Multiple pattern matching on multiple short strings over a large alphabet

Dalia Camacho García-Formentí

***Abstract*—Pattern matching algorithms have been important in bibliographic search and database queries. Most of the analysis has been concerned on the asymptotic complexity of the algorithms, in terms of the length of the string. However, queries on social media have gained relevance in research, and the asymptotic complexity may not reflect the best pattern matching algorithm for this particular case. Texts from social media tend to be short, for example tweets are less than 280 characters. Another characteristic of these strings is that they are defined over a large alphabet (more than $2,825$ characters), because emoji characters are included. This differs from the alphabets that have been used in previous experiments concerned with pattern matching. Moreover, pattern matching is not done over a single string, but over large sets of strings. Therefore, pattern matching algorithms in this context may have a different performance compared to the asymptotic complexity or the average complexity that relies mostly on the length of the string, snce the length of the strings on empirical exmperiments ranges between $10,000$ characters to 15.8 MB. Pattern matching can be divided into exact and approximate pattern matching, therefore. Currently search queries on twitter are constrianed to the case of exact pattern matching, but approximate pattern search may provide more relevant data. In this work we define an approximate pattern matching algorithm that makes use of tree structures to determine shifts. The data to compare the proposed method against previous algorithms was obtained through queries using the twitter API. Experiments are also performed with synthetic data in order to facilitate generalization.**

I. Introduction

Pattern matching can be used to solve different problems. One of the most important applications of pattern matching is in bioinformatics, especially in DNA, RNA, and protein sequence alignment.[1] However, it is also important in bibliographic search [2], in intruder detection systems,[3] database search,[4] and even in song-lyrics retrieval.[5]

Single pattern matching corresponds to the search of a single pattern in a given string, while multiple pattern matching consists of looking for several patterns. In the case of multiple pattern matching, logical operators (AND, OR, and NOT) can be used within the search. A naive approach to multiple pattern matching would be repeating a single pattern matching algorithm over the same text for every pattern. However, this is not the most efficient approach, thus algorithms specifically for multiple pattern matching have been developed.[2, 6, 7]

The pattern matching problem is also divided into exact and approximate. Approximate pattern matching considers that errors can occur in the pattern or in the text. Such errors may be insertions, deletions, substitutions or transpositions. Oftentimes transpositions are not considered within the possible errors. For example, in DNA sequence alignment transposition is not considered, since it is not the result of a single mutation.[9] However, in written language transposition errors are common typos and should be considered in some cases.[10]

Approximate pattern matching can also be extended from error tolerance to wildcard acceptance.[11] A wildcard is a symbol within the patterns that matches any character from the alphabet that generated both the text and the pattern. As an example of wildcard usage consider we want to find the amino acid Glycine within an RNA sequence. Glycine matches the patterns GGU, GGC, GGA, and GGG, therefore, if * is the wildcard, we can search for the pattern GG*, instead of searching for the four patterns.[12]

Most of the algorithms concerned with pattern and multipattern search focus on complexity and evaluate running times for long texts ranging from $10,000$ characters [13] to 15.8 MB (approximately $15,800,000$characters).[6] Although this approach has a great importance in several applications such as DNA alignment, this is not usually the case for texts coming from social media. For example, Twitter allows at most 280 characters per tweet, which are short texts compared to approximately 3 billion base pairs in the human genome.[14] Another aspect to consider is that if emojis are treated as characters within the alphabet, then the alphabet size is greater than $2,825$ ($2,789$ emoji characters in Unicode, 26 letters from the English language, 10 digits, punctuation marks, and other additional characters including the space character).[15]

Another point to take into consideration is that if searches are emoji-oriented, patterns may be of length one or at most two if flag emojis are used. Therefore, heuristic approaches such as the one of Boyer and Moore might not yield the best results due to short patterns and a long alphabet. In addition, other approaches cannot be implemented for this case. For example, the algorithm by Wu and Manber defines the shift and hash tables in terms of suffixes of length two or three. Moreover, if we are interested in tweets we are not interested in a single tweet, but in a large set of tweets. Thus, we are in the context of multiple pattern matching on multiple strings.

The importance of pattern matching on Twitter lies in its relevance on research. Twitter has been used to study influenza epidemics,[16] the impact of natural disasters, [17], and even predict political preferences.[18].

Currently the Twitter API provides several options for queries, specially with the Premium search operators. These contain keyword search, exact pattern search, emoji search, tweets from specific users, and many more that can be found in the Twitter Guide of Premium Search Operators.[19] However, there are no options for approximate pattern matching. Our aim is to define an approximate pattern matching algorithm that is better in average for this particular case. This may provide insight on the most appropriate algorithm to use

for query search. Moreover, this could further on lead the development of search tools on Twitter that allow users to retrieve tweets containing approximate patterns.

Within this work, we propose an approximate multipattern matching algorithm based on the algorithm proposed by Wu and Manber for exact pattern matching.[6] The changes to the algorithm include the usage of a tree structure instead of a table to determine the shifts, and that it allows errors. Also, we consider an improvement to the algorithm for emoji-oriented searches. We compare it with different approximate pattern search approaches, such as the one by Wu and Manber from 1992 and the one by Liu *et al.* that considers variable wildcards.[7, 11] We also compare it to exact pattern matching algorithms such as the ones by Aho and Corasick, the one by Wu and Maber (in which we base our algorithm), and one using suffix trees.[2, 6, 8]. The latter will be done to consider the same algorithm in case the user prefers exact pattern matching. We will evaluate the algorithms in terms of their theoretical complexity and by running a set of experiments. To carry out the experiments we use data from Twitter from three different queries and a synthetic dataset. We give a more detailed description of the experiments and data in Chapters IV-D and IV-F.

We consider it important to empirically evaluate the algorithms for four main reasons. The first one is most of the algorithms go over the whole string, this is not necessary for all queries. For example if a pattern is found and it corresponds to a query of OR operators, there is no need to continue searching for patterns in the string. The second reason lies in the fact that we are looking for patterns in several strings, this case to our knowledge has not been studied on previous works. Another reason is that even though the size of tweets has an upper bound of 280 characters, it still varies. The last reason is that our algorithm is based on the algorithm by Wu and Manber,[6] which has a good performance in the average case, but the worst case performance would be similar to a naive approach.

To our knowledge there are no previous works that consider the performance of pattern matching algorithms in multiple short strings over large alphabets. Nor have we found any attempts to include approximate pattern search on Twitter.

The scope of this work is limited to previously downloaded Twitter data and synthetic data. We focus only on the tweeted text, no other properties such as geographical location or user were considered. Moreover, other important aspects in information retrieval such as search architectures are not considered.[20]

Previous work related to pattern matching algorithms is described in the following chapter. On the third chapter we provide the concepts to formally define the pattern search problem, as well as the data structures used in the algorithms to compare along with a brief description of the algorithms. The methodology used define the proposed algorithm and the methodology to run empirical experiments is explained in the fourth chapter. Results will be presented in the five chapter, final conclusions will be given in the sixth chapter, and chapter seven explores possible future work.

II. Related Work

In 1975 Aho and Corasick developed a multiple pattern matching algorithm based on a deterministic finite state machine.[2] The construction of the states, go to, and failure functions are constructed from the patterns being searched.

Boyer and Moore defined a single pattern matching algorithm that instead of starting the comparison with the first character it initiated by comparing the last character in the pattern.[13] If there is a mismatch, the pattern is shifted to the right. The amount of positions the pattern is shifted depends on whether the character on the string exists within the pattern, and if so on the position of its last appearance in the pattern. This algorithm has a good performance in the average case but does not perform well for short patterns.

In 1994 Wu and Manber adapted the Boyer Moore algorithm by extending it to multiple pattern matching. For that purpose they added a hash table that would point to those patterns for which their suffix matched the string and and a prefix table that compared prefixes in order to avoid extra searches for words with common suffixes.[6] However, this approach did not solve the efficiency problem on short patterns.

Other approaches to exact pattern matching considered pattern matching on compressed text and on binary encodings. Kim and Kim in 1999 proposed using binary encoding and logical operations between strings to determine a match.[21] In 2002 Kida *et al.* redifined the Aho and Corasick algorithm to consider pattern matching in compressed texts via the LZW method.[22]

Wu and Manber also introduced an algorithm for inexact string matching that considered a maximum number of errors q that could be tolerated.[11] They considered several extensions for their algorithm such as inclusion of wildcards, multiple pattern matching, and even patterns expressed as regular expressions.

In 2010 Wang *et al.* proposed a fuzzy (inexact) search algorithm for MEDLINE database.[4]

Within the context of approximate pattern matching, Liu *et al.* suggested two algorithms for a variable number of wildcards in the patterns.[7] These algorithms search the patterns by making use of the suffix tree of the text.

All previous algorithms were evaluated for pattern matching on a single long string (at least $10,000$ characters).[13] The alphabets over which the patterns and strings were defined include the binary alphabet $\Sigma_{binary} = \{0, 1\}$, the one of DNA nucleotides $\Sigma_{DNA} = \{A, C, G, T\}$, and the one of the English language. In these cases the complexity is driven by the length of the strings and not by the alphabet, but for large alphabets and short strings the complexity may be driven by the preprocessing stage and by the search performed within the data structures defined in this stage. Here we present an algorithm for approximate multiple pattern matching whose supporting data structure depends on the number of characters present in the patterns instead of those present in the string (usually longer than the patterns) or in the whole alphabet. We consider strings of at most 300 characters defined over an alphabet containing over $2,825$ characters. Moreover, we consider a set of short strings instead of a single long string,

thus this work falls into the category of approximate multiple pattern matching on multiple strings.

III. Preliminaries and algorithms

There are two main objectives within this chapter, the first is to define the necessary concepts that characterize the pattern matching problem in general. The second one is to explain the algorithms we use to compare the proposed method.

A. Definitions and concepts

In this section, we provide the necessary concepts to define the pattern matching problem, as well as the concepts needed to understand the different algorithms aimed to find patterns within a string.

Let $\Sigma = \{a_1, a_2, \dots a_\eta\}$ be a finite alphabet of size $|\Sigma| = \eta$. Then, $S = s_1 s_2 \dots s_n$ is a string over the alphabet Σ, if $s_i \in \Sigma$ for all $i = \{1, 2, \dots n\}$.

A substring $S_{i,j}$ of S is given by $S_{i,j} = s_i s_{i+1} \dots s_{j-1} s_j$. A prefix of S is a substring of the form $S_{1,j}$ and a suffix of S is defined as $S_{i,n}$.[7]

The multi-pattern matching problem is defined as follows. Let $P = \{P_1, P_2, \dots P_k\}$ be a set of strings over Σ, these patterns are searched in a string S, also defined over Σ. The length of S is N, the length of P_i is m_i, and $m_i < N$. Usually m_i is much smaller than N.[6]

The previous concepts define the exact pattern matching. For approximate pattern matching, we also need a distance function d and a maximum error q. The distance function is defined as $d : \Sigma^* \times \Sigma^* \to \mathbb{R}^+ \cup \{0\}$, where Σ^* is the language over Σ. In this case a pattern is said to match if there exists a substring $S_{j,l}$ of S, such that $d(P_i, S_{j,l}) \leq q$.

In this work, we consider Levenshtein distance also known as edit distance, in which every error adds 1 to the distance between two strings.[23] The reason for considering the edit distance instead of other distance functions is we assumed that within the context of typos the four possible errors have the same impact.

We consider insertions, deletions, substitutions, and transpositions as possible errors. An insertion consists on inserting a character on the string, a deletion on deleting a character from the string, a substitution on changing a character for another one, and a transposition on changing the order between two adjacent characters.[23]

A naive approach to pattern matching would be to compare the first character of a pattern with the first entry of the S. If there is a match we would then compare the second character and so on until a mismatch occurs, or until all the characters match. If we observe a mismatch we shift the string one space to the right and repeat the process until a match is found or until the string ends. With this approach a single pattern is $O(Nm_i)$, and multi-pattern search would be $O(N \sum_{i=1}^{k} m_i)$. The naive approach for approximate pattern matching has a very similar process, the difference is that if a mismatch occurs and the number of errors is less than the maximum errors tolerated the shift depends on the type of the error and the process continues. The complexity of this algorithm is also $O(N \sum_{i=1}^{k} m_i)$.

$$\mathbf{R} = \begin{array}{c} \begin{matrix} \text{b} & \text{a} & \text{n} & \text{a} & \text{n} & \text{a} \end{matrix} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{matrix} \text{b} \\ \text{a} \\ \text{n} \end{matrix}$$

Fig. 1. Matrix R for pattern 'ban' and string 'banana' for exact pattern matching.

$$\mathbf{R}' = \begin{array}{c} \begin{matrix} \text{b} & \text{a} & \text{n} & \text{a} & \text{n} & \text{a} \end{matrix} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} \text{b} \\ \text{a} \\ \text{n} \end{matrix}$$

Fig. 2. Matrix R' for pattern 'ban' and string 'banana' for approximate pattern matching and edit distance equal to one.

B. Supporting Data Structures

To solve the problem in a more efficient manner, the use of heuristics, different data structures, and even a pattern matching machine have been proposed.[2, 6, 7, 11, 13] The data structures used are arrays, shift tables, hash tables, prefix tables, and suffix trees.

1) Arrays: Two-dimensional arrays (matrices) are used in the approximate pattern matching algorithm by Wu and Manber in 1992.[11] The columns of the matrix R represent the characters of the string S as they appear, and the rows represent the characters of the pattern P that is being searched. For exact matching $R[i, j]$ is one, if $P_{1,i}$ matches the substring $S_{j,j+i}$, and zero in any other case. In a similar way matrices for approximate pattern matching can be defined, by assigning a one to $R[i, j]$ if there are at most $err = q - 1$ errors, or if there are q errors and character $p_{i-q} = s_j$.

As an example of single pattern matching consider the pattern 'ban' and the string 'banana', then the matrix R for exact pattern matching is presented in Figure 1. Matrix R for the same search and approximate pattern matching with edit distance equal to one is shown in Figure 2.

2) Shift, hash, and prefix tables: A **shift** table defines the number of positions one has to move within the string S to continue with the search process when a mismatch occurs. This table has a row for each character in Σ for single pattern matching,[13] or a row for all possible blocks B of size b over Σ for the multiple pattern matching problem.[6] If a block is present in any of the patterns the shift will be determined by the position in which it occurs within the pattern. If the block is not a substring of any pattern, then the shift will be the same as the length of the shortest pattern.

The algorithm proposed by Wu and Manber in 1994 uses a **hash table**. [6] The hash table contains blocks B_i formed by the last 2 or 3 characters of the patterns in P and pointers to the patterns that have B_i as a suffix. The hash table also includes pointers to the **prefix table**. The prefix table contains all the prefixes of size 2 of the patterns in P. The reason for having this table is that there are common suffixes in English and by

TABLE I
TABLES USED IN THE ALGORITHM PRESENTED BY WU AND MANBER IN 1994 FOR THE STRING *'bananaistooshortforthisexample'* AND THE PATTERNS $P = \{anais, oosh, forth, this\}$. [6]

TABLE II
SHIFT TABLE

Suffix	shift
a	2
b	4
e	4
f	4
h	0
i	1
l	4
m	4
n	3
o	2
p	4
r	2
s	0
t	1
x	4

TABLE III
HASH TABLE

Suffix	pointer
h	2, 3
s	1, 4

TABLE IV
PREFIX TABLE

Prefix	pointer
a	1
f	2
o	3
h	4

TABLE V
POINTERS

pattern	pointer
anais	1
forth	2
oosh	3
this	4

using this table they reduce the number of comparisons.[6]

We show a simplified example of these tables considering blocks are of length one in contrast to the recommended length of two or three. We consider the alphabet $\Sigma = \{a, b, e, f, h, i, l, m, n, o, p, r, s, t, x\}$ where $|\Sigma| = 15$.[1] We use the string *'bananaistooshortforthisexample'* and the patterns $P = \{anais, oosh, forth, this\}$. The shift, hash, prefix and pointers tables are exemplified in Table I.

3) Suffix trees: A **suffix tree** associated to string S is a tree that contains all suffixes of S. To properly distinguish the suffixes, an additional character (\$) is added at the end of S then $S\$ = s_1 s_2 \dots s_n\$$. The suffix tree has $n+1$ leaves, where n is the length of S. The root node has exactly $t + 1$ direct children, where t is the number of unique characters in S.

The construction starts by adding the root node. Then each of the suffixes is added in order (decreasing). If the root node does not have a branch labeled with the first character of the suffix, add a branch labeled with this suffix. If instead there is a branch whose label starts with the first character of the suffix, compare the following characters of the suffix with the label until there is a mismatch. When a mismatch occurs change the label of the branch to the characters that match between the two strings and add two branches to the node. One will be labeled with the characters from the suffix that did not match the label, and the other one will be labeled with the characters from the label that did not match the suffix.[7] As an example the suffix tree for the string `banana` is presented in Figure 3.

[1]If we used blocks of length two, the shift table would be of length $15^2 = 225$ and is not very helpful for an example.

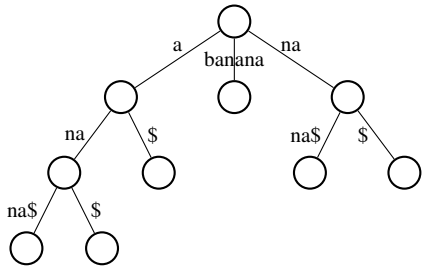

Fig. 3. Suffix tree for string `banana`.

4) Pattern Matching Machine: A **pattern matching machine** receives as input the string S and as an output it returns which patterns of P are present in S and the position of S in which they appear.[2] This machine consists of a set of states, a go to function, an output function, and a failure function. The states and functions are constructed from the patterns in P. The go to function is constructed from an initial state 0 all other states correspond to each prefix of the patterns. A transition is given by the go to function if the following character in the string is the following character in the prefix of the pattern. A mismatch does not necessarily mean one starts in state zero again. If a suffix of the string that corresponds to the current state is a prefix of another pattern, then the mismatch could be a match for a different pattern. Thus, the failure function is used to define which pattern will be compared when a mismatch occurs at a given state. The failure function also specifies which state should be evaluated next when a match occurs. The output function indicates when a pattern is found in the string and which pattern it is.

As an example consider the matching machine in Figure 4 for the patterns $P = \{`ana', `anais', `ban', and `nana'\}$.

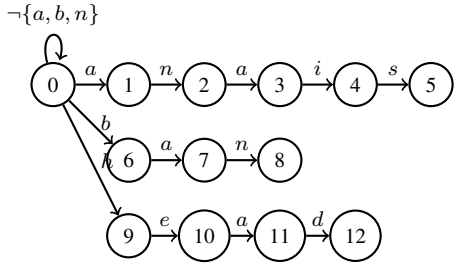

Fig. 4. Go to function for the patterns 'ana', 'anais', 'ban', and 'nana'.

$$Output(state) = \begin{cases} `ana', & state = 3 \ \ or \ \ state = 12, \\ `anais', & state = 5, \\ `ban', & state = 8, \\ `nana', & state = 9 \end{cases} \quad (1)$$

Fig. 5. Output function

$$Failure(state) = \begin{cases} 9, & state = 3, \\ 1, & state = 8 \ \ or \ \ state = 12, \\ 0, & \text{in other case.} \end{cases} \quad (2)$$

Fig. 6. Failure function

C. Algorithm Overview

We will describe the algorithms, whose running time will be compared for pattern search on short texts over a large alphabet.

1) Aho and Corasick 1975: The pattern matching algorithm by Aho and Corasick was specifically designed for the exact multiple pattern matching problem.[2] It consists on building a pattern matching machine from the set of patterns $P = \{P_1, P_2 \dots P_k\}$. The pattern matching machine receives as input string S the patterns in P that are substrings of S are returned as outputs along with the position of S in which they appear.

We decided to compare this algorithm, because even when it was one of the first attempts in multiple pattern matching, it still characterizes pattern matching algorithms based on automata.[7, 24]

2) Wu and Manber 1992: This algorithm uses matrices constructed as defined in Section III-B1.[11] It was originally defined for single approximate pattern matching. However, an extension for multiple pattern matching is provided. They suggest to concatenate all the patterns in P to create a new pattern. They also define an additional array M which keeps the positions that correspond to the first character of each pattern. They define additional arrays to compare the patterns simultaneously when the first character appears.

The reason for selecting this algorithm is its focus on pattern matching allowing errors and the multiple extensions that are given for various approximate pattern matching flavors such as pattern matching with wildcards, matching of regular expressions, and a combination between exact and approximate pattern matching.[6]

3) Pattern matching with suffix trees: Pattern matching using suffix trees is described by Haubold as a common method in bioinformatics. [9] It consists on creating a suffix tree for string S and looking for the patterns in the suffix tree. Liu *et al.* order the patterns in alphabetical order as a preprocessing procedure for the multiple pattern matching problem with variable wildcards,[7] likewise, this preprocessing step makes exact pattern search more efficient.

This approach was chosen because of its importance in bioinformatics and the fact that it is the base of more recent algorithms such as Wu *et al.* [7, 8]

4) Wu and Manber 1994: This algorithm was given for the exact multiple pattern matching problem with the restriction that all patterns in P have the same size, or that all patterns are considered to have the same size as the shortest pattern.[6] This method is based on the single pattern matching algorithm of Boyer and Moore.[13] Suffixes are compared first and a shift table is defined to determine the number of positions in the string that will not be compared when a mismatch occurs. A hash table is used to determine to which pattern a matching suffix corresponds, and the prefix table is used to reduce the number of comparisons in case the patterns end with commonly used suffixes such as 'ion' and 'ing'.[6]

This algorithm has been important for different areas, algorithms such as Kalign in bioinformatics and led to the development of search algorithms based on hash tables.[24, 25] Moreover, we use this algorithm as the basis for the algorithm we propose.

5) Liu et al. 2018: The algorithm proposed by Liu *et al.* is based on suffix tree search for multiple pattern matching. It also adds the possibility of a variable number of wildcards. An example of wildcard usage was given in Chapter I. Another example with a variable number of wildcards is the following. If we were interested in the word 'goal' when a soccer player scores, we might expect a variable amount of 'o's, 'a's, and 'l's. Therefore, we would be also interested in capturing texts with the word 'goaal', 'goaaal', 'goooaaall', and so on. Then, we can express the patterns we are looking for as go*[0,n]a*[0,n]l*[0,n], where the amount of wildcards ranges from 0 to n in each position. For this last example we should be careful when processing the data obtained. Since wildcards match all characters in Σ, spurious results may occur.

This algorithm focuses on pattern search with variable wildcards. After ordering the patterns and constructing the suffix tree T_S of S. The ordered patterns will be searched, when the position of a variable wildcard appears the last matching position is saved. Strating from the minimum number of wildcards the rest of the pattern is searched. If there is a mismatch we return to the position previously saved and add an additional wildcard until the pattern matches or a maximum number of wildcards is used.

The reason for selecting this algorithm is its novelty in the field of variable wildcard usage.

The algorithms here selected take different approaches and use different data structures. Then, our algorithm can be compared with various approaches in the context of pattern matching.

IV. Research Methodology

A. Problem characteristics

The problem for which we are comparing the different pattern matching algorithms has the following characteristics. Let $\Psi = \{S_1, S_2, \dots S_\psi\}$ be the set of strings in which we are searching the set of patterns $P = \{P_1, P_2, \dots P_k\}$. $S_i \in \Psi$ and P are defined over an alphabet Σ. The size of the string S_i is $|S_i| = N_i \leq 280$ and the size of the alphabet is $|\Sigma| = \eta \geq 2,825$.

B. Logical operators within queries

The context we are working on is related to data from Twitter, the multiple pattern matching problem could be further

on considered for queries rather than just in previously downloaded data. Therefore, we may want to consider the effect of using different logical operators within the search. Depending on the relation between the patterns through logical operators we may need to find one, some, or all of the patterns.

For example if the patterns are $P = \{\text{dog}, \text{cat}, \text{parrot}\}$, then we may search for: dog OR cat OR parrot, dog AND cat AND parrot, dog AND (cat OR parrot), (dog AND cat) OR parrot, and so on.

Since the amount of possible combinations of strings through logical operators is $\Omega(2^n)$,[2] we will only consider four cases.

In the first one, the search algorithm stops when a pattern is found. This would be comparable to the case in which all keywords are related with an OR operator. In the second one, we stop when a pattern is missing or when all patterns are found (depending on the algorithm used). This is equivalent to relate all keywords with an AND operator. In the third one, we look for all the patterns in P on the string S. This is analogous to define which patterns are present in the text. The first two cases approximate a lower bound of the running times, and the third case approximates an upper bound. For the last case the logical operators between words will be obtained randomly, this last case in average will approximate the average running times.

The third case is the one used by Aho and Corasick, by Wu and Manber in 1994, and by *Liu et. al*.[2, 6, 7] The algorithms presented by Wu and Manber in 1992 and the one using suffix trees are mostly described for single pattern search, thus they are comparable to the first and second cases. However, the Wu and Manber algorithm of 1992 is extended to deal with multiple patterns in the extension they propose.

C. Algorithm Characterization

To define the proposed algorithm we will combine the naive approximate pattern search algorithm with the multipattern matching algorithm by Wu and Manber. To do this we consider the size of the blocks of the shift table to be $b = 2$. Instead of a shift table and a hash table we propose a tree structure that contains both the shift and the hash.

D. Data

Articles related to pattern matching algorithms have used as data: random texts and random keywords, [2] or substrings of selected texts with topics ranging from Wall street Journal articles to DNA sequences.[6, 7, 13] Within this work we will evaluate the running times using both random strings and data from Twitter. The random strings will be of length ranging between 1 and 300, and the random patterns will range from 1 to 20. A total of $100,000$ random strings and 500 random patterns will be generated for the evaluation. Both texts and patterns will be created considering 75% of the characters will be low case letters whose the frequency will reflect that in the English language shown in Table VI.[26] The other 25% of the characters will be randomly selected with the same probability from the possible emoji characters, punctuation marks, numbers, and special symbols.

Letter frequency in English

Letter	Percentage
E	12.49%
T	9.28%
A	8.04%
O	7.64%
I	7.57%
N	7.23%
S	6.51%
R	6.28%
H	5.05%
L	4.07%
D	3.82%
C	3.34%
U	2.73%
M	2.51%
F	2.40%
P	2.14%
G	1.87%
W	1.68%
Y	1.66%
B	1.48%
V	1.05%
K	0.54%
X	0.23%
J	0.16%
Q	0.12%
Z	0.09%

TABLE VI

Influenza related keywords in query

flu			
cough	couhg	cogh	
sore throat	soar throat	sore throat	
headache	hedache	headacke	headake

TABLE VII

Data from Twitter was obtained using the Twitter API and the package `twitteR` in `R` software.[27, 28]

The query to obtain tweets was based on the keywords ('flu', 'cough', 'sore throat', 'headache') chosen by Culotta who studied influenza epidemics by analyzing tweets.[29] These keywords have proven to work better than others in terms of the correlation between tweets containing these keywords and influenza-like illnesses in health records.[30] Since influenza epidemics has been a common research topic using Twitter data, the four words used by Culotta were chosen.[16, 29, 30, 31, 32, 33]

Since we will also analyze approximate pattern matching, we added more keywords by adding typos to the four keywords provided by Culotta. The query contained the keywords in Table VII linked with an OR operator. The query was restricted to tweets in English generated since November 1st 2018 to November 28 2018. The total number of tweets obtained from this search was $14,813$.

Another query contained all possible keywords that can be extracted from the regular expression go*[0,2]a*[1,7]l to evaluate the running times of pattern matching with variable wildcards. Once again this query was restricted to tweets in English generated since November 1st 2018 to November 28 2018. The total number of tweets obtained from this search

[2]This can be easily seen as follows: the amount of logical operators per k strings is $k-1$, if we consider AND and OR only there are 2^{k-1} possible configurations of the logical operators. This estimate does not consider permutations between the strings.

was $3,376$. To obtain more data we replicated the query in Spanish for the regular expression go*[0,10]l for the same dates constrained to $50,000$ tweets[3].

E. Algorithm implementation

Apart from the algorithm presented in this work we will implement the methods for exact pattern matching that will be implemented are the one by Aho and Corasick,[2] the one by Wu and Manber,[6] and one using suffix trees. For approximate pattern matching we will compare the algorithm by Wu and Manber,[11] with the one by Liu *et al* [7].

All algorithms will be programmed in `R` version 3.5.1. All experiments will be conducted on a laptop with AMD A10-5745m apu with radeon(tm) hd graphics 4 processor, 1.0 GHz CPU and 8 GB main memory, running on a 64-bit Ubuntu 18.04.1 LTS OS.

F. Evaluation metrics

As an evaluation metric of the worst case scenario of the proposed algorithm we consider the theoretical complexity of our algorithm both in terms of time and space.

Also, several experiments will be held to show the performance of the algorithm proposed against previous algorithms in the context of multiple pattern matching on multiple strings in which the string length is small (less than 300) and the alphabet is large (at least $2,825$).

For both exact and approximate matching we will compare running times for an increasing amount of patterns (from 2 to 25) of a fixed length and for a fixed number of patterns whose length will range from one (emoji character) to fifteen. The number of strings in which the patterns are searched will vary from 500 to $50,000$ strings.

As we stated previously we will look at four cases, the first in which the algorithm stops when a pattern is found, the second one when the algorithm stops when all patterns are found or when a pattern is missing. These two cases correspond to a lower bound of the running times. In the third case all patterns must be searched for. The running for the third case correspond to an upper bound of the running times. In the fourth case logical operators between keywords will be obtained randomly. To achieve the latter, keywords will be permuted and logical operators will be assigned randomly. This last case will be repeated $10,000$ times to find the average running time which will approximate to the average running time for real queries.

Usually the running time used to define the data structures is not taken into account. However, in this case we will also consider it. There are two reasons for this. One is that we are searching over multiple strings. Thus, if we use suffix trees, a suffix tree must be defined for each string, but only a finite state machine must be built when using the Aho and Corasick algorithm.[2, 7] The second reason is that for some algorithms the preprocessing stage complexity is driven by the length of the alphabet, this is the case of the algorithm by Wu and Manber of 1994.[6]

We will also evaluate the space that each algorithm takes in terms of the space that the data structures occupy. This is also important considering the shift table must have at least $2,825$ entries if we define it in the same manner as Boyer and Moore,[13] or at least 2825^2 if we use the approach of Wu and Manber.[6]

[3]Constrained to the query $50,000$ tweets were found with the given keywords, this indicates there are more than $50,000$ with the given characteristics. However, we work only with these $50,000$ tweets.

V. Results

VI. Conclusions

VII. Future Work

Within this work approximate pattern matching is based on the Levenshtein distance and in wildcard usage. However, other distance metrics such as the Hamming distance, the Episode distance or the Longest Subsequence distance may be evaluated within the approximate pattern matching algorithm here provided.

Another area of opportunity and improvement is to parallelize the pattern matching algorithms. For pattern matching on multiple strings parallelization is straightforward, since strings are independent from each other. However, it may be more interesting to evaluate the possibility of parallelizing pattern matching on a single string.

References

[1] T. Lassmann and E. L. Sonnhammer, "Kalign – an accurate and fast multiple sequence alignment algorithm," *BMC Bioinformatics*, vol. 6, no. 1, p. 298, Dec 2005. [Online]. Available: https://doi.org/10.1186/1471-2105-6-298

[2] A. Aho and M. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[3] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 4, 2004, pp. 2628–2639.

[4] J. Wang, I. Cetindil, S. Ji, C. Li, X. Xie, G. Li, and J. Feng, "Interactive and fuzzy search: a dynamic way to explore medline," *Bioinformatics*, vol. 26, no. 18, pp. 2321–7, 2010.

[5] P. Knees, M. Schedl, and G. Widmer, "Multiple lyrics alignment: Automatic retrieval of song lyrics," in *ISMIR 2005, 6th International Conference on Music Information Retrieval, London, UK, 11-15 September 2005, Proceedings*, 2005, pp. 564–569. [Online]. Available: http://ismir2005.ismir.net/proceedings/1059.pdf

[6] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Tech. Rep., 1994.

[7] N. Liu, F. Xie, and X. Wu, "Multi-pattern matching with variable-length wildcards using suffix tree," *Pattern Analysis and Applications*, 09 2018.

[8] Bernhard Haubold and Thomas Wiehe, *Introduction to Computational Biology* , 1st ed. Birkhäuser Basel, 2006.

[9] ——, *Introduction to Computational Biology* , 1st ed. Birkhäuser Basel, 2006, ch. 2. Optimal Pairwise Alignment, pp. 11–42.

[10] K. Kukich, "Techniques for automatically correcting words in text," *ACM Comput. Surv.*, vol. 24, no. 4, pp. 377–439, Dec. 1992. [Online]. Available: http://doi.acm.org/10.1145/146370.146380

[11] S. Wu and U. Manber, "Fast text searching: allowing errors," *Communications of the ACM*, vol. 35, no. 10, pp. 83–91, 1992.

[12] Bernhard Haubold and Thomas Wiehe, *Introduction to Computational Biology* , 1st ed. Birkhäuser Basel, 2006, ch. C Molecular Biology Figures and Tables, pp. 279–284.

[13] R. Boyer and J. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[14] National Human Genome Research Institute, "The Human Genome Project Completion: Frequently Asked Questions," 2018, https://www.genome.gov/11006943/human-genome-project-completion-frequently-asked-questions/, Last accessed on 2018-11-11.

[15] Unicode Technical Reports, "Unicode® Technical Standard 51 UNICODE EMOJI," 2017, https://www.unicode.org/reports/tr51/Identification, Last accessed on 2018-11-11.

[16] C. Comito, A. Forestiero, and C. Pizzuti, "Twitter-based influenza surveillance: An analysis of the 2016-2017 and 2017-2018 seasons in italy," in *Proceedings of the 22Nd International Database Engineering & Applications Symposium*, ser. IDEAS 2018. New York, NY, USA: ACM, 2018, pp. 175–182. [Online]. Available: http://doi.acm.org/10.1145/3216122.3216128

[17] Z. Ashktorab, C. Brown, M. Nandi, and A. Culotta, "Tweedr: Mining twitter to inform disaster response," in *ISCRAM*, 2014, pp. 354–358.

[18] S. Kannangara, "Mining twitter for fine-grained political opinion polarity classification, ideology detection and sarcasm detection," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, ser. WSDM '18. New York, NY, USA: ACM, 2018, pp. 751–752. [Online]. Available: http://doi.acm.org/10.1145/3159652.3170461

[19] Twitter, "Search Tweets, Premium Search Operators," 2018, https://developer.twitter.com/en/docs/tweets/search/api-reference/get-search-tweets.html, Last accessed on 2018-11-30.

[20] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-time search at twitter," in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ser. ICDE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1360–1369. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2012.149

[21] S. Kim and Y. Kim, "A fast multiple string-pattern matching algorithm," in *In Proceedings of 17th AoM/IAoM Conference on Computer Science*, 1999.

[22] T. Tao and A. Mukherjee, "Multiple-pattern matching for lzw compressed files," in *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*, vol. 1, April 2005, pp. 91–96 Vol. 1.

[23] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001. [Online]. Available: http://doi.acm.org/10.1145/375360.375365

[24] P. Zeng, Q. Tan, X. Meng, Z. Shao, Q. Xie, Y. Yan, W. Cao, and J. Xu, "A multi-pattern hash-binary hybrid algorithm for url matching in the http protocol," *PLOS ONE*, vol. 12, no. 4, pp. 1–21, 04 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0175500

[25] Q. Zou, Q. Hu, M. Guo, and G. Wang, "Halign: Fast multiple similar dna/rna sequence alignment based on the centre star strategy," *Bioinformatics*, vol. 31, no. 15, pp. 2475–2481, 2015. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btv177

[26] Peter Norvig, "English Letter Frequency Counts:Mayzner Revisited or ETAOIN SRHLDCU," 2018, http://norvig.com/mayzner.html, Last accessed on 2018-11-28.

[27] J. Gentry, *twitteR: R Based Twitter Client*, 2015, r package version 1.1.9. [Online]. Available: https://CRAN.R-project.org/package=twitteR

[28] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2018. [Online]. Available: https://www.R-project.org/

[29] A. Culotta, "Towards detecting influenza epidemics by analyzing twitter messages," in *Proceedings of the First Workshop on Social Media Analytics*, ser. SOMA '10. New York, NY, USA: ACM, 2010, pp. 115–122. [Online]. Available: http://doi.acm.org/10.1145/1964858.1964874

[30] S. Doan, L. Ohno-Machado, and N. Collier, "Enhancing twitter data analysis with simple semantic filtering: Example in tracking influenza-like illnesses," in *Proceedings of the 2012 IEEE Second International Conference on Healthcare Informatics, Imaging and Systems Biology*, ser. HISB '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 62–71. [Online]. Available: http://dx.doi.org/10.1109/HISB.2012.21

[31] A. Signorini, A. M. Segre, and P. M. Polgreen, "The use of twitter to track levels of disease activity and public concern in the u.s. during the influenza a h1n1 pandemic," *PLOS ONE*, vol. 6, no. 5, pp. 1–10, 05 2011. [Online]. Available: https://doi.org/10.1371/journal.pone.0019467

[32] M. J. Paul, M. Dredze, and D. Broniatowski, "Twitter improves influenza forecasting," *PLoS Curr*, vol. 6, Oct 2014.

[33] D. A. Broniatowski, M. J. Paul, and M. Dredze, "National and local influenza surveillance through twitter: An analysis of the 2012-2013 influenza epidemic," *PLOS ONE*, vol. 8, no. 12, 12 2013. [Online]. Available:

https://doi.org/10.1371/journal.pone.0083672