Serdica J. Computing **13** (2019), No 1–2, 71–80

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

ON FUZZY MATCHING OF STRINGS*

Lyubomir Filipov, Zlatko Varbanov

Abstract. Fuzzy matching is a widely used technique in computer-assisted translation and some other fields (it is implemented in most database engines and is used in autocompleting of data, for example). In this paper, fuzzy matching in the aspects of approximate string matching is investigated. Basic algorithms like Soundex, Bitap, Boyer-Moore [1, 2] are covered. Using the results about those algorithms, several database engines are compared and a new way of handling fuzzy matching is offered.

1. Introduction. Fuzzy matching is known technique that is proven to work with matches that are not 100% perfect when finding correspondences between segments of a text and entries in a database. In some database engines (PostgreSQL, MongoDB) fuzzy matching is implemented but practically its usage is in the aspects of approximate string matching. Our research is focused to database systems and for that reason here we do not consider the aspects of fuzzy

ACM Computing Classification System (1998): H.3.3, I.5.4.

Key words: fuzzy matching, approximate string matching.

*This research was partially supported by University of Veliko Tarnovo Science Fund under Contract FSD-31-653-07/19.06.2017.

matching related to computer-assisted translation (about this topic see [6], for example).

In fuzzy matching if we have "Thomas Johnes", "Tom Johnes", and "Tomas Johnes", three of them will be all linked to the same person because it depends on how they sound and how they are pronounced. One of the best known algorithms based on homophones (a word that is pronounced the same as other word but varies in meaning) is Soundex. As is pointed out from Knuth [3] the first letter of input is retained and after that numbers are assigned to the remaining letters. Soundex could be named the ancestor of all homophones based algorithms. Soundex is used in ASR (Automated Speech Recognition)[5] because it is faster to match names preprocessed with the algorithm to the correct corpus entry. Bitap algorithm [4] is widely related to fuzzy matching and it is used in the Unix *agrep* function. Such algorithms are used within DNA pairs [7]. A common approach is used behind one of the top effective Unix functions *grep*.

2. Preliminaries. Generally, the problem for recognition of similar names in their various versions is an important practical task but in the current work the initial interest to this subject arose from a practical problem related to public utilities (electric power companies, water supply systems, natural gas companies, etc.). Some companies use only their websites where to post information there will be issues with the corresponding public utilities (no automated system to alarm via web services, SMS, or email messages). It is not convenient for the clients to watch over their websites all the time.

It is the main reason to create a specified web crawler that checks given website in every five minutes and it notifies all the people that were on the queue. After downloading the information for some time we have noticed that there are some mismatches between the ways how the streets and neighborhoods are named. It occurred because there are people that write the names of the streets/neighborhoods on keyboard without using a common name system. Because of typos and different styles of spelling the names there are mismatches in the data. For that reason we started to test different database engines and systems that could help us to determine what is the exact name of the street/neighborhood that is referenced.

Approximate string matching [4]. The approximate string matching problem is to find all locations at which a query of length $\boldsymbol{m}$ matches a substring of a text of length $\boldsymbol{n}$ with ***k-or-fewer*** differences.

Fuzzy matching is a partial case of **full-text search**. It refers to searching a single sequence or collection in a full text database. It differs from search based

on metadata or on parts of the original texts represented in databases (titles, abstracts, selected sections, or bibliographical references). It examines all of the words in every stored document as it tries to match the search criteria. Unix *grep* tool is performing full-text search with a strategy called "serial scanning" which scans the contents of the documents directly on the fly. In database platforms this search is organized using indexes (scanning the text of all the documents and building a list of search terms) and queries are performed on the referenced index, not on the original text.

Test Preparation. In order to compare several sources of information we need a common database shared among all items. We have generated a special database that contains company names. The relational database table contains only two fields ***id***, which is of type **INT** and ***companyName*** which is of type **VARCHAR(255)**. The generated table consists of **50 000 rows of unique company names**. This database was applied to different database engines and the results were compared.

MySQL is a relational database management system based on SQL (Structured Query Language). The application is used for a wide range of purposes, including data warehousing, e-commerce, and logging applications. However, the most common use of MySQL is for the purposes of web databases.

Fuzzy Matching with MySQL. If we have the company name "Agamba" and there is one typo in it, we have to make $3*n+1$ queries where n is the length of the string (company name in our case). This is very important because if there are any issues we have to check the actual letter, the symbol before the actual letter, and the symbol after that. In the case for "Agamba":

_Agamba, gamba, _gamba, A_gamba, Aamba, A_amba, Ag_amba, Agmba, Ag_mba, Aga_mba, Agaba, Aga_ba, Agam_ba, Agama, Agam_a, Agamb_a, Agamb, Agamb_, Agamba_

In MySQL, it has to be implemented using LIKE query. However (according to the documentation), LIKE queries could not be optimized and this means that they are going to perform a full search. The actual SQL query in MySQL looks like this:

```
SELECT * FROM
    `companyList`
WHERE
    `companyNames` LIKE '%_Agamba%'
        OR `companyNames` LIKE '%gamba%'
        OR `companyNames` LIKE '%_gamba%'
```

```
OR `companyNames` LIKE '%A_gamba%'
OR `companyNames` LIKE '%Aamba%'
OR `companyNames` LIKE '%A_amba%'
OR `companyNames` LIKE '%Ag_amba%'
OR `companyNames` LIKE '%Agmba%'
OR `companyNames` LIKE '%Ag_mba%'
OR `companyNames` LIKE '%Aga_mba%'
OR `companyNames` LIKE '%Agaba%'
OR `companyNames` LIKE '%Aga_ba%'
OR `companyNames` LIKE '%Agam_ba%'
OR `companyNames` LIKE '%Agama%'
OR `companyNames` LIKE '%Agam_a%'
OR `companyNames` LIKE '%Agamb_a%'
OR `companyNames` LIKE '%Agamb%'
OR `companyNames` LIKE '%Agamb_%'
OR `companyNames` LIKE '%Agamba_%';
```

Tests were performed on an Intel(R) Core(TM) i3-2350M CPU @ 2.30GHz with 8GB DDR2 RAM, with a regular HDD. The results on that machine were **15 results in 30 ms**. MySQL is not effective to be used for fuzzy matching. If we have a sequence with two mistakes, then we have to perform $(3 * n + 1)^2$ checks.

Apache Lucene is a search engine library written in Java, that is marked as high-performance and full-featured. It is pointed out to be suitable for any application that requires full-text search.

Algolia is a hosted full-text, numerical, and faceted search (exploring information by applying multiple filters on information elements with multiple explicit dimensions) engine capable of delivering real-time results from the first keystroke. Its model is flexible enough to accept the evolution and the multi-faceted nature of entities. As the number of documents is not bound to any limitation, then the scalability is an important expectation [2].

Fuzzy Matching with Algolia. Algolia is a paid service which runs over the top of Apache Lucene. The power of Algolia comes from their structure and servers. Compared to the others it is a **hosted service** which means that all the data should be hosted on their servers (it would be a drawback for certain cases where the data is really sensitive). It is a **paid service** which means that it is not suitable for all kinds of projects. When data be applied, the desired indexes are auto built which is easier than building the indexes by hand (especially when the user is not a database expert). It also supports fuzzy matching calls by default

without running extra query or writing custom functions. The results acquired from Algolia were **15 results in 1 ms**.

ElasticSearch (or simply Elastic) is a search engine based on Lucene. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. ElasticSearch is developed in Java and it is released as open- source under the terms of the Apache License.

Fuzzy Matching with Elastic. It is open-source and it could be installed on own servers. Elastic supports fuzzy matching by default but it has a limitation of up to two mistakes in a string sequence. It is not really straightforward to move all the data from another database or service to Elastic. When it comes to use Elastic, the users have to build the indexes by themselves. Building the correct indexes in this structure is very important (otherwise the results won't be correct). Using Elastic we have received **15 results in 5ms**, but we have to notice that Elastic was installed on our machine, not on a hosted server environment. Compared to Algolia, Elastic could be used if the data is sensitive and should not be open, if there are enough server resources and if there is database expert who can create the correct indexes.

3. Modification. After careful investigation, it was found that the mistakes occurred when typing on a keyboard are well processed in Google's services.

Google Suggest Queries. When search in Google, there is a feature that helps someone to get the desired content. If one is searching for "coding thoery", the message "Did you mean: coding theory" will appear. The Google search engine analyses all the searches that have been made and it is storing information about the location from which the search is coming, the results that have been shown and how many times user has clicked on certain items from the list. All this information based on the region and the users preferences is held by Google.

If we perform a query to

$http://suggestqueries.google.com/complete/search?output=toolbar\&hl=en\&q=$**ninaj**

trying to find an information, we will get the following JSON response:

```
[  "ninaj",
   [  "ninja",
      "ninja turtles",
      "ninja blender",
```

```
...
    ],
    { "google:suggestrelevance":[
            1250,
            601,
            600,
        ],
        ...
    }
]
```

In Table 1 the query parameters and the corresponding descriptions are listed. First part of JSON array shows what we have been searching for **ninaj** which is typo of the word **ninja**. After that we get the most famous results among other users that have type **ninaj**, which are also marked with **suggestrelevance**. From the result we could see that **ninja** is the first suggestion.

http : *//suggestqueries.google.com/complete/search?output* = **toolbar**&*hl* = **bg**&*q* = **ninjas**

Table 1. The query parameters and their descriptions

Parameter	Description
output	Type of response you want.
hl	Language's 2 letter abbreviation.
q	Your search item.

The API is really powerful giving the opportunity to select what kind of output the user expects (XML or JSON). Using the **hl** property the end users could provide the API with information from which region they need results (whether it will be based on all the stored results from Bulgaria, USA, Germany, etc).

Typos. When typing on keyboard people are making lots of typos but most of them are already well investigated in **SEO optimization** of websites. If we have to classify typos we could point out: **Skip letter**, **Double letters**, **Reverse letters**, **Skip spaces**, **Missed key**, **Insert key**.

We need to **generate all possible errors** in the words that we have if the set of words is known and it is not so large. In our case we have to generate all possible typos that could be made with the names of the streets/neighbourhoods. After that, we have to process all of these typos to the Google Suggest API and store the received results in a database. This means that each time when we get

no 100% match we could check our database in order to find the valid name of the street/neighbourhood.

The generation of typos can be made on keyboard press and it will not be based on the keyboard layout. It is easier to find all possible combinations of keyboard press typos and then apply different keyboard layouts over them.

Table 2. The keyboard enumeration used

$\sim_{K_1}$	1_{K_2}	2_{K_3}	3_{K_4}	4_{K_5}	5_{K_6}	6_{K_7}	7_{K_8}	8_{K_9}	$9_{K_{10}}$	$0_{K_{11}}$	$-_{K_{12}}$
$TAB_{K_{13}}$	$Q_{K_{14}}$	$W_{K_{15}}$	$E_{K_{16}}$	$R_{K_{17}}$	$T_{K_{18}}$	$Y_{K_{19}}$	$U_{K_{20}}$	$I_{K_{21}}$	$O_{K_{22}}$	$P_{K_{23}}$	$[_{K_{24}}$
$Caps\ lock_{K_{25}}$	$A_{K_{26}}$	$S_{K_{27}}$	$D_{K_{28}}$	$F_{K_{29}}$	$G_{K_{30}}$	$H_{K_{31}}$	$J_{K_{32}}$	$K_{K_{33}}$	$L_{K_{34}}$	$;_{K_{35}}$	$'_{K_{36}}$
	$<_{K_{37}}$	$Z_{K_{38}}$	$X_{K_{39}}$	$C_{K_{40}}$	$V_{K_{41}}$	$B_{K_{42}}$	$N_{K_{43}}$	$M_{K_{44}}$	$,_{K_{45}}$	$._{K_{46}}$	

If we use the standard keyboard, we could get all the neighbouring keys of **"J"**, which are **H, K, U, I, N, M**. In Table 2, a map of these with ids like **Q**—is K_{14}, **W**—K_{15}, **E**—K_{16}, etc., is presented. This means that **J** will be K_{32}, **H**—K_{31}, **K**—K_{33}, **N**—K_{43}, **M**—K_{44}, **U**—K_{20}, and **I**—K_{21}. With respect to the key **"F"** (K_{29}) the neighbouring keys are: **R**—K_{17}, **T**—K_{18}, **D**—K_{28}, **G**—K_{30}, **C**—K_{40}, **V**—K_{41}. Once we have this map generated we could apply different layouts on it. For example, we could apply Bulgarian Phonetic Layout and generate all the possible typos with the letter **"Й"** or **"Ф"**. The essential idea is that we are not restricted by the layout in order to generate all possible combinations with six neighbours. It doesn't require a special algorithm but is only a list of all possible neighbours.

K_1—K_2, K_{13}, K_{14}
K_2—K_1, K_3, K_{13}, K_{14}
K_3—K_2, K_4, K_{14}, K_{15}
K_4—K_3, K_5, K_{15}, K_{16}
K_5—K_4, K_6, K_{16}, K_{17}
K_6—K_5, K_7, K_{17}, K_{18}
K_7—K_6, K_8, K_{18}, K_{19}
K_8—K_7, K_9, K_{19}, K_{20}
K_9—K_8, K_{10}, K_{20}, K_{21}
K_{10}—K_9, K_{11}, K_{21}, K_{22}
K_{11}—K_{10}, K_{12}, K_{22}, K_{23}
K_{12}—K_{11}, K_{23}, K_{24}

K_{13}—K_1, K_{14}, K_{25}
K_{14}—K_2, K_3, K_{13}, K_{15}, K_{25}, K_{26}
K_{15}—K_3, K_4, K_{14}, K_{16}, K_{26}, K_{27}

K_{16}—K_4, K_5, K_{15}, K_{17}, K_{27}, K_{28}
K_{17}—K_5, K_6, K_{16}, K_{18}, K_{28}, K_{29}
K_{18}—K_6, K_7, K_{17}, K_{19}, K_{29}, K_{30}
K_{19}—K_7, K_8, K_{18}, K_{20}, K_{30}, K_{31}
K_{20}—K_8, K_9, K_{19}, K_{21}, K_{31}, K_{32}
K_{21}—K_9, K_{10}, K_{20}, K_{22}, K_{32}, K_{33}
K_{22}—K_{10}, K_{11}, K_{20}, K_{22}, K_{33}, K_{34}
K_{23}—K_{11}, K_{12}, K_{22}, K_{24}, K_{34}, K_{35}
K_{24}—K_{12}, K_{23}, K_{35}, K_{36}

K_{25}—K_{13}, K_{26}
K_{26}—K_{13}, K_{14}, K_{25}, K_{27}, K_{37}, K_{38}
K_{27}—K_{15}, K_{16}, K_{26}, K_{28}, K_{38}, K_{39}
K_{28}—K_{16}, K_{17}, K_{27}, K_{29}, K_{39}, K_{40}
K_{29}—K_{17}, K_{18}, K_{28}, K_{30}, K_{40}, K_{41}
K_{30}—K_{18}, K_{19}, K_{29}, K_{31}, K_{41}, K_{42}
K_{31}—K_{19}, K_{20}, K_{30}, K_{32}, K_{42}, K_{43}
K_{32}—K_{20}, K_{21}, K_{31}, K_{33}, K_{43}, K_{44}
K_{33}—K_{21}, K_{22}, K_{32}, K_{34}, K_{44}, K_{45}
K_{34}—K_{22}, K_{23}, K_{33}, K_{35}, K_{45}, K_{46}
K_{35}—K_{23}, K_{24}, K_{34}, K_{36}, K_{46}
K_{36}—K_{24}, K_{35}

K_{37}—K_{25}, K_{26}, K_{38}
K_{38}—K_{26}, K_{27}, K_{37}, K_{39}
K_{39}—K_{27}, K_{28}, K_{38}, K_{40}
K_{40}—K_{28}, K_{29}, K_{39}, K_{41}
K_{41}—K_{29}, K_{30}, K_{40}, K_{42}
K_{42}—K_{30}, K_{31}, K_{41}, K_{43}
K_{43}—K_{31}, K_{32}, K_{42}, K_{44}
K_{44}—K_{32}, K_{33}, K_{43}, K_{45}
K_{45}—K_{33}, K_{34}, K_{44}, K_{46}
K_{46}—K_{34}, K_{35}

Once having the list above we could apply different layouts to it in order to generate the possible mistakes (according to Table 2). It is easy by simply providing an array containing the layout data. In order to do this, we need an

initial array of the possible typos related to the layout that we are interested in and an input string sequence $S = s_1, s_2, ..., s_n$ of length n. For each symbol s_i $(1 \leq i \leq n)$ we find the corresponding symbol in the layout array. After that we generate all possible typos related to that symbol according to Table 2, and store them in the output array.

4. Conclusions. If there is a known set (with relatively small number elements) of fuzzy information, we could generate all the possible typos corresponding to any string sequence with elements from this set and after that to store the obtained results in a database. Once we have all the data we could make queries to Google Suggest Queries API and get the most common information that was expected when having a given typo (Skip letter, Double letters, Reverse letters, Skip spaces, Missed key, Insert keys). After matching the typos with the correct phrases we could continue with scraping our data making sure that we could handle everything that is coming.

The proposed approach is usable in autocomplete and search with typos related to different systems. It is not practically usable for large sets of data.

Remark: The authors are aware that in some database engines fuzzy matching is implemented as an internal command, but the current approach is specific to similar problems as described above (related to issues of companies for public utilities).

REFERENCES

[1] Boyer R., J. Moore. A fast string searching algorithm. *Communications of the ACM*, **20** (1977), No 10, 762–772.

[2] Dessaigne N., J. Martinez. A Model for Describing and Annotating Documents. In: Proceedings of the Conference on Information Modelling and Knowledge Bases XVII, 2006, 1–19.

[3] Knuth D. E. The Art of Computer Programming. Volume 3: Sorting and Searching. Addison-Wesley, 1973, 394–395.

[4] Myers G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. In: Proceedings of 9th Annual Symposium on Combinatorial Pattern Matching (CPM '98), Piscataway, New Jersey, USA, July 20–22, 1998, 1–13.

[5] Raghavan H., A. James. Using Soundex codes for indexing names in ASR documents. In: Proceedings of the Workshop on Interdisciplinary Approaches to Speech Indexing and Retrieval at HLT-NAACL 2004 (SpeechIR '04), 22–27.

[6] Vanallemeersch T., V. Vandeghinste. Improving fuzzy matching through syntactic knowledge. In: Proceedings of the 36th Translating and the Computer Conference, Westminster, London, UK, 2014, 90–99.

[7] Wu S., U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, **35** (1992), No 10, 83–91.

Lyubomir Filipov
Department of Informational Technologies
University of Veliko Tarnovo
2, T. Tarnovski Str.
Veliko Tarnovo, Bulgaria
e-mail: `lubomir_g_1991@abv.bg`

Zlatko Varbanov
Department of Informational Technologies
University of Veliko Tarnovo
2, T. Tarnovski Str.
Veliko Tarnovo, Bulgaria
e-mail: `vtgold@yahoo.com`

Received August 31, 2018
Final Accepted June 29, 2019