The Multi-Pattern Matching with Online User Requests for Determining Browser Capabilities

Ignat Blazhko[a], Tetiana Kovaliuk[b], Nataliya Kobets[c] and Tamara Tielysheva[a]

[a] National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 37, ave. Peremohy, Kyiv, 03056, Ukraine
[b] Taras Shevchenko National University of Kyiv, 64/13, Volodymyrska str., Kyiv, 01601, Ukraine
[c] Borys Grinchenko Kyiv University, 18/2 Bulvarno-Kudriavska str., Kyiv, 04053, Ukraine

Abstract
One of the most effective ways to get information about Internet users is to analyze User-Agent request header, because it contains information about the user's browser. Quick methods are needed for multi-pattern matching to instances of User Agents. The article considers the algorithm of patterns matching with input strings to determine the capabilities of the browser. The input strings are the headers of User-Agent queries, and strings with special wildcards represent search patterns. The developments in the field of solving the problem of comparing search templates with input lines are considered and analyzed. A new algorithm for solving the problem is proposed. The steps of the algorithm are given and the time complexity of the algorithm for the problem is analyzed. Comparisons of the performance of the received program with analogues are made and the advantage of the offered approach is shown. The program implementation was compared with the most prominent analogues and the advantage of the proposed approach has been shown in terms of execution speed.

Keywords 1
Pattern matching, wildcards, browser capabilities, User Agent request

1. Introduction

Nowadays, when many companies operate on the Internet, the need to identify user information is very acute. One of the most effective ways to obtain such information is to analyze the User-Agent request HTTP header, which identifies the program making a request to the server and requesting access to the website. The User-Agent request header contains specific information about the hardware and software of the device making the request. This information typically includes information about the browser, web visualization mechanism, operating system, and user's device, including iPhone, iPad or other mobile device, tablets, desktop computers. Web servers use User-Agent to maintain different web pages in different browsers in case of not adaptive web design, display different content for different operating systems, collect statistics that reflect the browsers and operating systems used by visitors. Web crawling bots also use User-Agent.

The data specified in the User-Agent request HTTP header [1] describes browsers (Chrome, Firefox, Internet Explorer, Safari, BlackBerry, Opera), search engines (Google, Google Images, Yahoo), game consoles (PlayStation 3, PlayStation Portable, Wii, Nintendo Wii U), offline browsers (Offline Explorer), links (Link Checker, W3C-check link), electronic readers (Amazon Kindle), validators, cloud platforms, e-mail libraries, scripts. By User-Agent, you can define functions that are supported by a web browser, such as JavaScript, Java Applet, cookie, VBScript, and Microsoft ActiveX.

COLINS-2021: 5th International Conference on Computational Linguistics and Intelligent Systems, April 22–23, 2021, Kharkiv, Ukraine
EMAIL: iob.veritas@gmail.com (I. Blazhko); tetyana.kovalyuk@gmail.com (T. Kovaliuk); nmkobets@gmail.com (N. Kobets), telyshevatamara@gmail.com (T. Tielysheva)
ORCID: 0000-0002-1383-1589 (T. Kovaliuk); 0000-0003-4266-9741 (N. Kobets); 0000-0001-5254-3371 (T. Tielysheva)

© 2021 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-2870/paper75.pdf

The User-Agent string is used to match the content when the source server selects the appropriate content or operating parameters to respond to the received user request. Thus, the concept of adapting content to the needs of users is realized. The User-Agent string is one of the criteria by which search engines can be excluded from accessing certain parts of the website using the robot exclusion standard (according to the robots.txt file). Information about users has obvious applications in the advertising sector, namely in cases where the user's device can operate as a criterion for determining the user's affiliation to the target market. Web analysis is an equally important use case. Due to the availability of information about customers, further analysis can significantly improve the decision about the published content; increase the conversion of the site by turning site visitors into real customers. Analyzing User-Agent request headers that are constantly updated also means that businesses will be aware of the emergence of new devices that visit their resource, and therefore related problems will be identified at an early stage. However, parsing the User-Agent request header is not a trivial task. Many software libraries are used to solve this problem by balancing between accuracy and speed of obtaining results. There are services that provide definition and analysis of the User-Agent request header, in particular, deviceatlas.com. The accuracy problem can be solved by using open lists of User-Agent templates, which are constantly updated. There are more than 62 million User-Agent strings [1], so existing solutions cannot quickly work with all of them. The speed of response to the client is another important indicator for business. Therefore, quick methods of analyzing these patterns are needed. These are methods for matching a large number of search patterns to User-Agent instances. The development of algorithms based on which libraries will be created to determine the capabilities of browsers is an urgent and necessary task for any web service.

The purpose of the research is to develop an algorithm for multi-pattern matching with wildcards to be able to detect browser capabilities based on browscap.org data [2]. The library should work faster than its alternatives – the speed should be comparable to libraries that use simplified (inaccurate) methods to detect browser capabilities. Browscap is an open project dedicated to collecting and disseminating a database of browser capabilities, actually a set of different files describing browser capabilities. Browscap has built-in support for using these files.

To achieve this goal we need to solve the following tasks:

- figure out browscap.org data structure and similarities between different user agents of the same family;
- review and analyze existing studies that solve the problem of multi-pattern matching with wildcards
- develop an algorithm for multi-pattern matching with the best known lower bound time complexity for determining the browser capabilities;
- develop software implementation of the algorithm and compare the speed of the proposed algorithm with analogue libraries, which give approximate results.

2. Problem Statement

Let the set of search patterns $P = \{p_1, p_2, ..., p_m\}$ be input to the search engine. Each search pattern $p_i, i = \overline{1, m}$ is a string of characters that belong to the alphabet Σ. Each pattern contains special characters "?" and "*", that do not belong to this alphabet, and are called wildcards. The character "?" corresponds to any single character in the alphabet Σ, the character "*" corresponds to any string of characters in the alphabet Σ, including the empty string. The search pattern $p_i, i = \overline{1, m}$ can be represented as a string $p_i = c_{i1} k_{i1} c_{i2} k_{i2} ... c_{ij} k_{ij} *$, where c_{ij} is the j-th wildcard character for the i-th pattern, k_{ij} is a keyword from a dictionary of unique keywords W, that is, a string of characters in the alphabet Σ. There is an additional condition for the wildcard character c_{i1}: it can be either skipped or equal to the character "*".

Input requests are in the form of a string $S = \{s_i\}, i = \overline{1, n}$, consisting of n characters in the alphabet Σ. It is necessary for each input string $S = \{s_i\}, i = \overline{1, n}$ to find all the search patterns from the set P

that matches to the query string, i.e. get a set of patterns $A = \{p_{a_1}, p_{a_2}, ..., p_{a_{anslen}}\}$, where p_{a_S} denotes the search pattern for the S-th request. The number of requests is not limited. Requests must be answered as they are received.

One of the features of this task is that the set of search patterns is very large and is in the millions of copies. Query string sizes typically range from 100 to 200 characters.

3. Related Works

The main tasks of text string analysis that arise in the development of information verification tools in information retrieval systems include:

- the task of matching text strings;
- the problem of calculating the distance between text strings;
- the problem of fuzzy match text strings;
- the task of finding the longest repeating text substring.

The algorithms for text string analysis are presented in the works of domestic and foreign researchers, in particular Stephen G.A., Knuth D.E., Morris J.H., Pratt V.R., Karp R.M., Rabin, M.O., Boyer R.S., Moore J.S., Fischer M.J., Hirschberg D.S., Hunt J.W., Szymanski T.G., Landau G.M., Vishkin U. Hamming R.W., Levenshtein V.I.

The task of determining the browser capabilities based on the analysis of the User-Agent HTTP header is reduced to the task of matching text strings, which are used as search patterns and incoming requests. Algorithms for matching search patterns with input strings have many important applications and are used in antivirus software, systems for detecting various types of attacks and intrusion prevention [3], in-text compression, text search, data mining, programming grammar checking rules [4] etc. For example, different types of attacks can be defined using rules, which are regular expressions that match a set of possible strings. Algorithms for working with regular expressions require pre-processing which is very memory-intensive and time-consuming. An example of the application of the algorithm for matching search patterns with input strings is the study of DNA sequences, where wildcards are used as a replacement for some components of protein sequences.

Research on the problem of comparing search patterns with text can be classified in the following areas:

- the case of one pattern and one string;
- the case of patterns with a fixed number of wildcards or additional restrictions on the number of patterns;
- the case of patterns with the possibility of a flexible task of wildcards, for example, the task of the minimum and maximum break length.

In the general case, the task of matching text strings requires localizing all occurrences of a pattern in the text. In paper [5] the widely used multiple string patterns matching algorithms have been analyzed and discussed. The main algorithms for matching text strings include the Knuth-Morris-Pratt [6], Rabin-Karp [7] algorithms, the Boyer-Moore algorithm [8] and its variations. One of the most efficient algorithms for finding substrings in a string is the Aho-Corasick algorithm [9], which groups all strings from a dictionary into a prefix tree and then converts it into a finite state machine in which the search is performed. The operating time is proportional $O(m+n+k)$, where n is the length of the pattern string, m is the total length of the dictionary strings, k is the length of the answer string, that is, the total length of occurrences of words from the dictionary into the pattern string. The Rabin-Karp search algorithm uses hashing to find words from the dictionary in the text. For text length n and m strings with length $|P|$, execution time is $O(n+|P|)$ a memory cost $O(m)$.

The Commentz-Walter algorithm [10] is used to search for multiple patterns at text, which combines the ideas of Boyer-Moore algorithms and Aho-Corasick. The time complexity of the search stage is $O(nl_{\max})$, where n is the length of the string, $l_{\max}$ is the length of the longest string from the dictionary. The Wu-Manber string-matching algorithm [11], based on the idea of the Boyer-Moore algorithm. The Wu-Manber algorithm is considered the fastest multi-pattern string-matching algorithm in practice. The main difference is that the algorithm does not consider individual characters, but blocks

of characters of a given length. The disadvantage of this algorithm is the slowing down of its speed with the increasing number of strings in the dictionary.

The Fischer-Paterson algorithm [12] for the string matching is based on the Fast Fourier Transform (FFT) [13] and has a time estimate of complexity $O(n \log m \log \sigma)$, where σ is the size of the alphabet Σ. Indyk's randomized algorithm [14] uses FFT to calculate the convolution between the pattern and text with the time estimate $O(n \log n)$. In [15], the optimal algorithm with time complexity $O(n + m + occ)$ is proposed, where occ is the number of pattern' occurrences in the text. Algorithms by K. Barton and K. Iliopoulos [16] use wildcards either only in the text or only in the patterns. The time complexity of the algorithms is defined as $O\left(\frac{n \log_\sigma m}{m}\right)$ and $O\left(\frac{n(g + \log_\sigma m)}{m}\right)$, respectively, where g is the number of wildcards in the pattern.

There are matching algorithms, which use bit parallelism to speed up the algorithm for text search patterns. Operations on the bits of the same machine word are performed in parallel. An example of using bit parallelism in search algorithms is the bitap algorithm (also known as the Shift-Or, Shift-And or Baeza-Yates–Gonnet algorithm) [17]. This algorithm is an approximate string-matching algorithm. The approximate equality of a substring to a given pattern is determined in terms of the Levenshtein distance. In [18], an algorithm for matching a pattern with a text, based on a directed acyclic word graph (DAWG) is described. A directed acyclic graph represents the suffixes of a given string in which each edge is labelled with a character. The characters along a path from the root to a node are the substring, which the node represents. An example of algorithms with several search patterns that have a fixed number of wildcard characters is an algorithm based on the Hamming distance [19] between bit vectors.

In [20-22] algorithms have proposed that use a finite automaton according to the Aho-Corasick algorithm and dynamic marking of ancestor nodes. The time complexity is $O((|P| + |t|)\frac{\log(k)}{\log\log(k)})$. The limitations of these algorithms are using one copy of the text, only wildcards of the form "*", and costly operations of construction and modification of automata. Algorithm [23] works with wildcards that specify a range of characters. It is based on the suffix tree. The estimation of the time complexity of this algorithm is $O\left(n + m\frac{n}{|\Sigma|}\sum_{i=0}^{l-1} Gi\right)$, where n is the length of the input text, m is the number of search patterns, $|\Sigma|$ is the number of characters in the alphabet Σ, G_i is the average length of the i-th range in the search pattern. The time complexity of this algorithm remains quite large and depends on the specified ranges of wildcards.

Thus, the existing methods have a number of disadvantages, with a large number of search patterns, they work very slowly, poorly scalable for large templates and texts, which makes them unsuitable for solving the problem.

4. Algorithm for matching search patterns with incoming query strings

4.1. Description of the method for solving the problem

To solve the problem of matching search patterns with input strings, an algorithm based on the prefix tree [24] as a data structure and a finite automaton built on a modified Aho-Corasick algorithm is proposed. The prefix tree is built based on input patterns according to the compressed prefix tree [25]. Each pattern is represented as a string of keywords and special characters "*" and "?" between them. Each edge in the prefix tree contains information about the transition keyword and a special character that precedes it. The tree is built only once when obtaining search patterns and is unchanged during the further operation of the algorithm. To reduce the memory consumption, each keyword in the set W of all keywords in the set P of the patterns is assigned a character from the alphabet Δ, i.e. there is a bijective mapping such that $f : \delta_j \to w_j$; $g : w_j \to \delta_j$.

An automaton built according to the Aho-Corasick algorithm is used to quickly search for keywords in the text of incoming requests. Terminal links are added to the automaton to speed up the search for keywords that are prefixes to other keywords. This allows you to find all keywords in the text S within

a linear time of the number of the keywords in the text. Links to nodes of the prefix tree will be added to the final states of the automaton, which may correspond to the beginning of some search pattern. This will cut off options that are not possible in advance. When a new string S is received, it is scanned from left to right and at the same time, a set of prefixes of possible search patterns is constructed, using information about transitions in the prefix tree.

4.2. Description of the data structure

A prefix tree (trie) is a data structure in the form of a tree, in which the path from the root to leaf defines a string. The strings with the same prefixes have a common path from the root with the length of that prefix. The prefix tree allows you to store an associative array, the strings of which are the keys. Unlike binary trees, the key is not stored in the tree leaf. The key value can be obtained by looking at all parent nodes, each of which stores one or more alphabet characters. The root of the tree is associated with an empty string. An internal node will contain a key if that key is a prefix of some other key in that tree. The prefix tree can be compressed to optimize memory usage. Nodes that have only one child are compressed. In this case, the transitions can contain not one character, but a whole string that corresponds to the transition.

Since prefix trees are built in such a way that all keys with a common prefix have common nodes that correspond to this prefix, this makes it possible to search without having a full key, but only its prefix. The result will be a set of possible keys that are stored in the tree.

4.3. Practical implementation options

The main difference between the implementations is the way of storing transitions between nodes. The chosen method depends on what needs to be optimized: runtime, memory consumption, or something else. For example, for a small alphabet, it may be advisable to store each node in an array in which the transitions for each of the characters in the alphabet will be recorded. This will make the access time to the next node as small as possible $O(1)$, but will increase memory consumption since not all characters of the alphabet will contain transitions. You can also use a red-black search tree [26], which will not increase memory consumption and will give a logarithmic access time $O(\log x)$ to the next node. Another alternative is the use of a hash table, which makes it possible to perform an operation of accessing an element in time $O(1)$ with an asymptotic estimate of the total memory cost with an ordinary array. Although it is possible that the search operation will be performed on $O(x)$, which is influenced by the fullness of the table and the choice of hashing methods.

4.4. Construction of the automaton based on the Aho-Corasick algorithm

The Aho-Korasik algorithm is a string search algorithm that allows you to find the elements of a certain dictionary in an input string. The algorithm finds a match as it reads the input string and returns matches found. The algorithm does not need to know the whole string in advance.

At the beginning of the algorithm, a finite state automaton is constructed based on strings from the dictionary. This automaton is a prefix tree with additional links, which are called suffix links. Such links indicate the longest suffix of the current state-string that exists in this automaton. The link indicates the initial state (root) in the absence of such suffix.

The behaviour of the automaton can be described by three functions: the transition function, the failure function, the output function. The transition function for each next input character of the line indicates in what state the transition needs to be made or informs that there are no such transitions. The failure function is used if there are no transitions. This function works as follows: while there is no transition from the state to the transition function, we make transitions by suffix links. If necessary, these transitions with suffix links are repeated until the initial state is reached. The output function allows you to check the correspondence of each state of the automaton to a certain string in the dictionary and returns it as a result.

Additional links (final or valid) can be added to speed up the search for dictionary strings in the input string. These links for each state of the automaton will point to another state, the string of which is the longest suffix of the current state-string that matching to some string in the dictionary. Suffix references can be pre-calculated during the construction of the automaton. This will allow you to find quickly all the words in the dictionary that are in the current input string and end with the given character while processing the next character. If the dictionary is known in advance, the construction of the automaton can be performed only once. The constructed automaton can be reused by transition to the initial state for each new input string.

4.5. Description of algorithm steps

The algorithm can be divided into two parts: the algorithm for pre-processing search patterns and the algorithm for processing incoming query strings. Consider the steps of the proposed algorithm.

Step 1. Let's build a set of all unique keywords W. For each i-th search pattern p_i we add k_{ij} to the set W: $W = \{w_j \mid j \in J\} = \bigcup_{i=1}^{m} \{k_{ij} \mid j = \overline{1, l_i}\}$.

Step 2. Each keyword $w_j \in W$ is matched with a number δ_j to obtain a bijective reflection: $f : \delta_j \to w_j ; g : w_j \to \delta_j$.

Step 3. Let's build a set B of converted search patterns, by replacing keywords in p_i with characters of the alphabet $\Delta : |\Delta| = |W|$ and removing the character c_{i1} in case of its presence: $\mathrm{B} = \{b_i = g(k_{i1}) c_{i2} ... g(k_{il_i}) \mid i = \overline{1, m}\}$.

Step 4. Based on the set B, we will construct a prefix tree T as follows. Each prefix $g(k_{ij})$ will be considered as an ordinary symbol c_{ij} of the alphabet Δ, if it is equal to "*" or absent. In this case, we will add an edge with a symbol to this prefix tree. Since the size of the alphabet can be quite large, in each of the nodes we will store a hash table with a list of edges, which are symbols $g(k_{ij})$, as a search key. This will allow you to quickly make transitions along specific edges, which are symbols of the alphabet Δ. If symbol c_{ij} equal to "?", we will additionally notice such edges to distinguish them from ordinary edges.

Step 5. Let's build a finite automaton according to the Aho-Corasick algorithm based on keywords from W. For each state that corresponds to a certain keyword w_j, we will additionally save number δ_j. Transitions for each of the states by alphabetic characters Σ will be stored in the hash table.

Step 5.1. Moving along the suffixed links, for all states of the automaton, we will preliminarily calculate the "final" links *termLink* to the nearest of the states, from which it is possible to get to the state corresponding to a certain keyword from W.

Step 5.2. For states of the automaton matching to a keyword w_j such that it is present in any search pattern at the first position and does not have a wildcard character that precedes it, we will store a link *headLink* to the corresponding node in the prefix tree T, which can be found by moving from the root of T along the edge $g(w_j)$.

Step 5.3. For states of the automaton matching to a keyword w_j such that it is present in any search pattern at the first position and has a wildcard character "*" that precedes it, we will additionally store a link *looseHeadLink* to the corresponding node in the prefix tree T, which can be reached by moving from the root T along the edge $g(w_j)$.

After performing pre-processing of the search patterns (the first five steps of the algorithm), the algorithm can accept text analysis requests – input string S. For each individual string, you need to do the following:

Step 6. Create an empty array q in which to store a pair of values: the index of string S, and the links to the trie T node. At the beginning of the operation, the state of the automaton corresponds to the root node $curState = rootState$.

Step 7. Scan the string S from left to right. For the current character $s_{idx}, idx = \overline{1,n}$, do the following:

Step 7.1. Let's move from the current state of the automaton to the new one by symbol s_{idx}, using the transition function in the Aho-Corasick algorithm, and update $curState$.

Step 7.2. If the current state of the automaton matches some keyword w_j and $|w_j| = idx$, then add a pair of values $(idx, curState.headLink)$ to the array q.

Step 7.3. If the current state has a reference to $looseHeadLink$, then add a pair $(idx, curState.looseHeadLink)$ to the array q.

Step 7.4. If the current state of the automaton matches some keyword w_j and $|w_j| \neq idx$, then $lb = idx - |w_j|$. For all pairs from an array q such that $q_i.idx \leq lb$ it is necessary to check up, whether the transition from a tree T node along an edge $g(w_j)$.

Step 7.4.1. If the transition exists and the edge is not marked with the symbol "?", make the transition on it, adding a pair idx and a node to the array q.

Step 7.4.2. If the transition exists and the edge is marked with the symbol "?", we will additionally check that $q_i.idx = lb - 1$ and add the pair idx and the node to which we have passed, to the array q.

Step 7.4.3. If a new pair of values has been added to q, check that the tree node in that pair matches some search pattern. If so, we will add this search template to the answer A.

Step 7.5. If the current state has a link $termLink$, go to this link and return to step 7.3.

Step 8. Return the set of found search patterns A and finish the algorithm.

As you can see, in the process of the algorithm, we move all possible paths in the prefix tree T, and this ensures that all search patterns are found in the string S, because they are all in T.

5. Test case

Let there be an alphabet $\Sigma = \{c_1, c_2, c_3, c_4, c_5\}$. The input is set $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ search patterns. Patterns are as follows: $p_1 = *c_1 * c_1c_3$; $p_2 = *c_1$; $p_3 = c_1c_2 * c_4c_2c_5$; $p_4 = c_2c_5 * c_1c_3 * c_4c_2c_5$; $p_5 = c_1c_2 * c_2c_5 * c_4c_2c_5$; $p_6 = c_2c_5 * c_1 * c_2c_5$; $p_7 = c_2c_5 * c_4c_2c_5$.

Select keywords from search patterns and get the set $W = \{w_1, w_2, w_3, w_4, w_5\}$ of keywords as: $w_1 = c_1$; $w_2 = c_1c_2$; $w_3 = c_1c_3$; $w_4 = c_2c_5$; $w_5 = c_4c_2c_5$.

Let's build a set $\mathrm{B} = \{b_1, b_2, b_3, b_4, b_5, b_6, b_7\}$ of converted search patterns based on keywords. The converted search patterns are of the form: $b_1 = *w_1 * w_3$; $b_2 = *w_1$; $b_3 = w_2 * w_5$; $b_4 = w_4 * w_3 * w_5$; $b_5 = w_2 * w_4 * w_5$; $b_6 = w_4 * w_1 * w_4$; $b_7 = w_4 * w_5$.

We construct a prefix tree T, using the set $\mathrm{B} = \{b_1, b_2, b_3, b_4, b_5, b_6, b_7\}$ (figure 1).

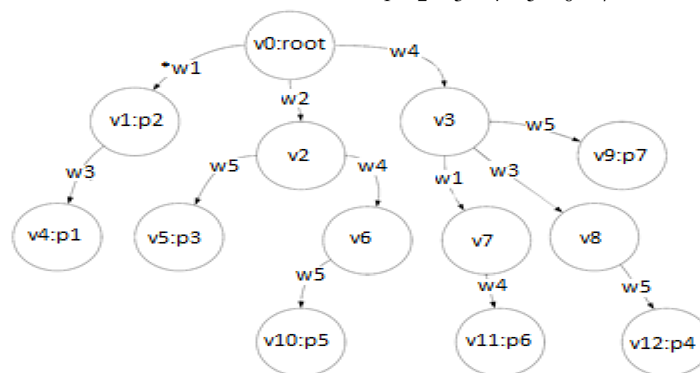

Figure1: Constructed prefix tree T

Using a set of keywords $W = \{w_1, w_2, w_3, w_4, w_5\}$ and the prefix tree T, we construct a finite automaton according to the modified Aho-Corasick algorithm (figure 2). To simplify the image of the automaton, suffix links that lead to the initial state are not shown in the figure.

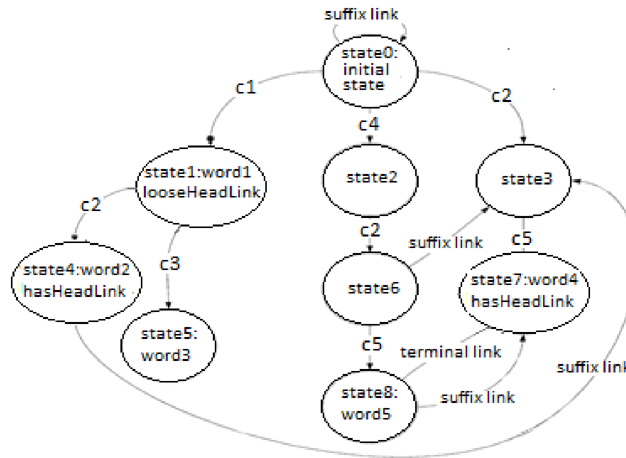

Figure 2: Constructed automaton according to the modified Aho-Corasick algorithm

Then comes the input string $S = c_2c_5c_5c_5c_1c_2c_4c_3c_3c_4c_2c_5c_2c_1c_3c_2c_5$. Let's start moving from left to right along the string S. Initially, the array q is empty. The elements added to q can be considered an implicit construction of the subtree of the prefix tree T, which for each subsequent character $s_i \in S, i = \overline{1,m}$ contains all the prefixes of the tree T.

When reading the second character from the input string, we go into the state $state7$ of the automaton, which corresponds to the keyword w_4. This keyword w_4 is labelled $hasHeadLink$, which means it can be the first keyword in the search pattern and must be at the beginning of the string S. Since the link to the corresponding node in the prefix tree T was saved in the automaton, let's add the node v_3 to the array q: $q = \{(2, v_3)\}$.

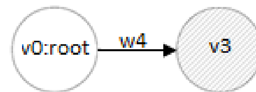

Figure 3: Prefix tree after adding nodes to the array q after processing the second character

Consider the prefix tree after processing the fifth character. The automaton is in a state $state1$ that matches the keyword w_1. This state has a final link $terminalLink$, the keyword has a label $looseHeadLink$, and therefore can be the beginning of a search pattern in any part of the input string S. We will add a node v_1 that matches this initial keyword to the array q. As we see v_1 corresponds to the full search pattern p_2. We will add it to the answer. We will also check the presence of transitions from the nodes added to the array q by the keyword w_1. Since there is a transition from the node v_3 to the node v_7, add these nodes to array q (Figure 4) and have the array in the form $q = \{(2, v_3), (5, v_1), (5, v_7)\}$.

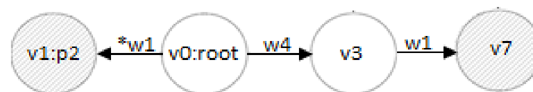

Figure 4: Prefix tree after adding nodes to an array after processing the fifth character

When processing the last character of the string S, the automaton is in a state $state8$ that matches to the keyword w_5. Among the elements of the array q is a node v_8, from which it is possible to go along the edge w_5 to the node v_{12}. The node v_{12} matches the search pattern p_4. Let's add this match to the answer. The state $state8$ has a final reference to the state $state7$ that matches the keyword $w4$.

But among the elements q there is no node from which you can go along the edge w_4 to a new, not yet visited node. At the end of the algorithm on the last character, the array q looks like this (Figure 5): $q = \{(2, v_3), (5, v_1), (5, v_7), (12, v_9), (12, v_{11}), (15, v_4), (15, v_8), (18, v_{12})\}$

As a result, five search patterns were found in the input string S: $A = \{p_2, p_7, p_6, p_1, p_4\}$. When executing the algorithm, we did not change the prefix tree, but only used information about the transitions. The automaton also remained unchanged, only the transition from state to state was performed. Since during the operation of the algorithm to find keywords in a string, an automaton is built according to the Aho-Corasick algorithm, this guarantees that all keywords are found, i.e. when parsing a string, we will not miss a single keyword.

Because the Aho-Corasick algorithm is used to find keywords in the string S when the algorithm is running, this guarantees that all keywords will be found, i.e. we will not miss a single keyword when parsing the string S. When processing a string S, we implicitly build a subtree of the prefix tree T, checking for each subsequent keyword the ability to increase the subtree by adding nodes that exist in T. In the process of the algorithm, we move all possible paths in the prefix tree T, and this ensures that all possible search patterns are found in the string S, because they are all in T.

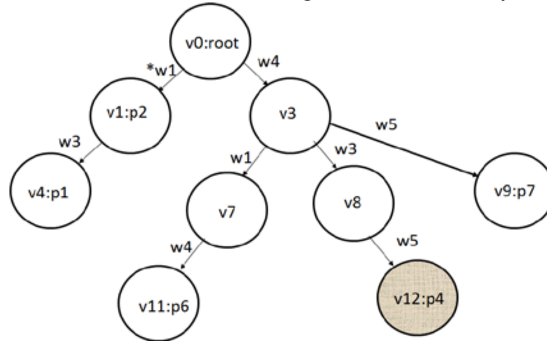

Figure 5: Prefix tree after adding nodes to the array q as a result of processing the last character

6. Analysis of algorithm execution time and memory consumption

The assessment of the complexity of the algorithm can be divided into two parts: the assessment of the pre-processing of search patterns, which is performed only once when the program is initialized, and the assessment of the processing of the next string coming to the program during its operation.

Consider first the stage of pre-processing of search patterns. Let's construct the set of all keywords in time $O(|M|)$, where $|M|$ is the total length of all search patterns. Let K is the total number of keywords. Then the complexity of constructing the set B at step 3 is $O(|M| + K \log |W|)$. Building a prefix tree will take $O(K \log |W|)$. Estimation of the time of construction of the finite automaton in step 5 is $O(|M| \log |\Sigma|)$. That is, in general, given the relationships between variables, the total score will not exceed $O(|M| \log |W|)$. In this case, the memory consumption is linear with respect to the total length of the search patterns, i.e. have an estimate $O(|M|)$.

Now let us move on to evaluating the time and memory of processing a string S that comes during program run. The algorithm goes through all the instances of the found keywords in the string. Let their number be equal occ. For each found keyword, we check all possible states in q, from which it is possible to navigate by this keyword. The number of such states is the number of prefixes in T that were found in the part of the string preceding our keyword. That is, the time complexity of the algorithm can be estimated as $O(occ \times prefnum \times \log |W| + anslen)$. For the worst case, number of prefixes $prefnum$ can be estimated as $O(|T|)$ if we do not count duplicates in the algorithm as possible different answers. The bottom score in the case where there are no matches with the prefixes T is $\Omega(occ)$. We can conclude that the algorithm should work well for our data type, when the input strings S are not overloaded with keywords, and the search patterns for the most part are not subsequences of each other.

During the pre-processing of search patterns at the first and second steps, memory is spent $O(|W|)$ on building keywords. At the third step, memory is spent $O(|M|)$ to build a set of transformed search patterns. At the fourth step, memory is spent $O(|M|)$ on building a prefix tree, and at the fifth step, memory will be spent $O(|W|)$ on building an automaton using the Aho-Corasick algorithm. So, the total memory costs are linear with respect to the total length of the search patterns, that is, they have an estimate $O(|M|)$. It should be noted that the actual memory consumption can be much less, it depends on the number of unique keywords in the search patterns and the degree of similarity of the prefixes of the search patterns. When processing a string S, memory is spent on support an array of nodes. That is, memory consumption is linearly dependent on the number of prefixes $O(prefnum)$ that were found in the string.

7. Software implementation

The software implementation of the developed algorithm is carried out in the Java programming language. The best library in terms of user-agent string analysis speed, which uses a complete list of search patterns from the browscap database [27], was chosen as an analogue for comparison. Software to determine the capabilities of the browser is created using modern tools. Cross-platforming is achieved through the use of a Java virtual machine (JVM), which implements the principle of "write once, run anywhere". The SBT system for automatically compiling projects written in Scala and Java is used to manage the dependencies and plug-ins required for any particular type. The OpenCSV library is used to read patterns, test user-agent instances, and parse CSV files. Comparison of the results of the developed program on the test case with the results of the library-analogue browscap-php is carried out using the JUnit framework. The Browscap-php library, which uses the browscap resource, checks the correctness of the program and returns information about the capabilities of the user's browser. JMH tools [28], which are one of the best for measuring the execution time of Java code in conditions close to real, were used for benchmarking. In this work, JMH is used to compare the speed of the program with similar programs.

The main program is presented in three packages: Trie, Parser and Capabilities. The Trie package is responsible for constructing and presenting compressed information about all input patterns in the form of trees. These are the steps of the offline operation of the algorithm. The Parser package is responsible for the main operation of the program, i.e. implements the online steps of the algorithm. The Capabilities package contains classes and interfaces for presenting information about browser capabilities.

The structure of the program consists of five components (figure 6).

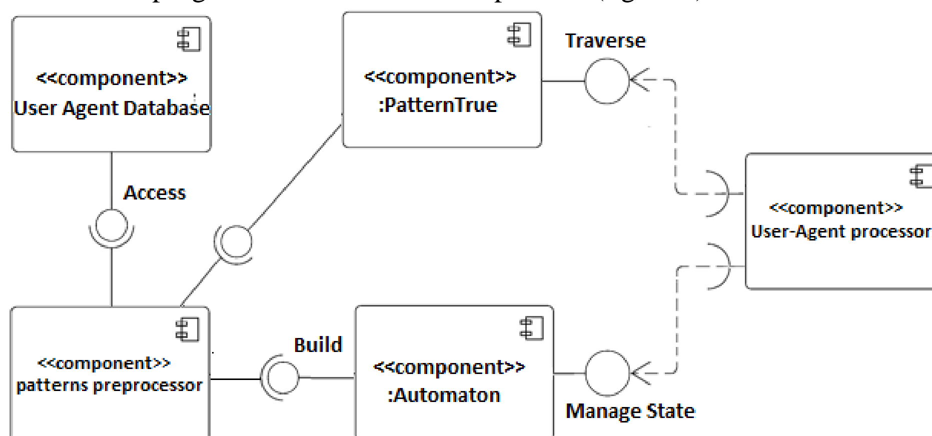

Figure 6: Component structure of the program

The User Agent DataBase component is responsible for presenting and interfacing browser features and capabilities, preserving the relationship between search templates and the browsers that match them. The Pattern Trie component implements and maintains a data structure in the form of a prefix tree, which is built at the stage of pre-processing of search patterns and is used when processing

incoming requests (rows). The Automaton component implements and maintains a finite automaton built according to the modified Aho-Corasick algorithm at the stage of pre-processing of search patterns, which is used in the processing of incoming requests (strings). The Patterns Preprocessor component is responsible for the initial processing of search patterns and builds a prefix tree and automaton. The User-Agent processor handles incoming requests (strings), works with the Pattern Trie and Automaton components to find matches among the search patterns with the input string.

Consider the input query as a string:

«Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36»

The program under consideration analyzes the User-Agent string and displays the following results (figure 7):

Key	Value
browser_name_regex	/^mozilla\/5\.0 \(.*mac os x 10.14.*\) applewebkit.* \(.*khtml.*like.*gecko.*\) .*chrome\/.* safari\/.*$/
browser_name_pattern	mozilla/5.0 (*mac os x 10?14*) applewebkit* (*khtml*like*gecko*) *chrome/* safari/*
parent	Chrome Generic
comment	Chrome Generic
browser	Chrome
browser_type	Browser
browser_bits	32
browser_maker	Google Inc
browser_modus	unknown
version	0.0
majorver	0
minorver	0
ismodified	false
cssversion	3
aolversion	0
device_name	Macintosh
device_maker	Apple Inc
device_type	Desktop
device_pointing_method	mouse
device_code_name	Macintosh
device_brand_name	Apple
renderingengine_name	Blink
renderingengine_version	unknown
renderingengine_description	a WebKit Fork by Google
renderingengine_maker	Google Inc

Figure 7: The result of the program to identify User-Agents

The developed program displays full information about the used device, operating system, and the browser. The parsed User-Agent string corresponds to the Chrome browser installed on MacOS. Browser features like JavaScript, iframe, and others are listed.

8. Results and discussion

Problem description. Research in the field of word processing and analysis methods was performed, in particular, a review of methods and algorithms for comparing search patterns with instances of texts to solve the problem of determining the capabilities of the browser was done.

Solution methods. Our own algorithm for solving the problem of comparing search patterns with input strings was developed based on the analyzed research. The strengths of the studied algorithms and their shortcomings were taken into account in the development of the algorithm. As a result, the developed algorithm received the best known estimate of the operating time from below.

Results. The idea of using a prefix tree and a finite automaton built on the Aho-Corasick algorithm to process incoming string queries were put forward. These data structures have been modified and interconnected to achieve this goal. Testing the correctness of the software application was carried out using the results of the official library browscap-php based on browscap data.

The list of thousands of the most popular User-Agents, requests from which entered the analytical system, was used as a corpus for testing.

A prototype library to determine the capabilities of the browser was developed on the basis of the developed algorithm. The speed of analysis of User-Agent strings collected in the browscap.org database is determined by 6,978±0,046 query processing per millisecond. Given that the database contains more than 62,5 million records of User-Agent strings, the search for the desired User-Agent can take 30 seconds or longer. The running time of the developed application was comparable for different real User-Agent strings. A significant advantage in the speed of processing strings by the developed program was found in comparison with the browscap-java library [27]. The proposed software implementation performs 40.584±0.415 string processing per millisecond. During the benchmarking and testing of the developed library, the efficiency of its work and a significant advantage in speed compared to the nearest competitor was demonstrated, namely a difference of 7 times. The results of benchmarking are presented in table 1.

Table 1
The results of the benchmarking

Library name	Speed of work unit/мs	Accuracy	Note
Developed application	40.584 ±0.415 t	Precise method	
browscap4jFileReade	0.0314	Precise method	The data is taken from the official websiteт
browscap-java	6.978 ± 0.046	Precise method	The library uses caching
browscap-php	0.002	Precise method	Data taken from the official GitHub repository
BitWalker	259.975 ± 1.712	Very low accuracy	Identifies 150 types of browsers
UAParser	0.682 ± 0.003	Average accuracy	Identifies approximately a thousand types of browsers
UADetector	0.161 ±0.005	Low accuracy	Identifies about 600 types of browsers

The value of the results. For businesses that operate on the Internet, the accuracy and speed of the results are very important for further effective work with users. The need to develop fast methods for matching a large number of search patterns (more than 200,000) with User-Agent instances (more than 62.5 million User-Agent records) is relevant.

Information about the source of the request may be necessary to solve the following tasks:

- redirecting requests to the mobile version;
- using specific styles for specific browsers;
- collecting statistics on the number of requests from different devices;
- creation of special rules for processing requests from robots;
- prohibition of access to the site for any web utilities and etc.

9. Conclusions

The problem of insufficient speed of existing libraries to determine the capabilities of the user's (client's) browser, which uses the browscap database, was formulated. A detailed analysis of the literature, available research and developments in the field of matching search patterns with text is made. An algorithm for matching an arbitrary number of search patterns with text instances in real-time has been developed. The time complexity of the algorithm and memory consumption were analyzed. The software implementation has shown a significant performance advantage over existing analogues for identifying browsers capabilities.

Further research can be done to optimize memory consumption during the operation of the algorithm, as well as to reduce the pre-processing time of search patterns in the stage of initialization of the library. You can also extend the ability to specify wildcards, such as making it possible to specify ranges of the number of characters that can be substituted.

10. References

[1] User Agents Database, 2020. URL: https://developers.whatismybrowser.com/useragents/database/.
[2] Browser Capabilities Project, 2020. URL: https://browscap.org/
[3] S. Jha, R. Sommer, C. Kreibich, Recent Advances in Intrusion Detection: 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15–17, 2010.
[4] A. Yakushev, Virtual Machine for Regular Expressions, 2018. URL: https://www.slideshare.net/AlexanderYakushev1/virtual-machine-for-regular-expressions.
[5] J. A. Islam, Analysis of Multiple String Pattern Matching Algorithms, International Journal of Advanced Computer Science and Information Technology, 3:4 (2014) 344–353.
[6] D. E. Knuth, J. H. Morris, V. R. Pratt, Fast pattern matching in strings. SIAM Journal on Computing, 6:2, (1977) 323–350, doi: 10.1137/0206024.
[7] R. M. Karp, M. O. Rabin, Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, 31:2, (1987) 249–260, doi: 10.1147/rd.312.0249.
[8] R. S. Boyer, S. J. Moore, A fast string searching algorithm. Communications of the ACM, 20:10 (1977) 762–772.
[9] A. V. Aho, M. J. Corasick, Efficient string matching: an aid to bibliographic search. Communications of the ACM, 18:6 (1975) 333–340.
[10] B. Commentz-Walter, A string matching algorithm fast on the average. In: Maurer H.A. (eds) Automata, Languages and Programming. ICALP, 71 (1979). Springer, Berlin, Heidelberg, doi:10.1007/3-540-09510-1_10.
[11] S. Wu, U. Manber, A Fast Algorithm for Multi-Pattern Searching, Technical Report TR-94-17 Department of Computer Science, University of Arizona, 1994.
[12] M. J. Fischer, M. S. Paterson, String Matching and Other Products. In: Complexity of Computation, SIAM-AMS Proceedings, (1974)113–125.
[13] H. J. Nussbaumer, Fast Fourier transform and convolution algorithms. Springer Science & Business Media, 2012.
[14] P. Indyk, Faster algorithms for string matching problems: Matching the convolution bound. In: Proceedings of the 39th Annual Symposium on Foundations of Computer Science, IEEE, (1998) 166-173.
[15] M. S. Rahman, C. Iliopoulos, Pattern matching algorithms with don’t cares. In: Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM, (2007) 116-126.
[16] C. Barton, C. Iliopoulos, On the average-case complexity of pattern matching with wildcards, CoRR, 2014.
[17] R. Baeza-Yates, G. H. Gonnet, A new approach to text searching, Communications of the ACM, (1992) 74–82, doi: 10.1145/135239.135243
[18] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences, Cambridge: Cambridge University Press, 2002, doi: 10.1017/CBO9781316135228
[19] Z. Xu, M. Xia, Distance and similarity measures for hesitant fuzzy sets. Information Sciences, 181:11 (2011) 2128-2138.
[20] G. Kucherov, M. Rusinowitch, Matching a set of strings with variable length don't cares, Theoretical Computer Science, 178 (1997) 129-154.
[21] M. Zhang, Yi Zhang, L.Zuo Hu, A faster algorithm for matching a set of patterns with variable length don't cares. Information Processing Letters, 110:6 (2010) 216-220.
[22] T. Haapasalo, P. Silvasti, S. Sippu, E. Soisalon-Soininen, Online dictionary matching with variable-length gaps. In: International Symposium on Experimental Algorithms. Springer, Berlin, Heidelberg, (2011) 76-87.
[23] Na Liu, Fei Xie, X. Wu. Multi-pattern matching with variable-length wildcards using suffix tree. Pattern Analysis and Applications, 21 (2018) 1151-1165.
[24] D. P. Mehta, S. Sartaj, Handbook of data structures and applications. 2nd ed., Chapman and Hall/CRC, 2018.

[25] D.R.Morrison, Practical Algorithm to Retrieve Information Coded in Alphanumeric. Journal of the ACM, 15:4 (1968) 514–534.
[26] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, Cambridge, Massachusetts, 2009, 308–309.
[27] browscap-java, 2020, URL: https://github.com/blueconic/browscap-iava.
[28] Java Microbenchmark Harness, URL: https://openidk.java.net/proiects/code-tools/imh/.