INFORMATION SYSTEMS IN MANAGEMENT

Information Systems in Management (2018) Vol. 7 (4) 227−236
DOI: 10.22630/ISIM.2018.7.4.20

TOWARDS THE DATA STRUCTURE FOR EFFECTIVE WORD SEARCH

WALDEMAR KARWOWSKI, PIOTR WRZECIONO

Department of Informatics, Warsaw University of Life Sciences (SGGW)

In the paper problem of searching basic forms for words in the Polish language is discussed. Polish language has a very extensive inflection and effective method for finding base form is important in many NLP tasks for example text indexing. The method for searching, based on open-source dictionary of Polish language, is presented. In this method it is important to design a structure for storing all words from dictionary, in such a way that it allows to quickly find basic words forms. Two dictionary structures: ternary search tree and associative table are presented and discussed. Tests are performed on the six actual and three crafted artificial texts and results are compared with other possible dictionary structures. At the end conclusions about structures effectiveness are formulated.

Keywords: Text Indexing, Associative Array, Ternary Search Tree

1. Introduction

Today, English is the dominant language not only in the Internet but also in many areas, primarily connected with science and technology. However, at the same time, the national languages are commonly used and generally the number of published documents is increasing. A huge number of documents prevents person from viewing them directly. Automatic analysis of the texts can help in reviewing big sets of documents. Especially text indexing is an important issue today, it helps to organize documents and it is widely applicable in knowledge management. During indexing, the very important task is to reduce inflectional and derivationally related forms of a word to a common base form. The process of reducing words to their base or root form is possible with stemming algorithms. The Polish language has a very extensive inflection and stemming is not an easy task in this situation. Effective stemming algorithms are designed for English, but they are not useful in Polish. Review of stemmers for Polish language is made in [10]. There are available two stemmer tools: Stempel [9] - algorithmic stemmer for Polish language and Morfologik [7] - morphosyntactic dictionary for the Polish language, which has the stemmer tool. The problem of automatic text indexing in agriculture domain, for documents in Polish language, was discussed by authors in [3, 4, 11]. Because dictionaries in Morfologik or Stempel were not big and integration with our system was not easy, we designed special custom method. We have to note that generally stemming extracts rather root that base form, but for our goals we need very simple stemming just finding base form for word. Our method involves use an open-source dictionary of Polish language [8]. This dictionary is an open project licensed under the GPL and CC SA licenses. It is continuously updated and contains more than 200000 records. Every record consists of base form and all inflected forms derived from it. We prepared collection of all words with pointers to base form; as a result, we can easily find the base form for any word. The basic problem is effective search for words in the collection, because we have together almost 4 million words. Simple method involves storing sorted words in a huge array together with pointers to base form and using the binary search. This method was applied in [3, 4, 11], it is memory consuming but works quite well. To improve searching we need more specialized structures than sorted array. Dictionary structure based on a trie was presented and tested in [5]. The primary disadvantage of trie structure is huge amount of needed RAM memory. In this paper we want to present two other structures: ternary search tree (TST) and custom dictionary based on associative array, which requires much less memory than trie and at the same time ensures relatively high efficiency. Main goal of the research was to implement, test and compare search efficiency for all mentioned structures. The results allow us to design and implement the best dictionary for the indexing task.

The rest of this paper is organized as follows: in Sect. 2 the dictionary structure based on a ternary search tree is presented and discussed. In Sect. 3 the associative array is shortly described. In Sect. 4 subsequently testing results for sorted list, trie, TST and associative array based on hash table are presented and discussed. Summary and final remarks are formulated in Sect. 5.

2. Dictionary structure based on a ternary search tree

In [5] we presented that a subsequent search of words in a sorted array, which contains almost four million items, takes some time, but the structure based on the trie makes the same task many times faster. The main disadvantage of structure based on trie is its huge size in memory. The ternary search tree is a kind of search tree where nodes are arranged in a manner similar to a binary search tree, but with three children: left, middle and right. Similarly to trie, each node in the TST stores only one character (a part of key) [1]. The left child stores character value which is less than the character in the current node. The right child stores character which is greater than the character in the current node. Middle child stores next character in the word. Additionally each node stores flag which denotes possible end of word. In other words each node in a ternary search tree represents a prefix of the stored strings. All strings in the middle subtree of a node start with that prefix.

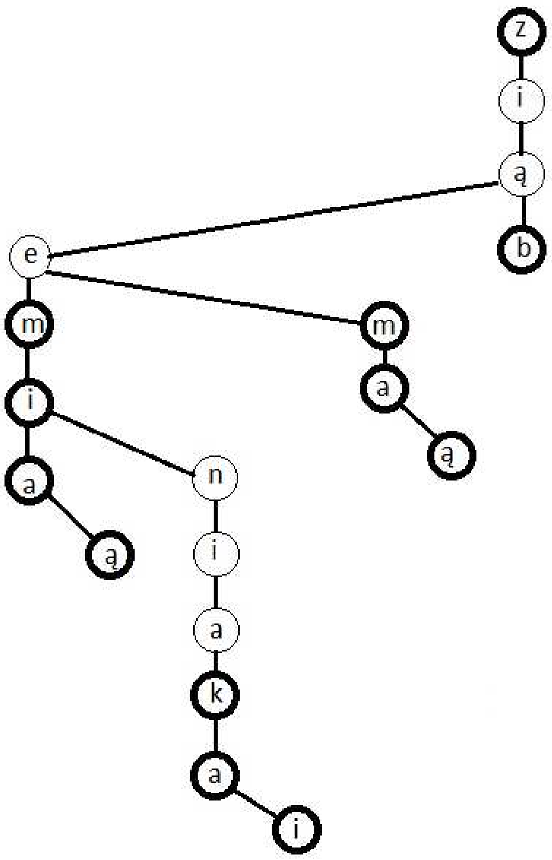

Figure 1. Words with the common prefix in TST

In the Fig. 1 ternary search tree for words: *z*, *ziąb*, *ziem*, *ziemi*, *ziemia*, *ziemią*, *ziemniak*, *ziemniaka*, *ziemniaki*, *zim*, and *zima*, is presented. The thick line is used for nodes which are the end of the word. For example node e represents prefix *zie*, node n represents prefix *ziemn*. Because we compare Unicode values of characters, value of the letter *ą* (0105 in hex) is bigger than value of the letter e (0065 in hex) what we can see in the Figure 1. We can compare unicodes because in searching, unlike in sorting, order of letters in the Polish alphabet is not important. Moreover we have to note that TST unlike trie depends on order of adding words. The dictionary based on TST allows to store big and small letters, which make possible to distinguish own names. In structure based on trie, such situation almost doubled size of every node which significantly increases its size in memory.

3. Custom associative array structure based on hash table

An associative array is a known data structure based on key/value pairs. The associative array has a set of keys and each key has a single associated value. For presented key, the associative array will return the associated value. An associative array is also called a map or a dictionary. Associative arrays are often implemented based on hash tables [2, 6]. There are many ready implementations of associative arrays in many programming languages, especially in Java or .NET libraries.

In our situation we have pairs (word, index) where index means index in table with indexes to base forms table. Of course word is a key and index is a value. We have to note that we cannot immediately points to base forms table, we need additional table with indexes to base forms table, because in Polish language we have many homographs. For example word *mają* has possible base forms: *maić* (verb), *mieć* (verb) or *maja* (noun - Lithodes maja). This additional table was implemented also in dictionary implementation based on trie described in [5].

To store hashed keys we prepared huge table with size over 4 million cells (i.e. bigger than number of words), we chosen 7199369 because it is prime number. For every word we count hash code in the interval [0, 7199368]. In other words at the last step of counting hash code for a given word, we take hash(word) as hash(word) mod 7199369. Such code hash(word) is taken as an index in the table, but cell value is additional index to array which stores pairs (word, index). Because it may happen that a few words have the same hash code (i.e. we have conflict), from this reason array which stores pairs (word, index) has additional attribute next i.e. it stores triples (word, index, next). Attribute next points to next cell where word has the same hash code.

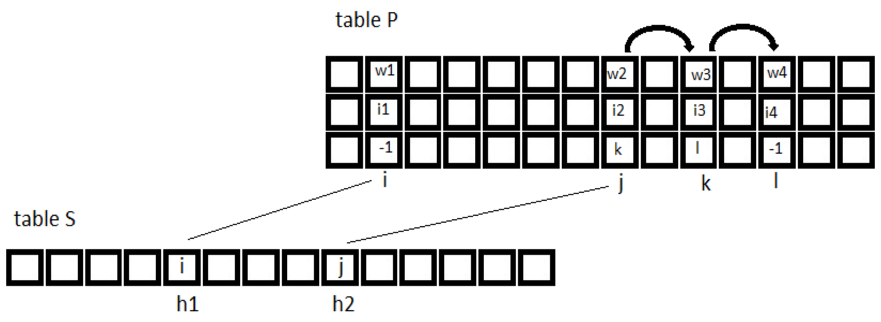

Figure 2. Association array based on hash table

In the Fig. 2 indexes in the table S are hash codes, for example h1 = hash(w1), h2=hash(w2), h2 = hash(w3) and h2 = hash(w4). Table B stores triples (word, index, next). Hash code h1 is unique, it points to triple at index i in the table P it is (w1, i1, -1). Index i1 points to the table with indexes to base forms table, but in this situation, attribute next in this triple is equal -1 (it denotes that we have not next words with the same hash code). Hash code h2 is not unique, in such situation j points to the first word w1 (i.e. triple with word w1) with hash code h2 in the table P it is (w2, i2, k). Index i2 points to the table with indexes to base forms table, but attribute next has value k. It means it points to the next word w3 with the hash code h2. Analogously next attribute has value l and points to third word with hash code h2 i.e. w4. Because we have not more words with hash code h2, triple (w4, i4, -1) has attribute next equals -1, it means that it was last word with hash code h2.

In other words we can say that to every cell in table S we attach list (chain) of pairs (word, index). We have to note that table S is huge to reduce to minimum number of conflicts. Average performance for associative array is constant, but maximum number of conflicts sets a worst-case performance.

4. Comparing effectiveness of dictionary structures

As in [5] our application was implemented in C#. There are 204050 base forms and total 3953809 words in dictionary [8], but without homographs we have 3801058 different words. We prepared six versions of our application based on: sorted array, trie structure, TST, associative array, .NET Dictionary class and .NET HashSet class. The first two versions were tested in [5], we tested them again because they were slightly improved in comparison with [5]. The following two versions are based on structures described in the previous sections, the last two versions based on .NET Framework 4.7.2. We tested applications based on Dictionary and HashSet classes because they are very similar to our implementation of associative array.

First we measured amount of RAM memory needed to store whole dictionary structures. Dictionary with sorted array of strings takes about 150MB of RAM memory, dictionary with trie structure takes more than 1.1GB of RAM memory, dictionary structure based on TST takes about 250MB of RAM memory, our associative array takes about 200MB of RAM memory, dictionary based on .NET Dictionary class takes about 250MB of RAM memory, similarly solution based on .NET HashSet class.

Next we prepared to measure average performance (time complexity) of searching in our applications. Theoretically binary search is equivalent to searching in balanced binary search tree and average performance is O(log n) where n is the number of words ($\log_2$ 3953809 = 21,9148117485834 ≈ 22). For binary search standard Array.BinarySearch method from .NET library was used. In trie structure searching performance does not depend on number of words, it depends on longest word (i.e. number of characters). In our dictionary [1] this number is 39 (*niedziewięćdziesięciopięcioipółletniego*), of course average performance is O(1). For TST structure average performance is O(log n) and worse O(n), but many depends on words and their order. In our case deepest branch of TST is 106. Of course it is bigger than 22 for binary search but in TST every time we compare only one pair of characters but for binary search pair of whole words. Associative array has constant searching performance O(1). Many depends on longest chain of words with the same hash code, in our situation this number is 6, and we have 2953044 different hash codes which means that most often we have only one element chain. In associative array we used standard .NET method GetHashCode() defined for strings, obtained value is taken without sign and modulo 7199369. For .NET Dictionary and HashSet searching performance is O(1). We have to add that for HashSet we had to defined EqualityComparer class and defined method GetHashCode() (for pair (word, index) we take word.GetHashCode()).

To practically compare the search results of two dictionary structures, we selected several publications from Agricultural Engineering Journal (Inżynieria Rolnicza) exactly the same that were used in the paper [5]. “Text A” is “Information system for acquiring data on geometry of agricultural products exemplified by a corn kernel”; “Text B” is “Assessment of the operation quality of the corn cobs and seeds processing line”; “Text C” is “Methodological aspects of measuring hardness of maize caryopsis”; “Text D” is “Evaluation of results of irrigation applied to grain maze”; “Text E” is “Extra corn grain shredding and particle breaking up as a method used to improve quality of cut green forage”; and “Text F” is “ Comparative assessment of sugar corn grain acquisition for food purposes using cut off and threshing methods”. Additionally, we have prepared three “artificial” texts. “Text X” contains two thousand times word contains; this word is not present in the Polish language dictionary. “Text Y” contains two thousand times word *niewybielały* and “Text Z” contains two thousand times word *niewybielałych*.

The results of the test are presented in Table 1 – Table 9, similarly like in [5]. In the header we put the number of words in the particular text. The measure is the number of processor ticks. Every test was taken two times: for one thousand loops, and ten thousand loops. The reason is that .NET Just In Time compiler prepares methods before the first run, if we run method next time, compiled method code is in memory. The result is influenced by a certain overhead during the first loop, it is something like *overhead_tics + n * tics_for_one_loop*. For bigger n we can better estimate average number of tics for one loop taking ratio tics/number of loops. Of course from many reasons, connected with .NET environment, number of tics can differ between two runs, but differences are not significant. We did the tests many times for every case and the results differed insignificantly.

Table 1. Text A (1655 words)

Number of loops	Binary search	Trie search	TST search	Associative array	NET Dictionary	NET HashSet
1000	33591897	342741	975962	160745	292288	243338
10000	316131531	3485992	9736250	1563227	2887232	2432310

Table 2. Text B (2622 words)

Number of loops	Binary search	Trie search	TST search	Associative array	NET Dictionary	NET HashSet
1000	53222999	500002	1355375	250009	469488	394600
10000	467090971	5009019	13488726	2452428	4665948	3940382

Table 3. Text C (2286 words)

Number of loops	Binary search	Trie search	TST search	Associative array	NET Dictionary	NET HashSet
1000	45597531	429401	1190847	213183	397152	331165
10000	406983513	4248994	11779211	2072389	3870395	3284391

Table 4. Text D (1429 words)

Number of loops	Binary search	Trie search	TST search	Associative array	NET Dictionary	NET HashSet
1000	28353564	250092	725084	130667	232243	211289
10000	256743505	2509729	7216749	1279460	2294568	2044434

Table 5. Text E (1618 words)

Number of loops	Binary search	Trie search	TST search	Associative array	NET Dic-tionary	NET HashSet
1000	32357241	327902	894725	158679	292624	240832
10000	307635139	3255707	8889389	1543999	2928692	2422897

Table 6. Text F (1963 words)

Number of loops	Binary search	Trie search	TST search	Associative array	NET Dic-tionary	NET HashSet
1000	39847235	356195	1097629	178936	323735	281710
10000	353018299	3530443	10278696	1739347	3182355	2787582

Table 7. Text X (2000 words)

Number of loops	Binary search	Trie search	TST search	Associative array	NET Dic-tionary	NET HashSet
1000	48424413	331754	1600371	101902	114812	205132
10000	459046035	3411198	16180765	1038634	1145758	2000187

Table 8. Text Y (2000 words)

Number of loops	Binary search	Trie search	TST search	Associative array	NET Dic-tionary	NET HashSet
1000	4884409	664130	1606728	224155	434106	331085
10000	48845906	6727924	16121447	2087378	4192262	3161444

Table 9. Text Z (2000 words)

Number of loops	Binary search	Trie search	TST search	Associative array	NET Dic-tionary	NET HashSet
1000	55643747	750973	1708222	262994	437585	332897
10000	503264504	7589691	17459160	2601833	4290230	3261948

We summarized all results in the Table 10, taking approximate number of tics, leaving only two important digits, to show general tendency. We can observe that searching results with associative array are the best. Anyway they are comparable with versions based on .NET structures: Dictionary and HashSet. Version based on trie is about 2 times slower, but still comparable with associative array. Searching with TST is significantly slower, about 15 times. This confirmed our previous theoretical considerations. Binary search is very slow regarding to all other methods, for example associative array is more than 200 times faster than binary search. We can observe that for example for texts D and E the TST searching was relatively slow, it means the tree structure affects search time. For associative array and .NET structures results are rather proportional to number of words in tested text. In this situation results for text X, word contains hash code not presented in dictionary and negative result is obtained immediately.

Table 10. Approximate number of tics for one loop

Text	Binary search	Trie search	TST search	Associative array	NET Dic-tionary	NET HashSet
A	31000	340	970	160	290	240
B	46000	500	1300	250	460	390
C	40000	420	1100	200	380	320
D	25000	250	7200	130	230	200
E	30000	320	8800	150	290	240
F	35000	350	1000	170	320	280
X	45000	340	1600	100	110	200
Y	48000	670	1600	200	420	310
Z	50000	750	1700	260	430	330

5. Conclusions and future work

We examined six dictionary structures for text analysis in particular for indexing text. Tests have shown that the structure based on the associative array makes searching faster than other structures. Five structures but trie utilize similar amount of RAM memory. For our purpose dedicated association table is the best choice. If somebody does not want to implement dedicated structure classes from .NET library are relatively good. However, the trie structure and TST are still useful in tasks such word completion or error correction. Trie is very good choice if we have limited dictionary, for dictionaries like [8] the better choice for mentioned tasks is TST which compromises the advantage of fast completion and reasonable RAM amount. The main conclusion is that our application should be developed based on associative array but parallel version with trie structure can be useful in special task like morphology study.

REFERENCES

[1] Bentley J., Sedgewick R., (1998) *Ternary Search Trees*. Dr. Dobbs Journal April, 1998

[2] Cormen, T. H., Leiserson, C. E.; Rivest, R. L.; Stein, C., (2001), *Chapter 11 Hash Tables, Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill

[3] Karwowski W., Wrzeciono P., (2014) *Automatic indexer for Polish agricultural texts*. Information Systems in Management 2014, Vol. 3, nr 4, pp. 229-238

[4] Karwowski W., Wrzeciono P., (2017) *Methods of automatic topic mining in publications in agriculture domain*. Information Systems in Management 2016, Vol. 6 (3) pp 192-202

[5] Karwowski W., Wrzeciono P., (2017) *The dictionary structure for effective word search*. Information Systems in Management 2017, Vol. 6, (4), s. 284-293

[6] Mehlhorn, K., Sanders, P. (2008), *Chapter 4 Hash Tables and Associative Arrays, Algorithms and Data Structures: The Basic Toolbox*, Springer

[7] *Morphosyntactic dictionary for the Polish language* https://github.com/morfologik/

[8] *Polish language dictionary*, http://www.sjp.pl

[9] *Stempel - Algorithmic Stemmer for Polish Language* http://getopt.org/stempel/

[10] Weiss D. (2005) *A Survey of Freely Available Polish Stemmers and Evaluation of Their Applicability in Information Retrieval*. 2nd Language and Technology Conference, Poznań, Poland, pp. 216-221

[11] Wrzeciono P., Karwowski W. (2013) *Automatic Indexing and Creating Semantic Networks for Agricultural Science Papers in the Polish Language*, Computer Software and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual, Kyoto