dlugolinsky, giang.ui, laclavik.ui, martin.seleng}@savba.sk

***Abstract*—A large amount of unstructured data is produced daily through numerous media around us. Despite that computer systems are becoming more powerful, even the commodity hardware, processing of such data and gaining useful information in time efficient manner remains a problem. One of the domains in unstructured data processing is Natural Language Processing (NLP). NLP covers areas like information extraction, machine translation, word sense disambiguation, automated question answering, etc. All of these areas require fast and precise Named Entity Recognition (NER), which is not a trivial task because of the processed data size and heterogeneity. Our effort in this research area is to provide fast tokenization and precise NER with linear complexity. In this paper, we present a character gazetteer with linear tokenization as well as NER and compare its two tree data structure representations; i.e. multiway tree implemented by hash maps and first child-next sibling binary tree. Our measurements shows that one outperforms the other in processing time, while the other outperforms it in memory consumption efficiency.**

***Keywords*-gazetteer; named entity recognition; natural language processing; text processing; tokenization;**

I. Introduction

Natural Language Processing (NLP) is a very important and interesting area in computer science affecting also other spheres of science, for instance molecular biology. A large amount of textual data is continuously produced in numerous media around us and therefore there is a need of processing it in order to gain required information. One of the most important processing steps in NLP is Named Entity Recognition (NER). The task of NER is to recognize occurrence of known entities in input texts usually coming from websites, web portals, social media, data dumps and documents. Known entities might be of arbitrary type, but the most used types in NER are names of persons, organizations, locations, times, and quantities. One of the commonly used NER techniques is *gazetteer* – a simple list of known entities, which are looked up in the input text. We use the term gazetteer interchangeably also with the term named entity dataset. There are various sources of named entities available; e.g. Google Data Dumps[1] of FreeBase[2], DBpedia[3], so it is not a problem to build up gazetteer lists for miscellaneous domains. In general, gazetteers do no depend on previously discovered tokens nor annotations. They expect raw text on input and find matches based on its content. According to the way of handling input text, there are two main approaches that gazetteers follow to find matches – token-based and character-based. Token-based gazetteers split input text into a sequence of tokens on which a matching is performed, while character-based gazetteers process input text character by character. The approach of input handling determines internal representation of the gazetteer list. Token-based gazetteers use hash maps and/or token trees, while character-based gazetteers use character trees, which behave like a finite-state machines (FSM). Both approaches affect memory consumption and speed. We present a summary of available existing gazetteers together with description of their approaches in the following section.

II. Related Work

Several gazetteer implementations are provided by Ontotext[4] in GATE [1]; i.e. *Hash Gazetteer* – gazetteer based on hash tables instead of FSM. Authors declare that it takes in average four times less memory and that it works three times faster than an optimized FSM implementation; *Stand-Alone Gazetteer* – version of the Hash Gazetteer that can be used without GATE as a Java library with minimal efforts in any application that needs to look-up huge lists of strings in text in a time and memory efficient manner.; *Large Knowledge Base Gazetteer*[5] – new-generation gazetteer, which provides support for ontology-aware NLP. It allows using of large gazetteer lists and speeds up subsequent loading of data by caching.; *Linked Data Gazetteer* – an experimental gazetteer that uses Linked Open Data for lookups.

Besides Ontotext gazetteers, there is a number of scientific works dealing with NER using comparable approaches. Authors in [2] present a gazetteer implemented as an FSM. Gazetteer is built at initialization time starting from the list of phrases that need to be later recognized with contextual dependency. Contextual dependency is also one of the assump-

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

[3]http://dbpedia.org/
[4]http://www.ontotext.com/collaborations/gate
[5]http://gate.ac.uk/sale/tao/splitch13.html#sec:gazetteers:lkb-gazetteer

tions in [3]. In the work [4], authors deal with automated gazetteer construction as well as with various problems such as entity-noun ambiguity, entity-entity ambiguity and entity boundary detection. A high-level rule-based language for building and customizing NER annotators is described in [5]. Unfortunately, the process of designing the rules themselves is manual and time-consuming, but the rule-based approach performance is comparable to that of machine learning. Pattern-based matching and validation in an unlabeled corpus is described in [6] and [7] respectively. The outcome is that patterns can be quite complex to fulfill different requirements. An evaluation of six different NER tools over microposts is provided in [8]. Some of the evaluated tools rely on gazetteers when detecting named entities. Other tools use word-based tokenization and chunking together with local text features to detect named entities with a help of machine learning techniques.

In general, tokenization is rarely performed outside gazetteer. It is because of its purpose to only find entities and not to deal with disambiguation. There are two main ways of tokenization distinguished in gazetteers; i.e. token-level and character-level tokenization. Gazetteers with token-level tokenization (from here on referenced as token gazetteers) faces several problems such as the need of multi-travel search and matching, finding word boundaries, chunking or processing of non-trivial strings and characters. These problems lead to longer running time and low processing performance of this kind of gazetteers. Therefore, in this paper, we deal with gazetteer based on character-level tokenization (from here on referenced as character gazetteer) and present an approach, which provides more precise results and much better performance than token-level equivalent. Our first idea of character gazetteer came from exercise terms in the Information Retrieval course at FIIT STU[6], but unfortunately the first implementation contained impurities that made the code unusable. The implementation presented in this paper has been restructured and improved for right functionality, better performance and more effective memory use.

III. Gazetteer Tree Data Structure

Entity datasets as the main source of gazetteer lists are usually large and have tendency to grow (adding new known entities). This fact significantly affects memory requirements, running time as well as the tokenization complexity, especially for token gazetteers. Therefore a new gazetteer data structure is required to fold entities at character level, which also enables fast linear tokenization of input text and produces required output such as references, occurrence quantity and positions in the input text. Tree structure is a powerful way of organizing data hierarchically and suits these considerations very well. Moreover, it provides easy and quick access operations in order to find elements and traverse through the structure. Therefore we have chosen tree data structure to represent one-to-many parent-child relations of characters in character gazetteer.

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

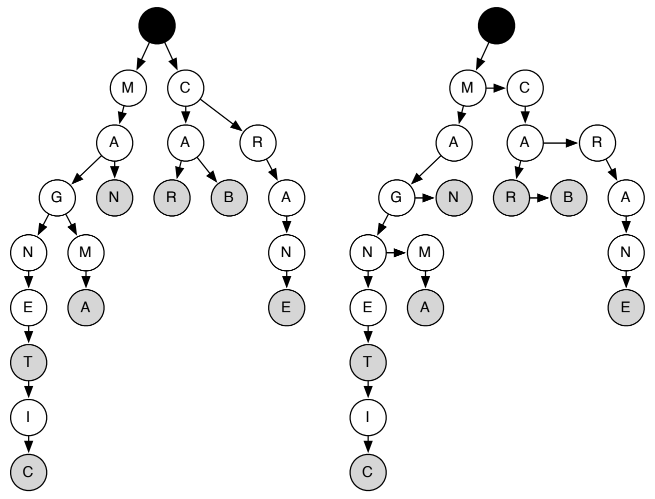


Figure 1. Two different representations of the same character gazetteer tree: (left) multiway tree and (right) first child-next sibling binary tree

We have considered two representations of the tree data structure; i.e. *multiway tree* and *first child-next sibling binary tree*. Multiway tree (from here on referenced as HMT – a hash map tree) was implemented by Java HashMap, which provides constant-time performance ($O(1)$ in average) for the basic operations (get and put). Each node of the tree had a hash map in which it could have arbitrary number of child nodes as its values; i.e. following characters. Hash maps were created with the default initial capacity (16) and the default load factor (0.75). The *first child-next sibling binary tree* (from here on referenced as CST – child-sibling tree) is a binary representation of the multiway tree, where each node refers only to its first child node and its next sibling node. This representation is not as fast as the HMT, but is more efficient in memory consumption. It is also expected that gazetteer trees would grow slower with the number of inserted entities because of human language limitation.

A simple example of the character gazetteer tree in both representations is depicted in Figure 1. Seven different entities are split into characters and arranged in the tree; i.e. magnetic, magnet, magma, man, car, cab and crane. Gray color indicates last characters of the entities and the black node is the root of the tree. The tree structure enables fast and straightforward search for all the possible entities in the input text.

Both HMT and CST are filled with Algorithm 1. Filling algorithm inserts provided entity into the gazetteer's tree so the gazetteer will be able to find it in the input text. Algorithm expects on input a string representing the inserted entity. It loops over all its characters while traversing and building the tree. Finally, a node representing the last

Algorithm 1 Insert entity into the character gazetteer tree

```
1: $node \leftarrow$ root node
2: for all $character$ in $entity$ do
3:     if $node$ has a child with $character$ then
4:         $node \leftarrow$ child
5:     else
6:         $child \leftarrow$ new node with $character$
7:         add $child$ to $node$
8:     end if
9: end for
10: mark $node$ as a matching node
```

character of the input string is marked as a matching node. To support case-insensitivity, entities should be converted to either lowercase or uppercase prior to insertion.

IV. Linear Tokenization and Named Entity Recognition

After a tree gazetteer is filled with entities, it is ready to be used on tokenization and finding of known entities in the input text. This process is performed by Algorithm 2. Complexity of the 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 the results. This algorithm is intended to be used on a stream of text of unlimited size.

Initially, we wanted strictly one-way traverse of input text, but the realization revealed that it is impossible to recognize all occurrences of NEs this way, especially the overlapping ones; e.g. a tweet *"I'm in museum of london"*, where two entities could be found: *ORG/museum of london* and *LOC/london*. Therefore an "unread" part on lines 12-14 was required to perform "rewind" of the input stream at the position of possible start of overlapping named entity. The tokenization complexity was increased but not too much, because entities are usually short.

Algorithm 2 Matching algorithm of the character gazetteer

```
1: $buf \leftarrow$ empty
2: $node \leftarrow$ root node
3: while characters on input do
4:     $ch \leftarrow$ next character from input
5:     normalize whitespace or skip multiple
6:     if $node$ has a child node mapped on $ch$ then
7:         add $ch$ to $buf$
8:         $node \leftarrow$ child node mapped on $ch$
9:         if $node$ is marked as a matching node then
10:            found an entity
11:        end if
12:    else
13:        if $buf$ contains character other than letter or digit
           then
14:            unread characters from $buf$ back on input until
               the first occurrence of the character that is not
               letter or digit
15:        end if
16:        $buf \leftarrow$ empty
17:        $node \leftarrow$ root node
18:    end if
19: end while
```

Case-insensitivity can be handled easily in the main loop of the matching algorithm. It is just required to convert the read character into lower-case or upper-case. It depends on the form chosen in the filling algorithm. Matching algorithm also deals with different whitespace characters, which are normalized to standard space (0x20). In addition, subsequent whitespace characters are skipped and treated as a single character. All of this is because of strict matching, which could fail on word sequences separated by multiple or non-standard whitespace characters.

V. Evaluation

Implementations of the CST and HMT character gazetteers were tested on memory consumption and processing time. There were three FreeBase datasets and one Wikipedia dataset used for the memory consumption testing; i.e. Freebase organizations (778,814 unique entities), Freebase locations (1,256,552 unique entities), Freebase persons (2,614,401 unique entities) and Wikipedia titles and alternative names (9,319,611 unique entities). While the Freebase datasets were obtained from Google Data Dumps, we have built Wikipedia dataset from XML dump file by extracting article titles and their alternative names. Wikipedia dataset was the largest one that we have used for the testing.

Figure 2 depicts measured values in the memory consumption tests of CST and HMT gazetteers. Both gazetteers were filled with the same data and compared. Measurements showed that CST required about 75% less memory than HMT. Also the bootstrap of the CST gazetteer was about 20% faster.

The ratio of characters stored per single tree node is depicted in Figure 3. We can see that the tree has a tendency to grow slower with the number of inserted entities as there can be proportionally more characters stored per one node. The weakness of the character gazetteer is in its memory consumption, especially for the HMT implementation, where each parent node requires to allocate memory for its hash map. Theoretically, each tree node can have so many child nodes as is the number of characters in the charset set, thus the tree structure can be very wide. For instance Unicode standard. Unicode code points are divided into 17 planes each of 65,536 (2^{16}) code points (characters). In version 6.1, six of these planes have assigned code points and are named. About ten percent of the potential space of Unicode is used at the moment. More concretely, Unicode can have 1,111,998 characters (= 17 planes $\times$ 65,536 characters/plane

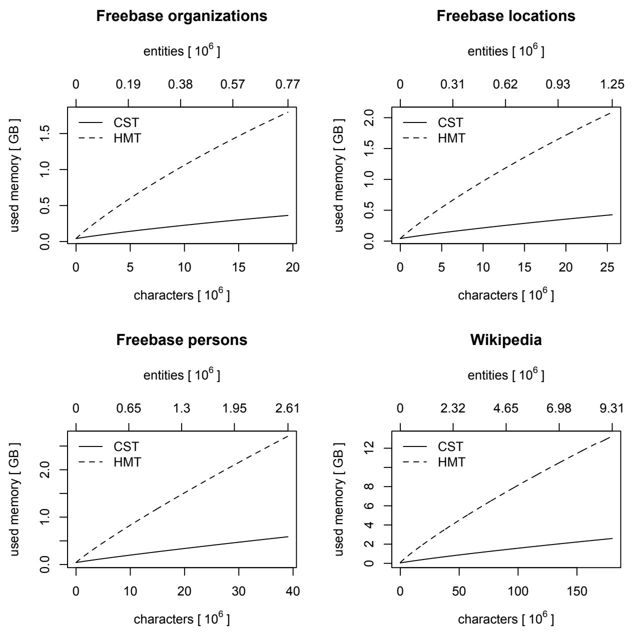


Figure 2. Memory consumption of character gazetteer tree according to different tree representation and fill data

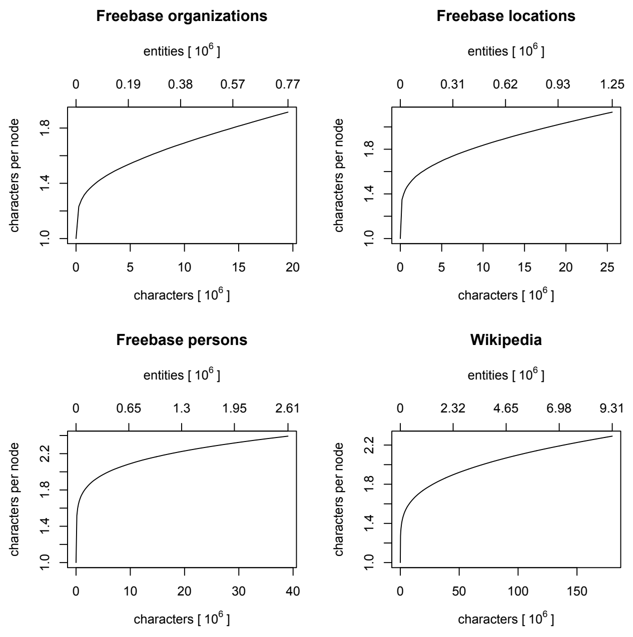


Figure 3. Average number of characters stored per single node in the character gazetteer tree

$-$ 2048 surrogates $-$ 66 non-characters) but currently just 109,384 ($\approx 2^{17}$) code points are actually assigned. Then maximum number of nodes in the gazetteer tree can be less than $\sum_{i=1}^{n} 2^{17^{i}}$ where n is the longest length of entities. Fortunately, the tree space is limited by human language and all combinations will never be covered, thus the management of the gazetteer tree in machine memory is possible also on a commodity hardware.

Processing time was measured by executing both gazetteer implementations over a set of 9,909 documents acquired from CoNLL-2003 datasets [9] with approximately 29 MB of text. Tested gazetteers were filled with the Freebase persons dataset. The test was repeated five times, while in each of the runs, there were four measures taken after each 7 MB of processed text. Repeated measures were were averaged and the results are depicted in Figure 4. We can see that HMT gazetteer has significantly outperformed the CST variant, which required more time on traversing its tree structure.

We have made some processing time tests also in [10], where we compared character gazetteer with Ontotext HashGazetteer and our implementation of token gazetteer through red-black tree. One of the tests was performed over a set of 1,390 documents from the CoNLL-2003 dataset. Gazetteers were filled with Freebase persons entities and executed. The result was that our character gazetteer slightly outperformed Ontotext HashGazetter, but its drawback was in 3.5 times larger memory requirements.

An example of the character gazetteer matching is depicted in Figure 5. There is a screenshot of a simple testing tool with an input text taken from Wikipedia on the left and a list of matched entities on the right together with their frequencies. Matched entities are highlighted in the

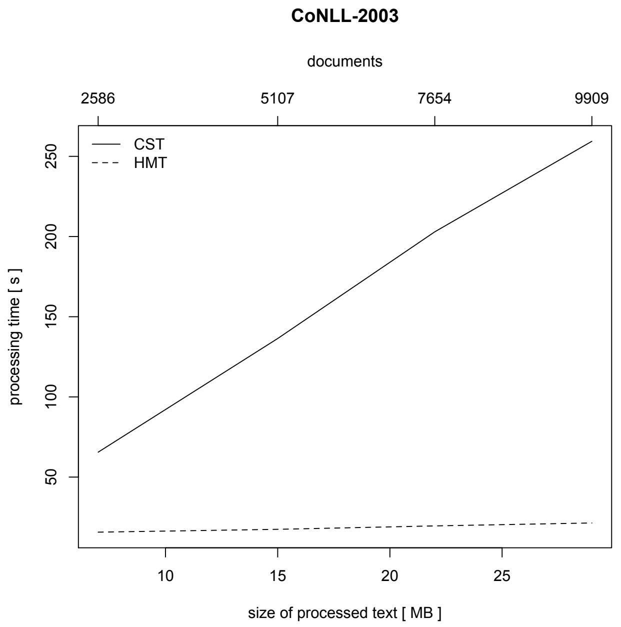
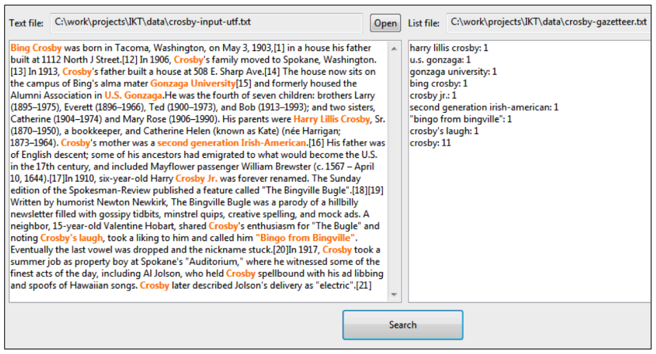


Figure 4. Processing times of Freebase persons filled CST and HMT gazetteers on a set of CoNLL-2003 documents


Text file: C:\work\projects\IKT\data\crosby-input-utf.txt

Bing Crosby was born in Tacoma, Washington, on May 3, 1903,[1] in a house his father built at 1112 North J Street.[12] In 1906, Crosby's family moved to Spokane, Washington.[13] In 1913, Crosby's father built a house at 508 E. Sharp Ave.[14] The house now sits on the campus of Bing's alma mater Gonzaga University[15] and formerly housed the Alumni Association in U.S. Gonzaga.He was the fourth of seven children: brothers Larry (1895–1975), Everett (1896–1966), Ted (1900–1973), and Bob (1913–1993); and two sisters, Catherine (1904–1974) and Mary Rose (1906–1990). His parents were Harry Lillis Crosby, Sr. (1870–1950), a bookkeeper, and Catherine Helen (known as Kate) (née Harrigan; 1873–1964). Crosby's mother was a second generation Irish-American.[16] His father was of English descent; some of his ancestors had emigrated to what would become the U.S. in the 17th century, and included Mayflower passenger William Brewster (c. 1567 – April 10, 1644).[17]In 1910, six-year-old Harry Crosby Jr. was forever renamed. The Sunday edition of the Spokesman-Review published a feature called "The Bingville Bugle".[18][19] Written by humorist Newton Newkirk, The Bingville Bugle was a parody of a hillbilly newsletter filled with gossipy tidbits, minstrel quips, creative spelling, and mock ads. A neighbor, 15-year-old Valentine Hobart, shared Crosby's enthusiasm for "The Bugle" and noting Crosby's laugh, took a liking to him and called him "Bingo from Bingville". Eventually the last vowel was dropped and the nickname stuck.[20]In 1917, Crosby took a summer job as property boy at Spokane's "Auditorium," where he witnessed some of the finest acts of the day, including Al Jolson, who held Crosby spellbound with his ad libbing and spoofs of Hawaiian songs. Crosby later described Jolson's delivery as "electric".[21]

List file: C:\work\projects\IKT\data\crosby-gazetteer.txt

```
harry lillis crosby: 1
u.s. gonzaga: 1
gonzaga university: 1
bing crosby: 1
crosby jr.: 1
second generation irish-american: 1
"bingo from bingville": 1
crosby's laugh: 1
crosby: 11
```

Figure 5. Highlighted positions of matched NEs in the sample input text

input text. character gazetteer provides linear complexity matching solution with fine-grained feature 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 or have similar features; e.g. noun, adjective, with postfix and prefix.

VI. Conclusion

We have presented a linear matching algorithm for character gazetteer and compared two tree data structures of the gazetteer. In comparison to HMT structure, CST structure benefits from memory saving, but it slows down the matching algorithm as there are more operations required for the basic tree operations. We would like to continue in improving the three data structure in order to decrease its memory requirements and make it more efficient for traversing. One possible direction of improvement could be in collapsing of nodes. Presented implementations of the character gazetteer are available for download[7].

Acknowledgment

This work is supported by projects VEGA 2/0185/13 “New methods and approaches on information processing and knowledge bases”, CRISIS (ESF ITMS 26240220060) “Research and development of new information technologies for forecasting and solving crisis situations and for the security of populace” and CLAN (APVV-0809-11) “Cloud Computing for Big Data Analytics”.

References

[1] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, N. Aswani, I. Roberts, G. Gorrell, A. Funk, A. Roberts, D. Damljanovic, T. Heitz, M. A. Greenwood, H. Saggion, J. Petrak, Y. Li, and W. Peters, *Text Processing with GATE (Version 6)*, 2011. [Online]. Available: http://tinyurl.com/gatebook

[2] D. Maynard, V. Tablan, C. Ursu, H. Cunningham, and Y. Wilks, “Named entity recognition from diverse text types,” in *Recent Advances in Natural Language Processing 2001 Conference*, 2001, pp. 257–274.

[3] X. Liu, S. Zhang, F. Wei, and M. Zhou, “Recognizing named entities in tweets,” in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, ser. HLT ’11. Stroudsburg, PA, USA: Association for Computational Linguistics, 2011, pp. 359–367. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002472.2002519

[4] D. Nadeau, P. Turney, and S. Matwin, “Unsupervised named-entity recognition: Generating gazetteers and resolving ambiguity,” in *Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, L. Lamontagne and M. Marchand, Eds. Springer Berlin Heidelberg, 2006, vol. 4013, pp. 266–277. [Online]. Available: http://dx.doi.org/10.1007/11766247_23

[5] L. Chiticariu, R. Krishnamurthy, Y. Li, F. Reiss, and S. Vaithyanathan, “Domain adaptation of rule-based annotators for named-entity recognition tasks,” in *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP ’10. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 1002–1012. [Online]. Available: http://dl.acm.org/citation.cfm?id=1870658.1870756

[6] M. Laclavik, L. Hluchý, M. Seleng, and M. Ciglan, “Ontea: Platform for pattern based automated semantic annotation,” *Computing and Informatics*, vol. 28, no. 4, pp. 555–579, 2009.

[7] Z. Kozareva, “Bootstrapping named entity recognition with automatically generated gazetteer lists,” in *Proceedings of the Eleventh Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, ser. EACL ’06. Stroudsburg, PA, USA: Association for Computational Linguistics, 2006, pp. 15–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=1609039.1609041

[8] Štefan Dlugolinský, P. Krammer, M. Ciglan, and M. Laclavík, “MSM2013 IE Challenge: Annotowatch,” in *Making Sense of Microposts (#MSM2013) Concept Extraction Challenge*, 2013, pp. 21–26. [Online]. Available: http://ceur-ws.org/Vol-1019/paper_21.pdf

[9] E. F. Tjong Kim Sang and F. De Meulder, “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*, ser. CONLL ’03. Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, pp. 142–147. [Online]. Available: http://dx.doi.org/10.3115/1119176.1119195

[10] G. Nguyen, Š. Dlugolinský, M. Laclavík, and M. Šeleng, “Token gazetteer and character gazetteer for named entity recognition,” in *In 8th Workshop on Intelligent and Knowledge Oriented Technologies*. Nakladateľstvo STU, 2013.

[7]http://ikt.ui.sav.sk/gazetteer