UDC 004.02
DOI: 10.15587/2706-5448.2023.286382

Kyrylo Kleshch,
Volodymyr Shablii

COMPARISON OF FUZZY SEARCH ALGORITHMS BASED ON DAMERAU-LEVENSHTEIN AUTOMATA ON LARGE DATA

The object of research is fuzzy search algorithms based on Damerau-Levenshtein automata and Levenshtein automata. The paper examines and compares solutions based on finite state machines for efficient and fast finding of words and lines with a given editing distance in large text data using the concept of fuzzy search.

Fuzzy search algorithms allow finding significantly more relevant results than standard explicit search algorithms. However, such algorithms usually have a higher asymptotic complexity and, accordingly, work much longer.

Fuzzy text search using Damerau-Levenshtein distance allows taking into account common errors that the user may have made in the search term, namely: character substitution, extra character, missing character, and reordering of characters. To use a finite automaton, it is necessary to first construct it for a specific input word and edit distance, and then perform a search on that automaton, discarding words that the automaton will not accept. Therefore, when choosing an algorithm, both phases should be taken into account. This is because building a machine can take a long time. To speed up one of the machines, SIMD instructions were used, which gave a speedup of 1–10 % depending on the number of search words, the length of the search word and the editing distance.

The obtained results can be useful for use in various industries where it is necessary to quickly and efficiently perform fuzzy search in large volumes of data, for example, in search engines or in autocorrection of errors.

Keywords: *fuzzy search, Levenshtein automaton, Damerau-Levenshtein distance, editing distance, finite state machines.*

Received date: 05.07.2023
Accepted date: 24.08.2023
Published date: 28.08.2023

How to cite
Kleshch, K., Shablii, V. (2023). Comparison of fuzzy search algorithms based on Damerau-Levenshtein automata on large data. Technology Audit and Production Reserves, 4 (2 (72)), 27–32. doi: https://doi.org/10.15587/2706-5448.2023.286382

1. Introduction

In the modern information society, large arrays of text data are becoming a necessary component in many areas of human activity, such as: trade, medicine, science, economy, and information technology. Fuzzy search allows to efficiently find the information you need, where there may be inaccuracies, errors or incomplete information. However, searching for the necessary information in such large data sets can be laborious and time-consuming.

The basic idea behind fuzzy search is to find strings that closely match a given pattern string called a search term. Unlike an exact search, where it is necessary to find an exact match to a pattern, a fuzzy search allows for some error or inaccuracy between the specified string and the searched string. At the same time, each match is given a numerical characteristic – the similarity degree of the found string to the template [1]. This type of search is widely used in search engines, as it allows getting close results, even in the absence of exact matches in the query. Thus, fuzzy search improves the user experience and provides more accurate and correct search results.

Also, fuzzy search has an important application in the field of computer vision. Optical character recognition systems often face problems due to noise, artifacts, font variations, and other factors that can cause errors in the recognized text. The use of fuzzy search methods helps to improve the quality of word recognition and correct errors [2].

Fuzzy search algorithms are not new, and have been used for quite some time. However, the amount of information that needs to be and can be stored in various data repositories has increased significantly recently. It is for this reason that the emphasis should be placed on the speed of execution and optimization of the algorithm, even if the accuracy will decrease slightly. One of the key concepts in fuzzy search for evaluating the similarity of two strings is the concept of edit distance. Editing distance is defined as the minimum number of conversion operations on one line to make it identical to another.

A widely used type of editing distance is the Levenshtein distance. It is defined as the minimum number of operations (insertion, deletion and replacement of characters) required transforming one string into another. To more accurately account for user errors when typing, a modified Lowenstein distance, known as the Damerau-Lowenstein distance, is used [3]. It adds one more operation – the transposition of two symbols. One of the most common methods for calculating the Levenshtein distance is the algorithm developed by Robert Wagner and Michael

ISSN 2664-9969

Fischer [4]. Its main idea is to apply dynamic programming to find the editing distance. There are also alternative approaches to algorithms based on dynamic programming, for example, the use of finite automata that accept all strings that differ from a given pattern by no more than a given distance [1].

As part of the study, the task of finding words on a large set of textual data, which is constantly updated and changed, was considered. The user may search for words without knowing the correct spelling, and may make mistakes while typing on the keyboard.

The aim of research is to analyze various fuzzy search algorithms and choose the best one for a specific set of input data. Build dependencies and draw conclusions about the optimal Damerau-Levenshtein automaton for search words of different length and editing distance. Analyze the feasibility of building an automaton with small input data. This will make it possible to implement a whole system of fuzzy search using several algorithms based on finite state machines, which will choose the optimal approach depending on the search word and input data.

2. Materials and Methods

The object of research is fuzzy search algorithms based on Damerau-Levenshtein automata and Levenshtein automata. A graphical representation of a finite state machine can be presented in the form of a state diagram or transition table. A state diagram provides an intuitive visual representation of states, transitions, and final states, while a transition table provides detailed information about each state and all possible transitions. Fig. 1 shows an example of an automaton that accepts lines ending in an odd number of «a» characters in the form of a state diagram [3]. The states of the automaton are indicated by circles, and the initial state is indicated as 0. The double circles are responsible for the final states of the automaton. Transitions between states are indicated by arrows.

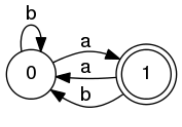


Fig. 1. Image of the automaton in the form of a state diagram

Finite automata are divided into 2 types: Deterministic Finite Automaton (DFA) and Nondeterministic Finite Automaton (NFA). The main difference between DFA and NFA lies in the transition behavior. DFAs have a unique transition for each input symbol, while NFAs can have multiple transitions or ε-transitions for a single input symbol. Instead, ε-transitions are called transitions that read an empty string. When moving to a state with ε-transitions, the automaton finds itself simultaneously in the states to which these transitions lead. The machine is shown in Fig. 1 is deterministic because it has a unique transition for each state and symbol, and therefore has only a single current state [5].

2.1. Conducting an experiment. In practice, 4 fuzzy search algorithms were compared: the standard Damerau-Devenshtein algorithm and 3 different editing distance calculators TreeAutomaton, HashAutomaton, and Table-Automaton. For each algorithm, software implementations were written in the C++ programming language.

2.2. TreeAutomaton. The initial stage is the construction of a non-deterministic automaton in the form of a prefix tree. To build an automaton that would accept words that differ from the template by no more than a given distance, it is necessary to go through all possible variants of operations on the template, the total cost of which is less than the maximum allowed. Step-by-step description of the algorithm:

1. As long as the queue is not empty, let's take the first element from the queue and process it. Each queue element corresponds to a combination of the current character from the pattern and the remaining edit distance.

2. If the state index is equal to the word length, let's note the current node as the final state [6].

3. It is provided that the total score with the insertion value does not exceed the maximum allowable distance, let's create a new node in the tree with the insertion symbol and add a new state to the queue.

4. Let's try to create new states with different actions: insertion, deletion, transposition and replacement of symbols. If the total current penalty amount does not exceed the maximum possible editing distance, then let's add such a state to the queue and create it.

In Fig. 2, it is possible to see an example and the result of the construction of the NFA for the template «ab» with a maximum editing distance of 1. The states that are the nodes of the tree are marked with a circle. The double circle is responsible for the final states of the automaton. The symbol «?» any character is marked, including «a» and «b», initial state is 0. This automaton accepts any words that have an edit distance to the pattern «ab» less than or equal to 1. For example, for the input word «ba» edit distance is 1 because one transposition is enough to turn it into an «ab» pattern. When reading the first character «b» let's find ourselves in the states marked as 10 and 1. After that, when reading «a» let's go to states 11 and 2. State 11 is final, so the word is accepted by the automaton. To calculate the distance, when building the automaton, it is necessary to store in each node of the tree a value equal to the sum of the distances of the previous operations performed in order to reach this node. This value represents the editing distance of the sequence.

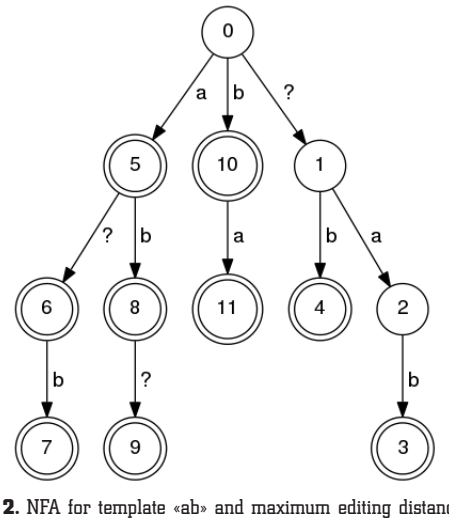


Fig. 2. NFA for template «ab» and maximum editing distance 1

Since NFA simulation is a costly process, the next step is to determinize it to ensure fast word verification.

The goal of determinism is that each state of the automaton has only one transition for each symbol [5]. To achieve this, let's combine for each state a universal transition with its other transitions. Thus, let's avoid the need to go to several states at the same time. After determinization, the result of NFA can be seen in Fig. 3, where the symbol «?» is responsible for all symbols except those for which there are already transitions from this state. For example, for state 0, the character «?» is responsible for all characters except «a» and «b».

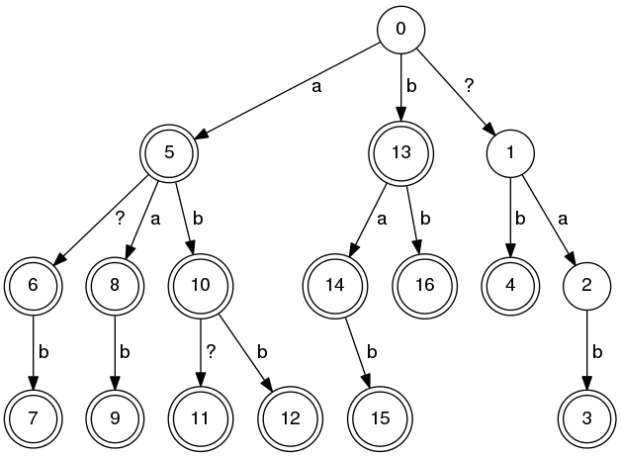


Fig. 3. DFA for template «ab» and maximum editing distance 1

Automatic word checking is very simple. In the cycle, let's go through each character of the input word in parallel, updating the current state in which we are. If a nonexistent transition is hit, the iteration terminates because the input word has an editing distance to the template greater than the maximum allowed.

2.3. HashAutomaton. In this solution, there is no binding to the tree structure. Let the cost of each operation, be it deletion, insertion, transposition, or replacement, be the same and equal to one, which allows building a more structured automaton that speeds up the construction and determinization of the NFA. Each state of the automaton corresponds to a certain configuration of the number of processed symbols of the template and the number of editing operations applied at the same time. Each transition between states corresponds to a specific operation. States that have completely processed the template are final. Fig. 4 shows the NFA for the pattern «ab» and a maximum edit distance of 1.

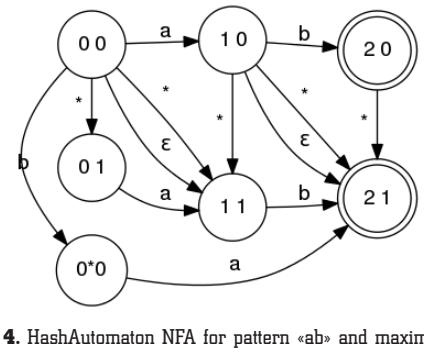


Fig. 4. HashAutomaton NFA for pattern «ab» and maximum edit distance of 1

The first number in the state name corresponds to the number of processed characters of the pattern, and the second number corresponds to the edit distance. «*» denotes transitions accepting any symbol, «ε» denotes zero transitions, the initial state is «0 0» [7].

Next, it is necessary to build a DFA, which is much more convenient and effective for the word verification process. To construct a DFA, it is necessary to go through all transitions of the NFA, creating new states in the DFA for each unique combination of states of the NFA. After the NFA determinism is complete, the automaton is ready to check words. Word verification is quite similar to the prefix tree-based automaton verification discussed earlier.

2.4. TableAutomaton. Based on the automaton described in [8], let's modify the given implementation to support unicode character transposition operations. The NFA underlying TableAutomaton is no different from the NFA in HashAutomaton. The main advantage of TableAutomaton is the absence of the need to explicitly construct the automaton. Having a template and the maximum editing distance, the machine can immediately start checking words. The main observation underlying TableAutomaton is that the result of deterministic NFA in HashAutomaton for words like «free» and «tree» will be the same, but will be different from DFA for the word «pain» or «soon». If to rewrite the words, giving each unique symbol its own number in order, the connection between these words will be obvious. Let: free=1233=tree, pain=1234, soon=1223. That is, the set of unique NFA states that can be obtained from one state depends only on the currently checked symbol and occurrences of this symbol in the template, starting from the shift at which this state is located.

To reflect this fact, the concept of a characteristic vector is introduced, which is a bit mask, where the value in a position is equal to 1 only when the symbol of the template at that position is equal to the symbol being checked. However, only characteristic vectors of length $2 \cdot d+1$, where d is the maximum editing distance, are important for the transition between states. Given this, it is necessary to pre-calculate all possible transitions and all possible states for any pattern at the beginning of the program. The algorithm is similar to NFA determinism, but instead of template symbols, characteristic vectors should be used and the resulting unique state configurations should be saved in the transition table.

It is worth noting that this approach is practical only for small values of the maximum editing distance, since the size of the table grows exponentially with its value, since there are unique values of the characteristic vector, where d is the maximum distance [7].

When checking a word for each symbol, a characteristic vector is calculated for a given template, where, depending on its value and the current state, the next state of the automaton is selected from the table of all transitions. At the end of all incoming characters, it is necessary to check whether the current state is final.

3. Results and Discussion

As part of the study, all 4 algorithms were tested for correctness of work, and performance tests for various input data were developed and analyzed.

To check the correctness, a program was developed that would use a dictionary of words and compare the edit

ISSN 2664-9969

distance calculated by one current solution to the edit distance calculated using a ready library Damerau-Levenstein algorithm. The principle of the test: a dictionary of all possible words in English is read and the test is started for each of the solutions. Within the test, the editing distance, pattern, and words to be tested are randomly selected. The editing distance is then checked using the library algorithm and automata-based solutions. All 4 algorithms successfully passed the test on the correctness of work on a dictionary containing 370 thousand English words.

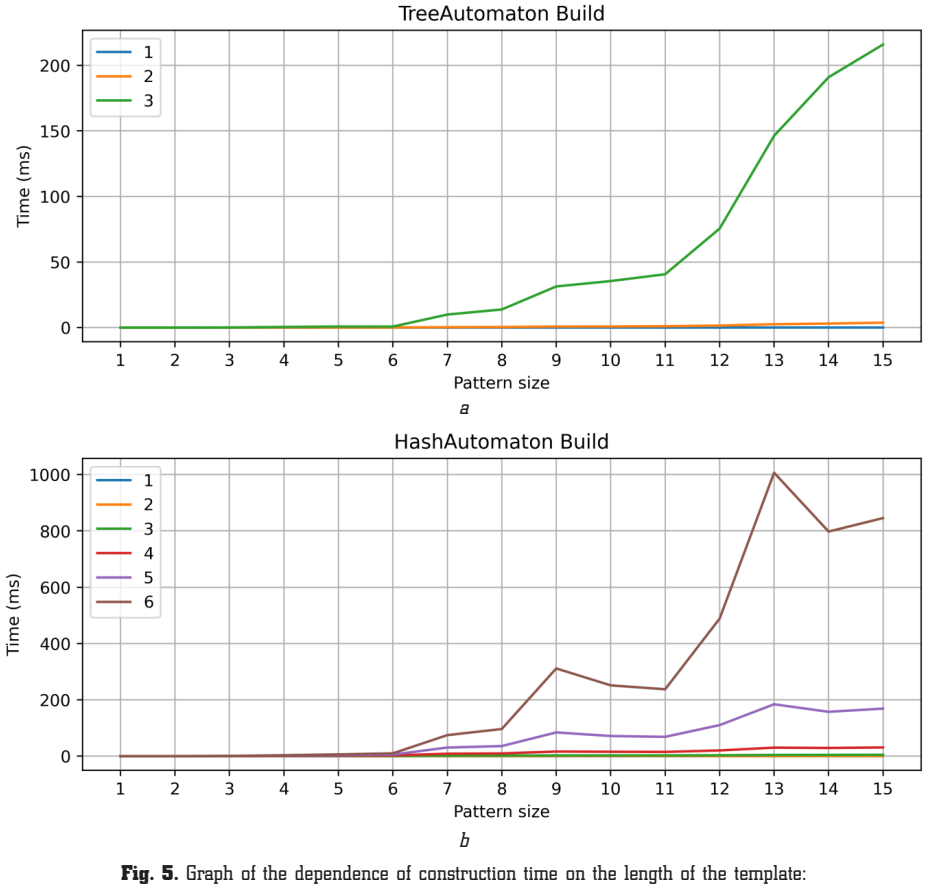
To check the performance, tests were carried out, which determine the time of building the automaton and the time of checking the word for each of the solutions. It is necessary to calculate the time for two variants of checks, matches and non-matches, that is, words that are not accepted by machines. In addition to the time spent, the amount of memory used is an important criterion for evaluating the algorithm [9]. Therefore, it is necessary to measure the amount of RAM occupied by each machine. The google::benchmark framework was used to conduct and create tests.

Construction time and memory measurements were made only for TreeAutomaton and HashAutomaton, since the TableAutomaton transition table is already written in the program. In Fig. 5 shows graphs showing the dependence of the construction time on the length of the template for different values of the maximum editing distance. For TreeAutomaton, it was only possible to construct automata for distances up to 3, since larger distances take significant construction time and consume a lot of memory.

Fig. 6 shows graphs that reflect the dependence of the maximum amount of used memory on the length of the pattern. As the maximum editing distance increases, the required amount of memory increases. Time and memory for TreeAutomaton grow very quickly. Let's note that larger templates and editing distances require more calculations and consume more memory. However, HashAutomaton proved to be more efficient in terms of construction time and memory usage compared to TreeAutomaton. This is due to the basic complexity of their automaton structures and the way they are implemented.

As the pattern size increases, the time also tends to increase. The time for TreeAutomaton and TableAutomaton is significantly affected by the maximum editing distance. In the first case, this is due to the use of a data structure of an associative container based on trees, which asymptotically has a logarithmic search time, but with a small number of elements it works faster than a similar container in the C++ language based on hashing [10]. In the second case, this is due to the need to calculate the characteristic vector for each character of the line. This is due to the fact that a characteristic vector must be calculated for each character to verify a word. To speed up this process, it is possible to use SIMD-based optimization of processor instructions.

The result of the general test is shown in Fig. 7. This test was conducted on a dictionary consisting of 370,000 English words. Each of the solutions works faster than a simple approach using the Damerau-Levenshtein algorithm. The lack of explicit automaton construction does not provide a significant advantage to TableAutomaton, since the construction of automata occurs only once at the beginning.



a

b

Fig. 5. Graph of the dependence of construction time on the length of the template: *a* – TreeAutomaton; *b* – HashAutomaton

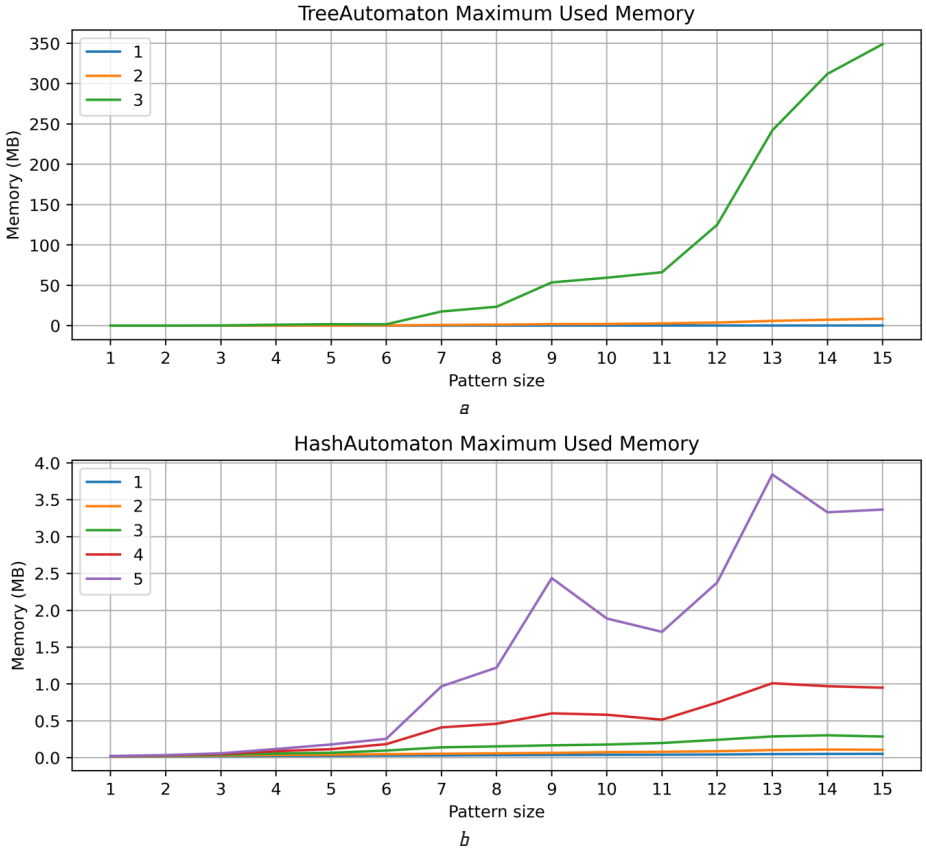


Fig. 6. Graph of the dependence of the maximum amount of used memory on the length of the template: *a* – TreeAutomaton; *b* – HashAutomaton

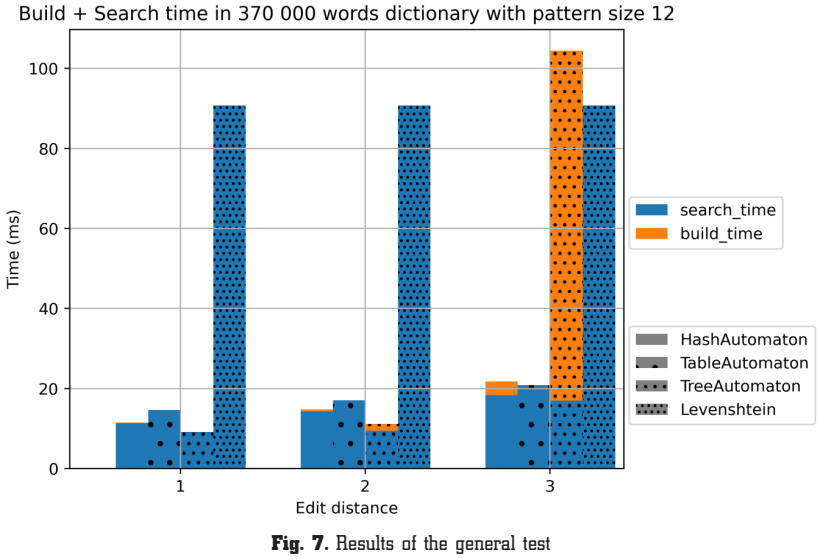


Fig. 7. Results of the general test

As a result of the study, recommendations were made for the development of a fuzzy search system.

With small permissible editing distances of 1–2, it is possible to use TreeAutomaton. TableAutomaton is best for medium edit distances of 3–4, and HashAutomaton for long search terms.

It is worth noting that the construction of the automaton makes sense when the size of the text data is more than 5000 words, otherwise it is better to use the simple Damerau-Levenshtein algorithm.

The conditions of the martial law in Ukraine did not affect the obtained results in any way, because the execution time of the algorithm depends only on the number of operations that it needs to perform.

However, the war in Ukraine has affected the search queries that users make more often and the information that is stored in cloud data stores. There was much more information related to the situation at the front.

Other modifications of fuzzy search algorithms are planned in the future. For example, it is possible to add the use of character similarity tables so that characters that are next to each other on the keyboard, or that are semantically similar, are more similar and appear higher in search results than unrelated characters.

4. Conclusions

In this paper, a comparison of fuzzy search algorithms based on the Damerau-Levenshtein distance was made. The usual Damerau-Levenshtein algorithm and 3 algorithms based on finite state machines were compared. In Section 2, several software implementation options were developed and described, including TreeAutomaton, HashAutomaton, and TableAutomaton. Tests were conducted to check the correctness of the implementation and to

ISSN 2664-9969

evaluate the performance, which includes: the construction of the automaton, word verification, and general tests that check the speed of the automata in comparison with the trivial approach.

Based on the conducted tests and analysis of solutions, it can be concluded that all developed solutions based on finite state machines work faster than a simple approach using the Damerau-Levenshtein algorithm for searching text data of more than 5000 words. This is explained by the fact that additional time is spent on building automata, while the usual algorithm immediately starts searching. On a dictionary with 370,000 words and maximum editing distances from 1 to 3, each machine showed results 5–9 times faster.

HashAutomaton is recommended for general use because the amount of memory used is significantly less for large edit distance values, and the speed of word validation is not significantly different from TreeAutomaton. This machine turned out to be the most versatile.

TreeAutomaton is recommended for situations where you need a different cost for edit operations, or for an edit distance between 1 and 2. At small edit distance values, it shows the fastest performance.

TableAutomaton is recommended for situations where the search term changes frequently, as it does not need to be constructed and determinized for any pattern and small edit distance values. The main speed limitation of this machine is the need to calculate the characteristic vector for each input symbol, which can be accelerated by 5–10 % by using SIMD instructions for its calculation. However, this solution is not universal, as it requires the use of a central processor with support for such instructions.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this study, including financial, personal, authorship, or any other, that could affect the study and its results presented in this article.

Financing

The study was conducted without financial support.

Data availability

The manuscript has no associated data.

References

1. Navarro, G. (2001). A guided tour to approximate string matching. *ACM Computing Surveys, 33 (1)*, 31–88. doi: https://doi.org/10.1145/375360.375365
2. Schulz, K. U., Mihov, S. (2002). Fast string correction with Levenshtein automata. *International Journal on Document Analysis and Recognition, 5 (1)*, 67–85. doi: https://doi.org/10.1007/s10032-002-0082-8
3. Boytsov, L. (2011). Indexing methods for approximate dictionary searching. *ACM Journal of Experimental Algorithmics, 16*. doi: https://doi.org/10.1145/1963190.1963191
4. *Damerau–Levenshtein distance*. Available at: https://www.geeksforgeeks.org/damerau-levenshtein-distance/
5. Snášel, V., Keprt, A., Abraham, A., Hassanien, A. E. (2009). Approximate String Matching by Fuzzy Automata. *Man-Machine Interactions*. Berlin, Heidelberg: Springer, 281–290. doi: https://doi.org/10.1007/978-3-642-00563-3_29
6. Baeza-Yates, R., Navarro, G.; Hirschberg, D., Myers, G. (Eds.) (1996). A faster algorithm for approximate string matching. *Combinatorial Pattern Matching. CPM 1996. Lecture Notes in Computer Science. Vol 1075*. Berlin, Heidelberg: Springer. doi: https://doi.org/10.1007/3-540-61258-0_1
7. Girijamma, H. A., Ramaswamy, H. A. V. (2009). An extension of Myhill Nerode Theorem for Fuzzy Automata. *Advances in Fuzzy Mathematics, 4 (1)*, 41–47.
8. Ramon Garitagoitia, J., Gonzalez de Mendivil, J. R., Echanobe, J., Javier Astrain, J., Farina, F. (2003). Deformed fuzzy automata for correcting imperfect strings of fuzzy symbols. *IEEE Transactions on Fuzzy Systems, 11 (3)*, 299–310. doi: https://doi.org/10.1109/tfuzz.2003.812682
9. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2022). *Introduction to Algorithms*. MIT Press, 1312.
10. Mihov, S., Schulz, K. U. (2004). Fast Approximate Search in Large Dictionaries. *Computational Linguistics, 30 (4)*, 451–477. doi: https://doi.org/10.1162/0891201042544938

✉***Kyrylo Kleshch,*** *Assistant, Postgraduate Student, Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, ORCID: https://orcid.org/0009-0006-8133-3086, e-mail: kleshch.kirill@gmail.com*

Volodymyr Shablii, *Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, ORCID: https://orcid.org/0009-0004-5113-3572*

✉*Corresponding author*