Software Engineering Department
Ort Braude College
Course 61998: Extended Project in Software Engineering

Capstone Project Phase 1
22-1-R-15

Enhancement of the K-mismatch search algorithm Via tree data-structures

Karmiel – October 2021

Authors:
Sameer Kandeel
Arkan Mohamad

Supervisor:
Dr. Zakharia Frenkel

Abstract :

As the years pass, the need for information is growing and with it the amount of information is exponentially growing, this results in a larger database that requires a longer search time. This raises the need for a fast and efficient searching algorithm. Especially an algorithm that supports finding a word called a search word in a text file that contains a large amount of words. The goal of our project is to do a significant improvement of the Basic algorithm for the k-mismatch problem by creating a data-structure that contains all of the words combinations from the MCS and using it as a map.

keywords*: K-mismatch problem, MCS (q-gram filters).*

Table of Contents

1. *Introduction*

The k-mismatch search problem is related to the development of an algorithm for quick finding to all positions in a given text, which are correspondent to words similar to a word of search. The similarity is defined via the Hamming distance (and lately, an enhanced version of the algorithm defines the similarity via the Levenshtein distance).
Conditions considered with each search are similarity threshold, sizes of text, alphabet and the searched words.
In the naïve algorithm comparison of the searched word(s) with the text should be carried out at each position of the text, which is not acceptable for many cases. For example, if want to do an indexing (i.e., to find all similar words for each word in the given text) of the text composed by 10^9 words (about 1G) – 10^{18} comparisons should be carried out which is practically impossible.

Recently, the idea of gapped q-gram filters was generalized for quick solution of the k-mismatch problem under a wide range of conditions.
Another idea for enhancing the algorithm was proposing to use Levenshtein`s distance as the Distance between two words.
Although the previous enhancements made a step ahead in terms of resources needed, the problem with computer resources remains , especially with the RAM, since the latest algorithm improvements are based on building text-maps and on building as many as possible (as many as needed/relevant) of them, so for instance, if the file we are going to apply the search on is 1GB, we will need about 32GB of RAM to store 32 maps for the text, which is a lot, and practically, not so many computers have this amount of RAM.

In the current project, we aim to decrease the required amount of RAM needed to perform the exact same searches with the exact same conditions that were practically “impossible” on a “weak” computer, or even a common one; by applying the use of tree data structures on the text-maps and then being able to discard maps that are not needed, which is a good thing for CPU too (many un-needed comparisons will be discarded).

2. *Background and Related Work*

In the following section, we review the existing k-mismatch approaches with a special focus on the speed of each algorithm for this problem, and have far have we come from the naïve-algorithm's performance.

2.1. Fuzzy Search – Approximate String Matching

Fuzzy search or approximate string matching is a task for searching of strings matching a pattern approximately (i.e. not necessarily exactly)[1]. This type of search is well known for supporting a search patterns in large DNA or protein databases. The fuzzy search applications are becoming wider every year, especially with the exponentially growing of data. The task of search should be executed as quickly as possible, while requiring minimal computational resources (RAM, CPU, etc.).

The task may vary, according to selected similarity metrics between the query and searched patterns. It can be for example, Hamming Distance (k-mismatch problem), Levenshtein Distance (k-mismatch with insertions and deletions), etc.

2.1.1. Landau and Vishkin "Kangaroo Method"

In this method the preprocess stage consists of building an LCA (least common ancestor) tree from the text, so that every tree's leaf contains every possible word ending from the text and the pattern. Every node that is not a leaf regarded as an ancestor. A more detailed explanation of the algorithm can be found at the adjacent article .[8]

The search phase of the algorithm runs as following: The algorithm checks every index in the text.

If the word that starts in the current index is similar to the pattern (rather than exact) under the given threshold, the number of operations that are taking place in every indexed word depend on that given threshold (K). If the indexed word is exactly the same, the search will be done in 1 operation, if the indexed word has 1 mismatch the search will be done in 2 operations, 2 mismatches – 3 operations and so forth until the number of mismatches is greater than the given threshold. At that time the indexed word's search will cease and the algorithm will advance to the next index. We can easily notice that the search run-time results only by the size of the text itself and the given threshold, rather than the probability of the occurrences of the pattern within the text.

A more accurate greater lower bound for the search run-time of the algorithm is:

Θ (N + P1*1 + P2*2 + … + Pi*I + … + Pk*k) || $0 < i \leq K$ Where Pi is the expected amount of words that are similar to the pattern with i mismatches. We can easily see that the number of operations for a search in this method is greater than N operations, therefore C>1.

2.1.2. Naïve Algorithm Of K-Mismatch

As mentioned, the fuzzy search uses a distance function D, for this method the distance function is Hamming Distance. Hamming distance measures the minimal substitutions required in order to change one string P to another T, let $P=p_1p_2\ldots p_m$, and $T=t_1t_2\ldots t_m$ then the Hamming Distance HAM(P,T) = The number of locations j where $p_j \neq t_j$.

Example:

T = ARKAN P = ASKIN

HAM(T,P) = 2.

The naïve algorithm of K-Mismatch is implemented by receiving an input $T=t_1t_2t\ldots t_n$, $P = p_1p_2\ldots p_m$ and K the output is calculated by first calculating for each i in T, $HAM(P,t_it_{i+1}\ldots t_{i+m-1})$, and then choosing the positions that the HAM = K.

Example:

Input:

P = ABBAAC.

T = ABCAABCAC.

K = 2.

Counting mismatches:

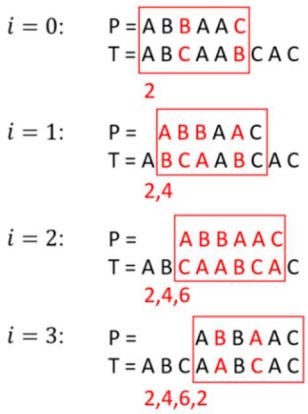


Figure 1. *Example of counting mismatches*

Choosing relevant positions according to K:

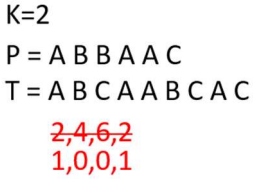


Figure 2. *Example of choosing relevant positions*

The problem with the naïve algorithm is that it is slow with a running time of O(nm),

where n = |T|, m = |P|. Therefore, a better algorithm is devised

2.2. k-mismatch methods for approximate string matches

The core element of a word-oriented search method is a dictionary. The dictionary is a collection of distinct searchable strings (or string sequences) extracted from the text.

Dictionary strings are usually indexed for faster access. A typical dictionary index allows for exact search and, occasionally, for prefix search.

Given the search pattern p and a maximum allowed edit distance k, these methods retrieve all dictionary strings s (as well as associated data) such that the distance between p and s is less than or equal to k.

During indexing, all unique text substrings with lengths in a given interval (e.g., from 5 to 10) are identified. For each unique substring s, the indexing algorithm compiles the list of positions where s

occurs in the text. Finally, all unique substrings are combined into a dictionary, which is used to retrieve occurrence lists efficiently.

In our project we are going to improve a filtering algorithm by implication of a tree-related data structure usually used in the direct methods. Below we propose a short review of the main types of the data structures and some methods commonly applied in the field.

To sum up the indexing methods that can be used in the string matching problem, we divided the methods into two main groups:

— Direct methods

— Sequence-based filtering methods

Each category can be further subdivided based on characteristics specific to the method type.

2.2.1. Direct search methods

look for complete patterns. They include: prefix trees, neighborhood generation, and metric space methods. A neighborhood generation method constructs a list of words within a certain edit distance from a search pattern. Then the words from this list are searched for exactly. Both prefix trees and most metric space methods recursively decompose a dictionary into mutually disjoint parts, i.e., they create a hierarchy of space partitions. A search algorithm is a recursive traversal of the hierarchy of partitions. At each recursion step, the search algorithm determines whether it should recursively enter a partition or backtrack, i.e., abandon the partition and return to a point where it can continue searching. Such a decision is made on the basis of proximity of the search pattern and strings inside the partition.

2.2.2. A filtering search method

has a filtering step and a checking step. During the filtering step, the method builds a set of candidate strings. The checking step consists in element-wise comparison of the search pattern and candidate strings. Most filtering methods compute the edit distance between a candidate string s and the search pattern p to discard strings s such that $ED(p, s) > k$. Alternatively, it is possible to use a fast on-line algorithm, which essentially applies a second filter. Filtering algorithms can be characterized by a filtering efficiency. It is informally defined as a fraction of dictionary strings that are discarded at the filtering step. A complementary measure is filtering inefficiency, which is defined as one minus filtering efficiency. Filtering inefficiency is computed as the average number of verifications during the checking step divided by the number of dictionary strings.

2.2.3. Sequence-based filtering methods

employ short string fragments to discard non-matching strings without evaluating the edit distance to a search pattern directly. We call these fragments features. Sequence-based filtering methods can be divided into pattern partitioning and vector-space methods. Pattern partitioning methods rely on direct indexing of and searching for string fragments, e.g., using an inverted file. Vector-space methods involve conversion of features (usually q-grams and unigrams) into frequency vectors or signatures. Then frequency vectors and signatures are indexed using general vector space methods.

2.2.4. **In addition**, there are **hybrid methods** that integrate several access methods to improve retrieval time. Many hybrid methods also rely on filtering. Hybrid methods are very diverse: we do not discuss them separately from other algorithms.[2]

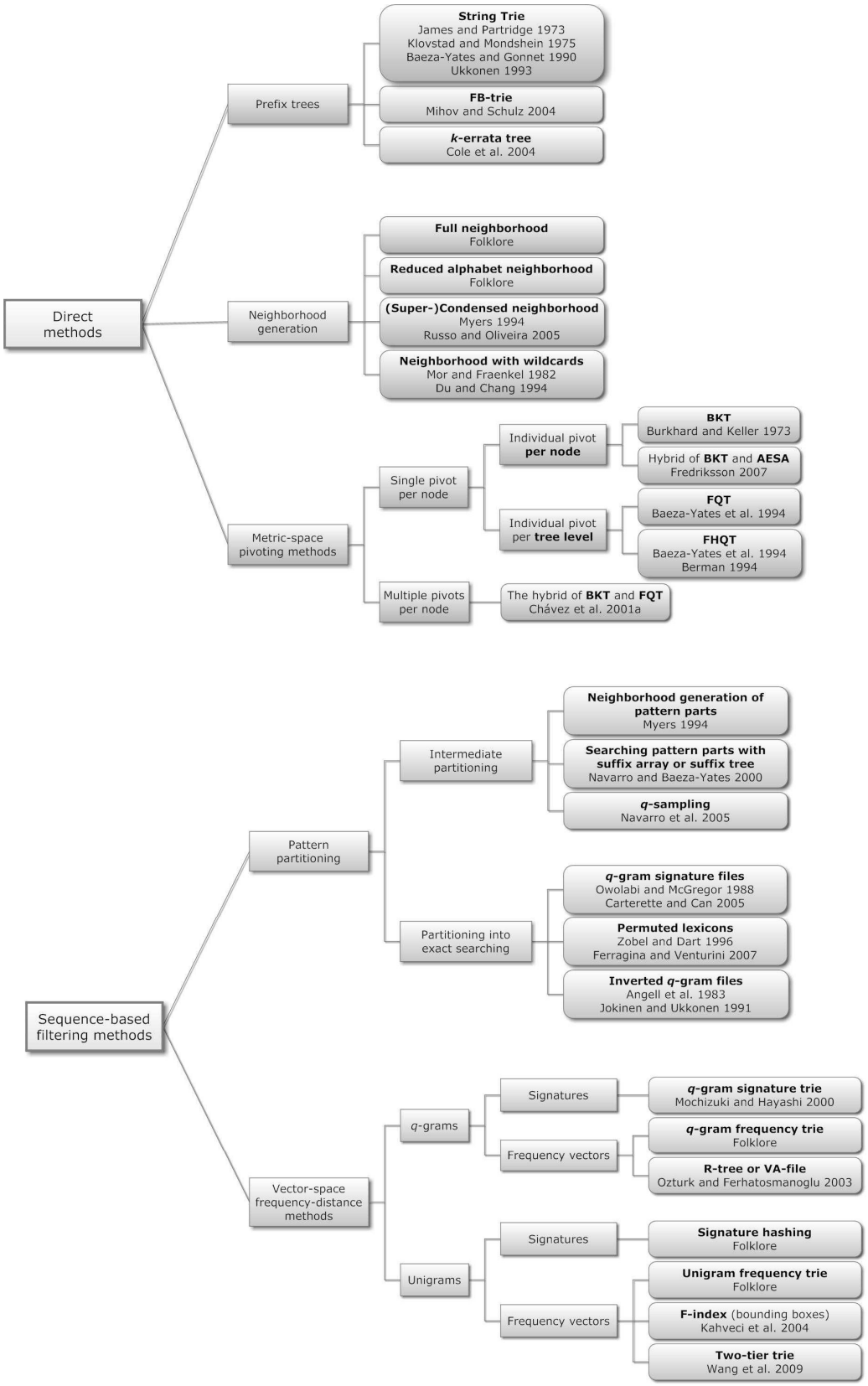


Figure 3. *Taxonomy of search methods*

2.3. Trees & Tries

Before explaining about the different tree-based data structures , we`ll talk about the Trie.

2.3.1. Tries

The word "**Trie**" is an excerpt from the word "**retrieval**". Trie is a sorted tree-based data-structure that stores the set of strings.[3] It has the number of pointers equal to the number of characters of the alphabet in each node. one of the basic and conceptually in our view most important data structures for pattern matching. A trie is also sometimes referred to as prefix tree since it is a tree data structure and stores common prefixes of strings together. With a trie it is possible to store and/or to index a dictionary (i.e., a set of strings). If the problem instance is one long text, it is possible to index all suffixes of the text in the trie , In the case of a text which is structured as words, it is also possible to index all words of the text in the trie; searches can then be carried out only for prefixes of words and not for arbitrary substrings.

Using trie is beneficial when we have a very long strings having the common prefix and you want to use them as keys for set or map. Trie will provide you with performance, close to the one of hash table (still worse on average, however), but with some benefits, like partial search (currently by prefix substring).

2.3.2. Trie Data-Structure

The trie of a set of strings D is a rooted tree where each edge is labelled with one character of the alphabet Σ. All outgoing edges of a node are required to have different labels. Each node represents the string which is formed by concatenating the characters on the path from the root node (this string is called the path label of the node; the node with path label r is be denoted by r). All descendants of a node with path label r therefore have path labels sharing the common prefix r. A trie of a dictionary D contains nodes for those and only those nodes that represent a prefix of a (or a complete) string ($s_i \in$ D), the root node represents the empty string. A node with a path label s_i that is an element of the underlying dictionary ($s_i \in$ D) is called terminal. To simplify the explanation and to obtain a one-to-one correspondence between leaves and terminal nodes, each string of D is appended with a special terminal character \$ ∈/ Σ that is not contained in the alphabet.[4]

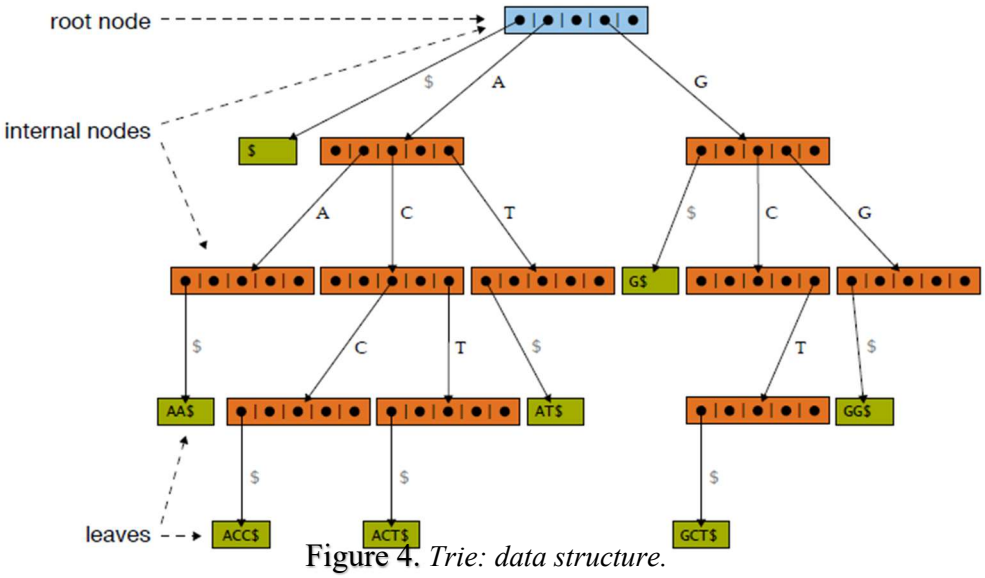


Figure 4. *Trie: data structure.*

A trie for the dictionary $D = \{ \epsilon, \text{“AA”}, \text{“ACC”}, \text{“ACT”}, \text{“AT”}, \text{“GCT”}, \text{“GG”}, \text{“G”} \}$. Each entry $s_i \in D$ is terminated with a special \$ symbol to achieve a one-to-one correspondence between leaves and terminal nodes. In this example, the nodes store child pointers using an array of fixes size (with entries corresponding to the characters A,C,G,T, and \$).

2.4. Direct-Search methods

In order to present our chosen approach in this project , we`ll focus on the methods that are based on tree-related data structure , firstly , in the direct search methods , we`ll explain further about the prefix-trees methods , which are , String-trie, FB-Trie and the k-errata data-structure .

2.4.1. FB-trie

The term FB-trie stand for stands “forward and backward trie”.

this approach to improves the retrieval time , it combines pattern partitioning and tries. It requires a pair of tries: the trie built over the dictionary strings and the trie built over the reversed dictionary strings. The idea of using an index built over the reversed dictionary strings was proposed by Knuth [1973] (see also [Knuth 1997], p. 394).

The FB-trie uses a pair of string tries to satisfy a query. The search algorithm is a two step procedure that divides the pattern into two parts and searches for the first part with $t \leq k/2$ errors and for the second part with $k - t$ errors. In most practical situations, the number of child nodes in a trie decreases quickly as the node level increases. Therefore, the computational cost is dominated by the first step of the search algorithm, where the maximum allowed edit distance is only $t \leq k/2$. However, we are not aware of any rigorous analysis that supports this conjecture.[2]

2.4.2. K-Errata Tree

The approach of Cole et al. [2004] blends partial neighborhood generation with the string trie and treats errors by recursively creating insertion, substitution, and deletions subtrees[2]. For transposition-aware searching, transposition trees should also be created. The method uses a centroid path decomposition to select subtrees that participate in the process: the goal of the decomposition is to exclude subtrees with large numbers of leaves and, thus, to significantly reduce the index size. Coelho and Oliveira [2006] proposed a similar data structure: a dotted suffix tree, which uses only substitutions subtrees.[2]

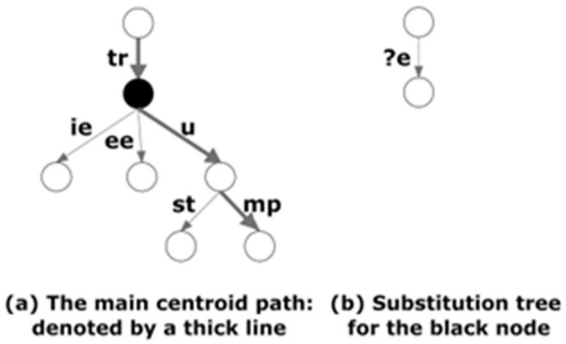


Figure 5. *Example of a centroid path and of a substitution tree. The original trie stores keys: "tree","trie","trump", and "trust".*

Unlike the k-errata tree, a substitution tree created for a node $ includes all subtrees of $. A centroid path starts from a root of a tree. In each node $, the centroid path branches to a subtree with the largest number of leaves (ties are broken arbitrarily). A subtree of $ that does not belong to the centroid path is an off-path subtree. Each off-path subtree is a starting point of another centroid path. It can be further recursively analyzed to identify centroid paths that start from its descendants. The main centroid path is the one that starts from the root of the entire tree. Figure 6, Panel (a), illustrates the concept of centroid path decomposition. Consider, for example, the black node, which is a direct descendant of the root. Its rightmost subtree has two leaves, while all other subtrees have only one

leaf each. Therefore, the main centroid path branches to the rightmost node. This node has two children, which are both leaves. In this case we have a tie and decide to continue the main centroid path to the right. To make the search faster, error trees are created: insertion, deletion, substitution, and possibly transposition trees. In addition, error trees that belong to the same centroid path may be merged to form a hierarchy of trees. This allows one to search for the pattern in more than one tree at the same time. We do not discuss all the details of this procedure and address the reader to the paper by Cole et al. [2004] for a complete description. Instead, we consider an example of a simplified approximate search procedure that allows only substitutions. This procedure relies only on substitution trees (to treat insertions, deletions, and transpositions one needs insertion, deletion, and transposition trees, which are created in a similar way). During indexing, a substitution subtree is created for every internal node $ as follows:
(1) Every off-path subtree of $ is cloned;
(2) The first character of the subtree root label is replaced by the wildcard character "?" that matches any character;
(3) The modified subtrees are merged: see Panel (b) of Figure 6 for an example;
(4) In a general case of $k > 1$, each error subtree is recursively processed to handle remaining $k - 1$ errors.[2]

2.5. Sequence-based filtering methods

we`ll be explaining about two of the q-grams methods , q-gram signature trie and q-gram frequency trie.

a filter is a process that removes some unwanted components or features from a specific data.
The actual of filtering data can be done on almost an attribute or any attribute value found in the database. You can restrict the data you're viewing or searching to items of interest by adding filters. For example, you might limit the huge dataset to anything that is part of your data.
filtering data is a common practice especially for Big Data.
Mapping strings to frequency vectors is an example of a filtering method that consists in using a less computationally expensive algorithm to filter out the majority of data elements.

2.5.1. q-Grams

Many string matching algorithms rely on a fairly large alphabet for good performance. The idea behind using q-grams is to make the alphabet larger. When using q-grams we process q characters as a single character. There are two ways of transforming a string of characters into a string of q-grams. We can either use overlapping q-grams or consecutive q-grams.[9] Data structures using the q-grams of a text (defined below) are very popular indexes for pattern matching on the one hand because they are quite simple (the idea as well as the implementation), and on the other hand because they perform well in practical applications. There are many variants of q-gram based index structures. What they all have in common is that they store some kind of inverted lists: Instead of storing for each text position the q-gram starting at that position, the index stores for each q-gram at which text positions it occurs. This corresponds basically to an inversion of Q pos q from a mapping " "text position ➞ q-gram" to a mapping "q-gram ➞ text position". The various index structures differ in how many and which q-grams are actually indexed and how the information is stored. A q-gram index storing the q-grams of a text allows to efficiently perform exact pattern matching for patterns of length exactly q. The q-gram index can, however, also be used to search for shorter and longer patterns.

2.5.2. q-gram signature trie

An extendible hashing scheme resolves bucket overflows by reorganizing the hash function and file structure locally, so it is very suitable for fast key retrieval of dynamic key sets. However, it cannot search keys that contain a given string as substrings efficiently. In this paper, in order to design this substring search in extendible hashing, signature vectors are introduced as hash values, and a trie structure as an extendible hash table, where each vector is composed by a bit stream.[5]

2.5.2.1.signature vectors

When a substring search is required in extendible hashing all the buckets must be examined. This is essentially due to the function $h_L(x)$ having no information regarding substring x. Therefore this paper proposes a method of reflecting the information about substring x on $h_L(x)$. Then **signature vectors** [3, 8] used in text retrieval strategies are introduced into $h_L(x)$, and the following function SIGN(x) is defined to associate substring x with a signature vector.

[Function SIGN(x)]
Let v be the following signature vector of length m.
$v = b_{m-1} \bullet\bullet\bullet b_i \bullet\bullet\bullet b_2 b_1 b_0$, ($0 \le i < m$, bi is a bit value).
We introduce a function f, such that f (ab) for two adjacent characters ab returns one of the integers 0 through m−1. The following steps define a signature function v = SIGN(x) that returns a signature vector v for string x.
(1) All the bits b_i are set to 0.
(2) b_i = 1 such that i = f (ab) for every adjacent character pair ab in string x.
The function f of SIGN is denoted by SIGN : f (ab).
[End]

Suppose that the signature function SIGN is defined as SIGN : (a + 2b) **mod** 8, where **mod** is a modulo operation. The signature vector SIGN("shima") for key "shima" is determined as follows.

f("sh") = (19 + 2 • 8) **mod** 8 = 35 **mod** 8 = 3, f("hi") = 2
f("im") = 3, f("ma") = 7
SIGN("shima") = 10001100

Figure 6 shows signature vectors for the key set K, and the trie for Figure 6 is illustrated in Figure 7 [5]

x	SIGN(x)	x	SIGN(x)
akita	11100010	nara	00110001
fukui	10101001	niigata	11001011
hiroshima	10101101	okayama	10101000
kagawa	10100010	okinawa	10100011
kochi	00101110	osaka	10100000
kumamoto	10101100	tokushima	10101100
mie	10001000	tokyo	10100100
nagasaki	10100011		

Figure 6. *An example of signature vectors.*

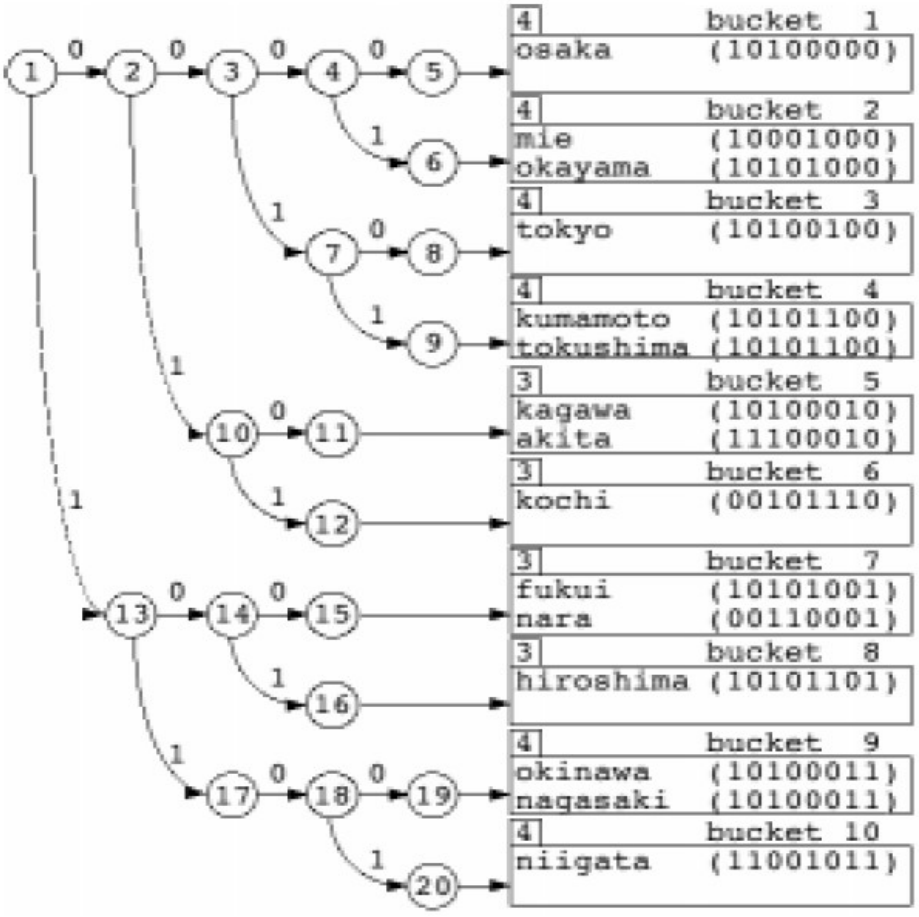


Figure 7. *An example of the extendible hash table constructed by the trie with signature vectors.*

2.5.3. q-gram frequency trie

a given string q, the frequency is how much the string q appears in the database.
q-gram frequency trie is a trie with a frequency value in each node.
The frequency trie is used to facilitate the process of deciding grams, by pruning the trie to decide grams, if the frequency trie is a huge trie with a huge data, that's took more time to prune subtries to decide the grams.

id	string
0	stick
1	stich
2	such
3	stuck

Figure 8. *DB-Strings*

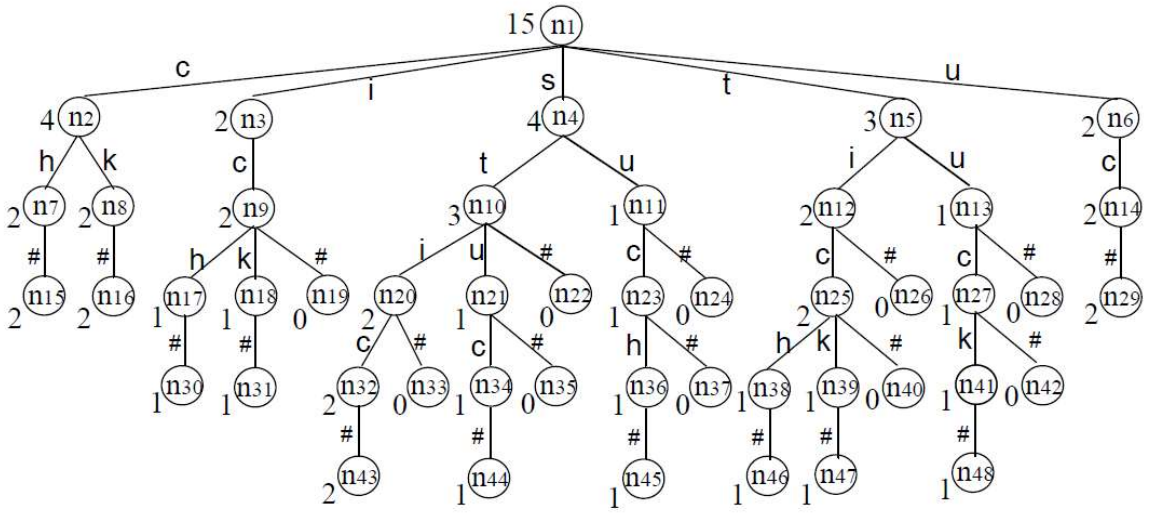


Figure 9. *A gram-frequency trie*

Fig. 9 shows the frequency trie for the strings in Fig. 8 . For instance, the frequency number "2" at node *n*43 means that the gram **stic** occurred 2 times in the strings. The frequency number "3" at node *n*10 means that the gram **st** appears 3 times.
The node has a single leaf child node, *n*22, whose frequency is 0, meaning there is no substring of **st** in the data set without an extended gram of **st**. [6]

2.6. known algorithms for the k-mismatch problem

in this section we`ll present some of the known algorithms for the k-mismatch problem:

2.6.1. Basic Algorithm Of K-Mismatch

In our project we are going to develop a new improvement of the "Basic algorithm" algorithm developed several years ago in our department by Prof. Z. Volkovich and Dr. Z. Frenkel [].

Basic algorithm of K-Mismatch is directed to finding of a Minimal Configuration Set (MCS). MCS is a set of filters concerning to a necessary condition of matching of our word with text at given threshold of similarity. For example, designation "XXX" means a 3 consequent letters word (filter). MCS can also include 'wild cards', denoted by '-'. For example, a designation "X-XX" refers to a three-letter word search, where a match is required in the first letter in the word, mismatch in the second, and then two sequential matches. [7]

According to definition, MCS covers all possible match-mismatch combinations corresponding to the given word size and similarity threshold. The term 'minimal' means that each filter is necessary for this covering.

In order to create our set S, we must know the size of the word (s), and a similarity threshold (t). here we describe an algorithm to build the MCS:

1. Start from the empty set S;
2. Generate all combinations of positions of matches/mismatches for given s and t. Each combination should begin from match.
3. The amount of such combinations can be estimated as $\binom{s-1}{t}$. For example, if s = 20 and t = 8, the amount of the combinations is 75582.
4. For each combination check the presence of a configuration from the set. If such configuration is not present, take some configuration from the combination and add it to the set S.
5. After finishing sections 2-3, we obtain a set of configurations, so that in each combination presents at least one of the configurations from the set. However, there is a possibility that this set can be reduced. So, we should check it.
6. For each configuration C_i from the set S do the follow:
7. If there no such combinations, where C_i is a single conformation from the set S, delete this configuration from the set. [7]

It should be noted that the result of the procedure, i.e., the generated Minimal configuration set, is depending on the order of the generated combinations and configurations. The flowchart of the algorithm is shown in Fig. 10.

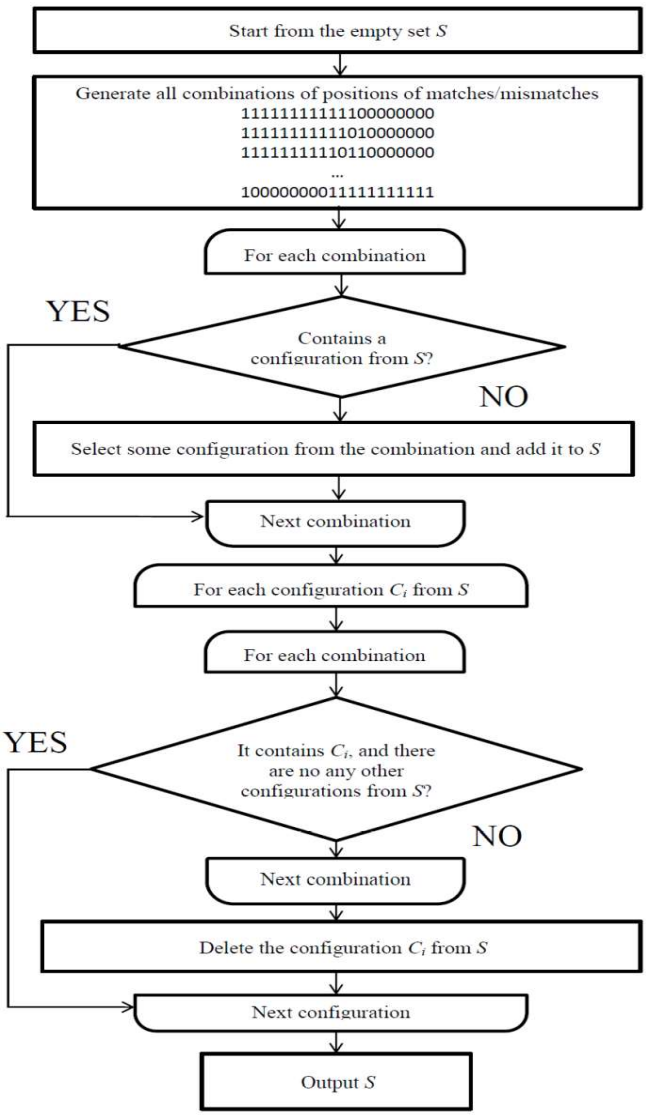


Figure 10. *Flowchart of the 'basic' algorithm*

W=4, |S|=5
S={XXXX, XXX-X, XX-X-X,X--XXX, XX-XX}

Figure 11. *An example of MCS*

Now we will describe an algorithm that searches for words in the MCS:

1. Make the pre-proceeding: Map all words in all configurations from the set S in text T;
2. For the word for search: extract all words in all configurations from the set S set S_w.
3. For each extracted word in S_w: compare the word for search with corresponding places in the text (according to the map in (1)). Select the places, where the similarity meets the requirements. [7]

W = "AARRCCDDEE", and MCS S = {XXX, XX-X},

the extracted set S_w contains 15 words:

1.	AAR	9.	AA-R
2.	ARR	10.	AR-C
3.	RRC	11.	RR-C
4.	RCC	12.	RC-D
5.	CCD	13.	CC-D
6.	CDD	14.	CD-E
7.	DDE	15.	DD-E
8.	DEE		

For the part of text "GGGGGGG**RC**G**D**GGGGGGGGG", contained the word RC-C (in the configuration XX-X), the comparison with W will be only in one place:

GGGGGGG**RC**G**D**GGGGGGGGG
• • • | | • | • • •
AAR**RC**C**D**DEE

Figure 12. *Example of MSC usage in basic algorithm*

2.7. Algorithm requirements

The basic algorithm is actually converting time complexity of the search to mammary space complexity. So, there are two main parameters of the MCS quality: required RAM for mapping and the search speed. The required RAM is defined by the MCS size (each configuration requires a different map), and the speed is defined by amount of the required comparisons of the search term with the text.

a) The occurrence(s) of the produced words (probability to find them in the text) of S_w, Ps_w – should be as small as possible. Generally, we can say that the product $|S_w|\times Ps_w$ should be as small as possible, because this is, in fact, the complexity of the search o($|S_w|\times Ps_w\times n$), where n is size of the text

b) With a small number of configurations, since many configurations |S| may make it impossible to map our text because of a very large map size. As well as a low probability Ps_w may cause and ineffective algorithm, i.e. when a large amount of words in S_w are not present in the text.

Another approach to solve the k-mismatch problem, is finding the positions of the words that match the word to search or words that are similar to it according a threshold as quickly as possible. This approach does not require a large amount of RAM and is notably faster.

This approach is dependent on the text alphabet, string size, threshold and RAM. This approach is based on the probability to find a word in a text, this depends on the number of matches in our MCS (X's), the probability to find a word in a text can be estimated as $\left(\frac{1}{\Sigma}\right)^t$, where Σ is the alphabet size, and t is the number of matches. The more matches we need the bigger the size of MCS $|S|$, and this results in a larger $|S_w|$.

In this approach, in some types of searches, the larger the size of the MCS is, leads to a less amount of words produced. This amount can be estimated as:

$$n_{c(i)} = |W| - |C(i)| + 1$$

$n_{c(i)}$ – Amount of words produced from word |W| in the configuration C(i).

$|W|$ – Size of word W.

$|C(i)|$ – Size of configuration $C(i)$ including wild cards.

This approach focuses on finding the optimal MCS for a given size of a word to search |W| and a similarity threshold T, thus minimizing the search complexity which can be denoted as $|S_w| \cdot P_{s_w}$, where P_{s_w} is the probability to find a word in the text.

Explanation of the set scoring as "amount of words generated from the set of filters for a given search term": consider the first set in Fig.5 S = {XXX, XX-X}. The set was found for search term size |W| = 20, and the similarity threshold T = 60% (i.e. up to 8 mismatches are permitted). The amount of words generated from the set of filters for a given search term is 35: consider a search term: ABCDEFGHIJKLMNOPQRST

a) For the filter XXX, the words will be:

	ABCDEFGHIJKLMNOPQRST
1	ABC
2	BCD
3	CDE
4	DEF
5	EFG
6	FGH
7	GHI
8	HIJ
9	IJK
10	JKL
11	KLM
12	LMN
13	MNO
14	NOP
15	OPQ
16	PQR
17	QRS
18	RST

b) For the filter XX-X, the words will be:

	ABCDEFGHIJKLMNOPQRST
1	AB-D
2	BC-E
3	CD-F
4	DE-G
5	EF-H
6	FG-I
7	GH-J
8	HI-K
9	IJ-L
10	JK-M
11	KL-N
12	LM-O
13	MN-P
14	NO-Q
15	OP-R
16	PQ-S
17	QR-T

Figure 13. *Filters example table*

One of the ways for improvement of the basic algorithm is a heuristic approach providing the MCS of smaller size.

2.8. Improvement Over The Basic K-Mismatch Algorithm

As we have mentioned before, we aspire to make searching faster, faster search is achieved when having longer configurations, and longer configurations means more matches, which in turn leads to bigger size of the MCS and a bigger number of configurations.

The problem is having a big number of configurations to compare to somewhat a small text.

The problem exists due to the need of creating a big MCS, while a big MCS leads to faster search speeds, it takes a lot of memory which in turn causes us to split the text into smaller parts, in each part we check each configuration on every position and the probability of having a successful search is very low and we end up performing a lot of unnecessary comparisons.

We are proposing a method where we exclude these unnecessary comparisons, which subsequently means saving on search times and performing faster searches.

Our proposed method works in this way: We build a configuration tree, a tree that contains all of the configurations produced in the MCS, and after that, we map the words produced by the filters from the text in this configuration tree.

Each leaf of the tree holds the latest letter of each produced word, each letter in the leaf holds an index of its word, the word and the position of the text and its index are saved in a dynamic array that maps all of the words that are found in the tree.

Our approach is based on the basic k-mismatch algorithm but with a major difference in the task of building a data-structure of the map (after building the MCS), we first build a configuration tree, one that contains all of the configurations of the set, for Example:

A configuration tree for the set : {XXX , XX-X}

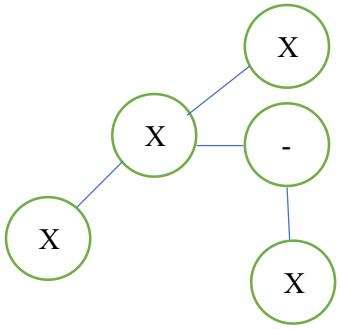


Figure 14. *configuration tree*

The idea behind the tree, is that each “X” contains all of the Text-Alphabet, so “X” itself is not only one node, it's a group of nodes that resembles a level in the tree, each node points to its next and its previous nodes, and is able to hold an index.

The task of mapping the words is basically indexing each node (letter) to its previous/next node, the first letter doesn`t have a previous node, and the last letter doesn`t have a next node but has an index.

Each time a node has an index, the index points to its place in the array, writes the appropriate word by tracking down the path of the word, and saves the position of the word in the text.

Example of mapping the text : “ABBA”

With the set : {XXX , XX-X} , Words that get produced : {ABB , AB-A , BBA}

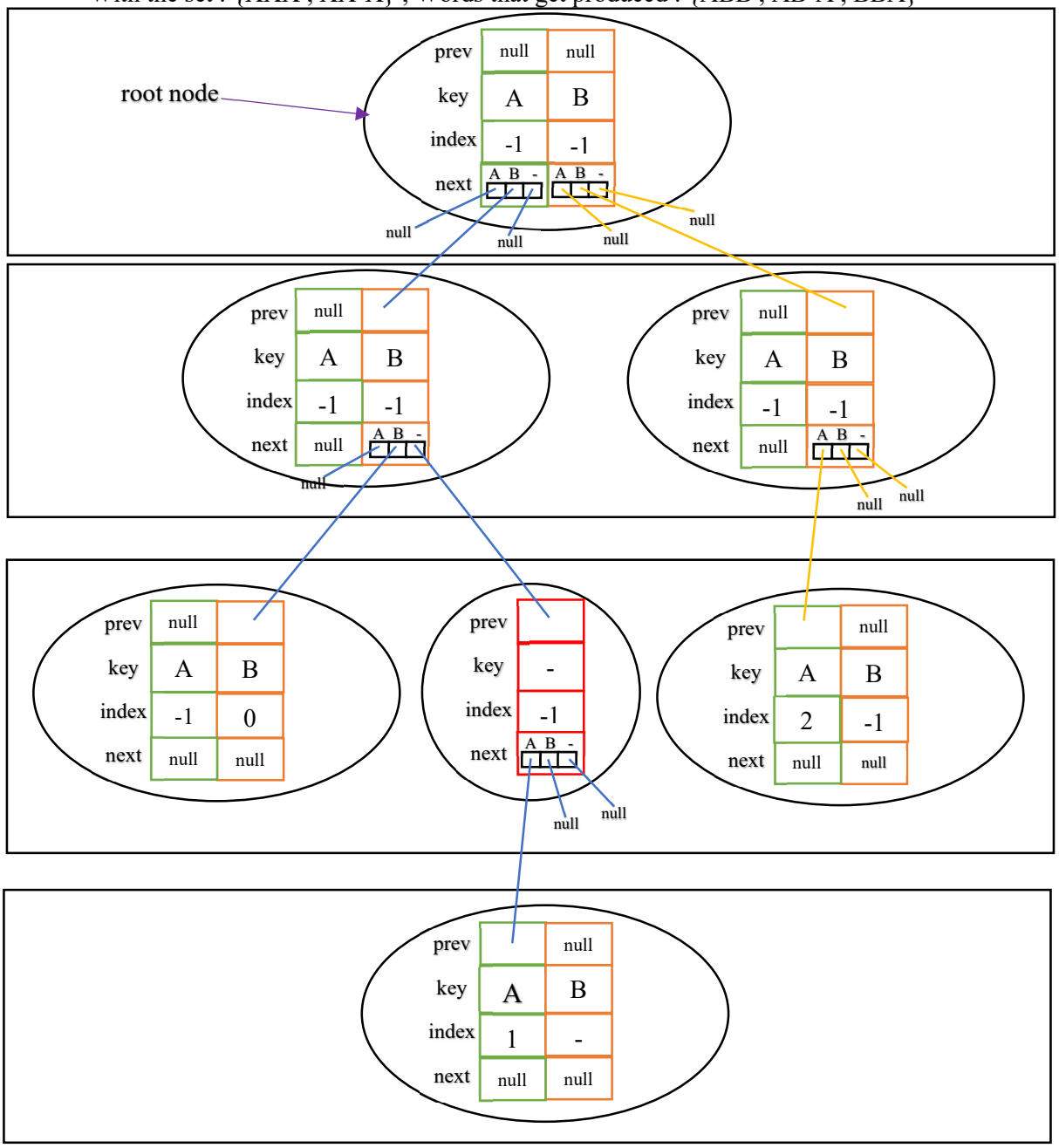


Figure 15. *configuration tree after mapping the words*

word	index	position
ABB	0	1
AB-A	1	1
BBA	2	2

Figure 16. *Index array*

The benefit of using a tree as a map for searching is that in the process of searching for a specific word, when we find that a certain path doesn`t contain the word we are searching for we can automatically eliminate the possibility of finding the word in any filter that has the same path beginning, and this thing ensures faster search times, for it can eliminate many un-needed searches with different filters.

The advantage of using a tree for searching is that on each level we eliminate the branch that we didn't go into, and thus we save of searching time and we also need less computing power.

Next to the built dynamic tree is a dynamic array which stores the words being produced by the text filters with the index number (it`s order in the produced words), and it`s position in the text.

2.8.1. Example Runs

Based on the previous example, the input is the same input:

text : "ABBA"

with the set : {XXX , XX-X}
Then the words that get produced : {ABB , AB-A , BBA}
and therefore the configuration tree is the same tree that shown in Fig.15 .

2.8.2. Successful search

Now we want to search the word "BBA" in the configuration tree (the path with the bold green line):

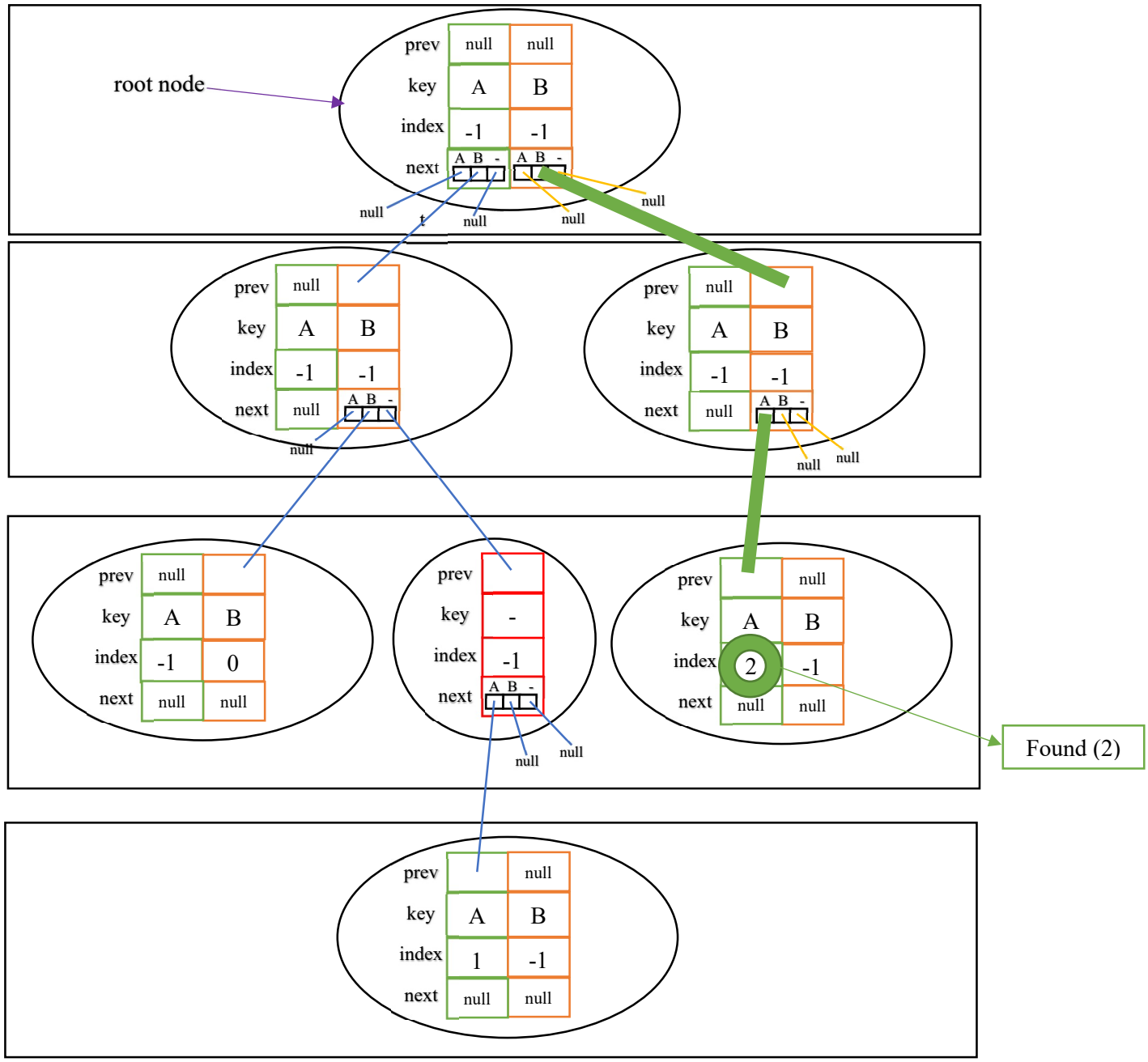


Figure 17. *Searching in tree for the word "BBA"*

In Fig.17, we started from the root node and we noticed that the next pointer of the letter 'B' isn't null. Now "B" is our first letter. Now we check if the next pointer of the current node points to the letter 'B', and we find that the condition is met, now we check if the next pointer of the second node of the letter 'B' isn't null, and only then we could say that we have found "BB". now we continue to the next letter 'A' following the same principle, and we that we found an 'A' after that, then we have "BBA", we now want to check if the word is in the text "ABBA", for that we must check if the index of the third node of the letter 'A' is not equal to (-1), and the condition is met, then this index is an address in the Index array (shown in fig.16), then the word "BBA" is found in the text "ABBA".

2.8.3. Unsuccessful search

Now we want to search the word "BBB" in the configuration tree (the path with the bold red line) :

Figure 18. *Searching in tree for the word "BBB"*

In Fig.18, we started from the root node and we noticed that the next pointer of the letter 'B' isn't null. Now "B" is our first letter. Now we check if the next pointer of the current node points to the letter 'B', and we find that the condition is met, now we check if the next pointer of the second node of the letter 'B' isn't null, and only then we could say that we have found "BB". now we continue to the next letter 'B' following the same principle, and we find that we don't have a B in the next index, now the condition isn't met and we have reached an unsuccessful search and the word "BBB" isn't found in the text "ABBA".

2.8.4. The flowchart of our algorithm

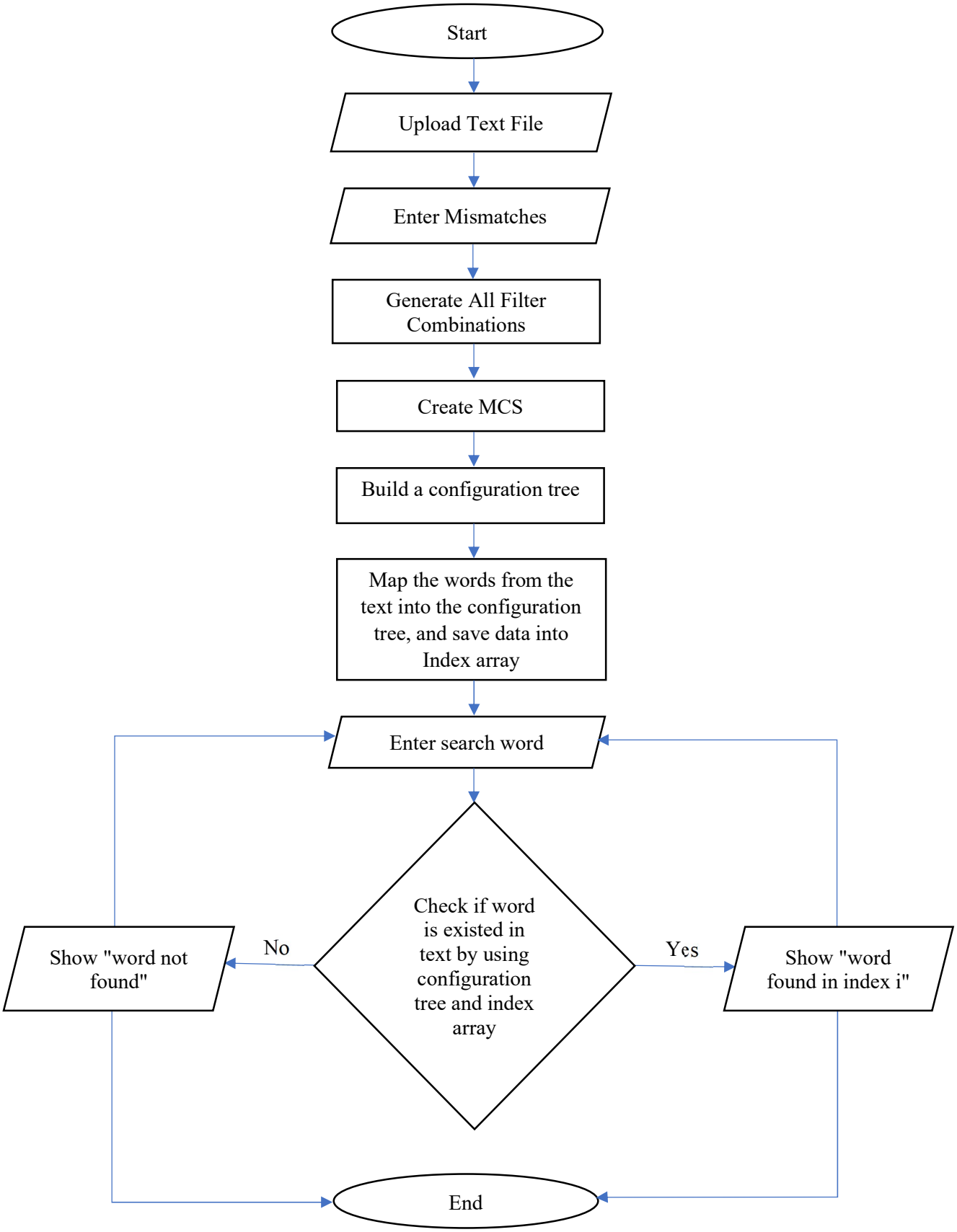


Figure 19. *Flowchart of our algorithm*

3. *EXPECTED ACHIEVEMENTS*

In this project we want to make a significant improvement of the Basic algorithm for the k-mismatch problem.
The idea is creating a data-structure that contains all of the words combinations from the MCS and to use it as a map.
The advantage of the used data structure is that it can track combinations that have a similar prefix (the length of the prefix is unknown), the advantage is that when the algorithm finds that a certain word isn`t found when comparing it to an existing combination, it can withdraw from searching all of the other combinations that have the same prefix (the prefix of the searched used combination until we reached a null pointer).
As a result, our improvement minimizes many un-needed searches from happening, which reduces a lot of time in the process of searching, and reduces the effort put by the CPU.
We expect this data-structure to be used in every application of the k-mismatch algorithm as it improves on the algorithm's CPU time while maintaining all the other aspects of the original method.
As a result, we expect a dramatic increase in search speeds when long configurations are used for relatively short texts.

4. *ENGINEERING and RESEARCH PROCESS*

4.1. PROCESS :

At first, we had to understand in depth the basic algorithm: how it works, what parameters it has and what its purpose is by searching for articles on algorithm.
We started thinking and trying to find which parameters affect the algorithm through experiments to see what parameters affect the efficiency and speed of the algorithm the most.
Using the calculations and experiments that we had done, we started developing our algorithm
and designing software engineering diagrams.
After we finish this part, we will check whether our proposed algorithm is better and faster than basic k-mismatch algorithm in the following ways:

1. We are going to develop our algorithm using C++ programming language.
2. We are going to make sure that our algorithm is working properly.
3. We will calculate our time and compare it to the basic k-mismatch algorithm.

Our project is based a complicated algorithm for the k-mismatch search by q-gram filtering approach. So, the task requires a deep understanding of the algorithm with all corresponding theoretical basis. Our goal is to achieve a significant improvement of the speed of search, that requires finding of an original solution and involvement a lot of energy and creativity.
The main idea of the improvement is to map the word combinations (all of the words in the text) differently by using a dynamic tree-based data-structure with the assist of a dynamic-array. The data structure will improve search speeds by avoiding un-needed searches.
Actually, we are going to do:

1. Write and test the code
2. Build a set of long configuration MCSs for different search parameters
3. Create texts for search and words for queries
4. Run the search by previous and new algorithms and compare results.

4.2. PRODUCT

4.2.1. PRELIMINARY SOFTWARE ENGINEERING DOCUMENTS

4.2.1.1. Requirements (USE-CASE DIAGRAM):

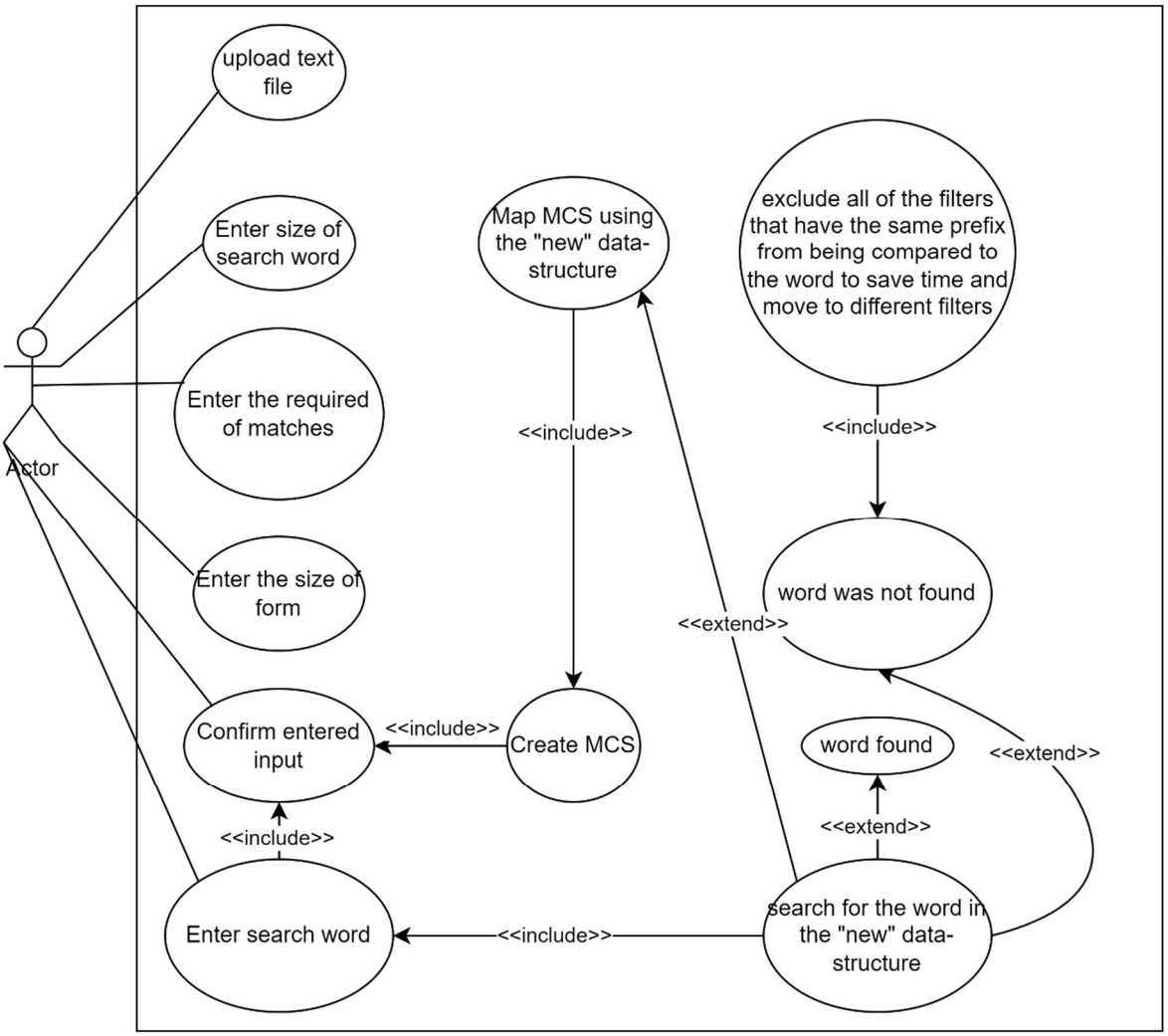


Figure 20. *Use case diagram*

4.2.1.2. Design (GUI, UML diagrams)

- **GUI** :

i. This will be our "MAIN PAGE SCREEN", where the user can choose the text file in which he wants to search for a word, he attach his text file, choose the amount of matches, size of form, and size of the search word :

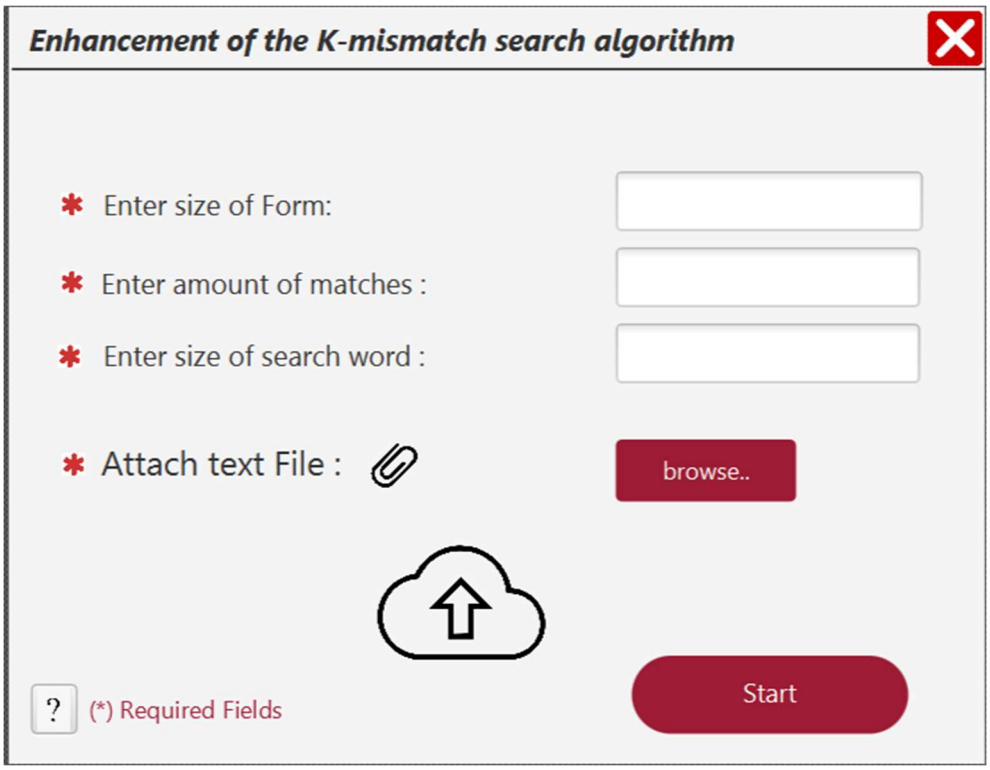


Figure 21. *Main GUI (Home page)*

ii. This will be our "LOADING PAGE SCREEN", in this process we will prepare our program : Loading data from the text file and Creating MCS and Mapping the words from the text…
after this important process, our program will be ready to receive a search word from the user :

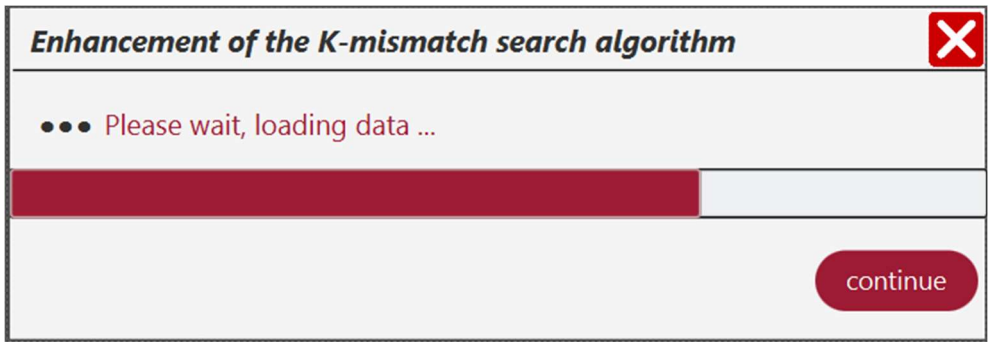


Figure 22. *Loading screen (Loading page)*

iii. This will be our "SEARCH PAGE SCREEN", in this screen window the program waiting a search word from the user to check if the word is existed in the file text, and the user will insert a search word :

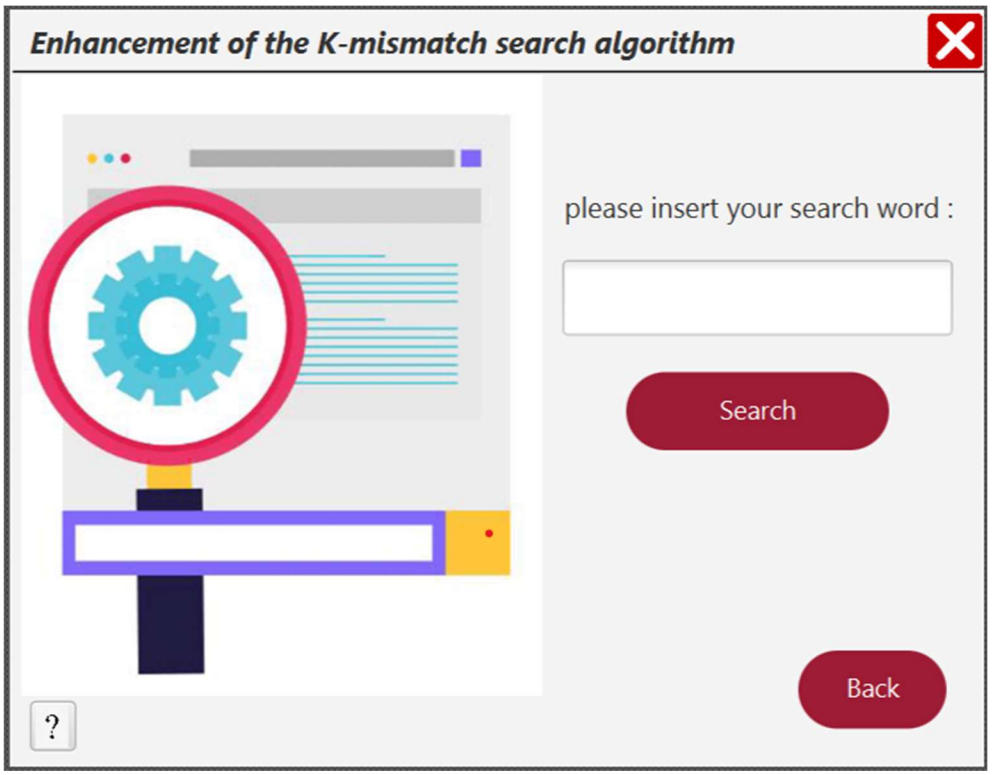


Figure 23. *Search screen*

iv. These will be our "RESULT SEARCH PAGES SCREEN", this result page will be one from the following windows :
1. For a positive result : the search word is found in the file text, and it's shown in Fig. 24.
2. For a negative result : the search word is not found in the file text, and it's shown in Fig. 25.

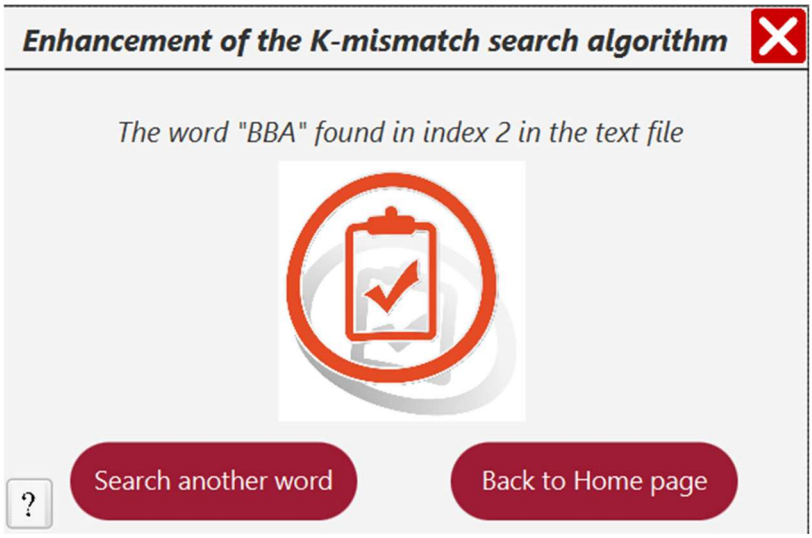


Figure 24. *Positive result of search*

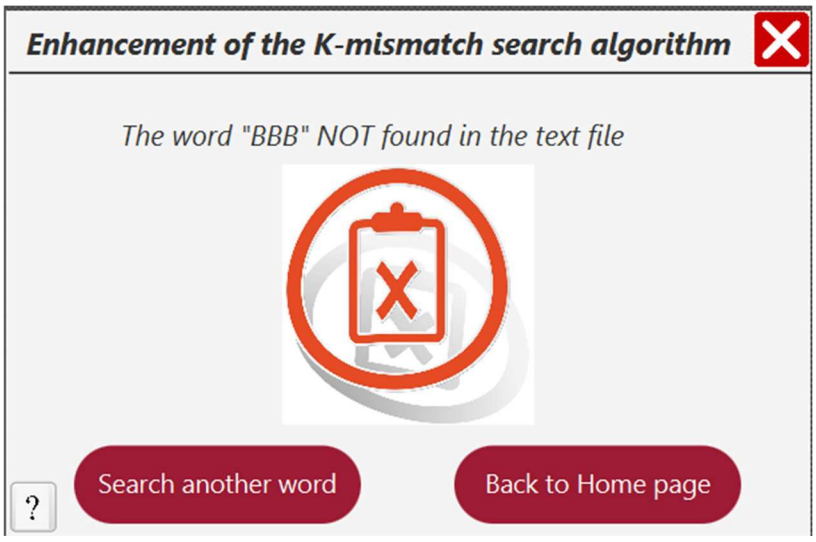


Figure 25. *Negative result of search*

- **UML diagrams (CLASS DIAGRAM):**

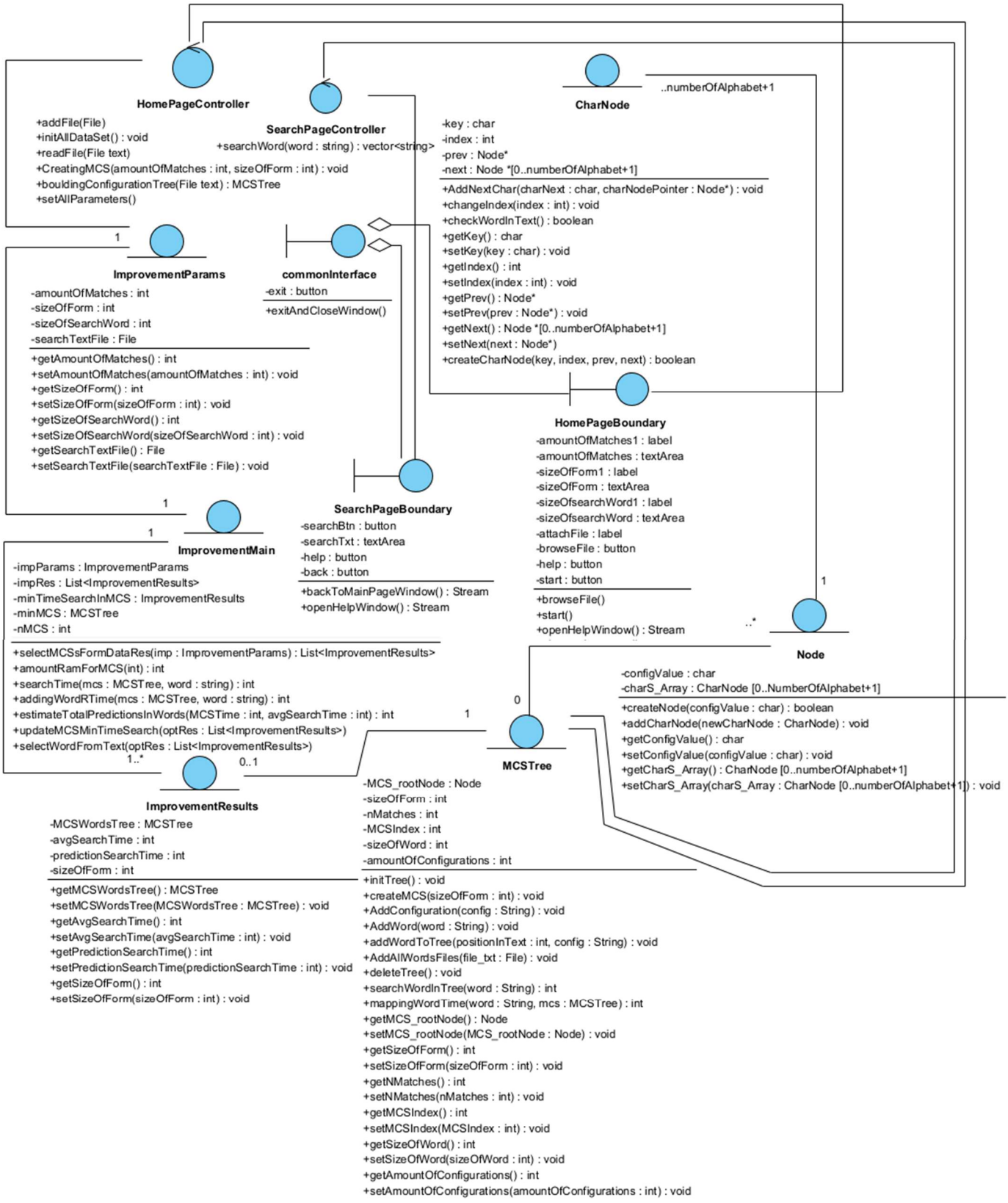


Figure 26. *Class diagram*

5. *EVALUATION and VERIFICATION PLAN*

A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code, does it do what it is supposed to do and do what it needs to do.

A. GUI components will be tested manually for verifying functionality and correctness of actions:

Table 1: *Main GUI (Home page that shown in fig.21)*

Test name	Description	Expected results	Comments
AddFile1	Press "browse.." button	Open browse file window	
Start1	Press "Start" button	Move to "Loading Screen"	All the inputs are legal, The program received all of them, and program shows the" Loading screen" and started the loading process .
Start2	Press "Start" button	Show message: "Please add a text file"	The user didn't add a text file
Start3	Press "Start" button	Show message: "Please Enter amount of matches"	"amount of matches" field in EMPTY
Start4	Press "Start" button	Show message: "Please Enter size of form"	"size of Form " field in EMPTY
Start5	Press "Start" button	Show message: "Please Enter size of search word"	"size of search word" field in EMPTY
Start6	Press "Start" button	Show message: "illegal amount! Please Enter legal amount of matches"	"amount of matches" input field is illegal (negative input or the input isn't a number)
Start7	Press "Start" button	Show message: " illegal size! Please Enter legal size of form"	"size of Form" input field is illegal (negative input or the input isn't a number)
Start8	Press "Start" button	Show message: " illegal size! Please Enter legal size of search word"	"size of search word" input field is illegal (negative input or the input isn't a number)
Exit1	Press " X " button	The program will ended and the window closed	

Table 2: *Loading screen GUI (that shown in fig.22)*

Test name	Description	Expected results	Comments
Continue1	Press "continue" button	Move to "Search Screen"	The program ended its' loading process and it's ready to receive a word from the user, and the program shows the "Search Screen"
Continue1	Press "continue" button	Show message: "Please wait until the program ended its' loading process ! "	The program is still in loading process..
Exit1	Press "X" button	The program close the window	The program ends

Table 3: *Search screen GUI (that shown in fig.23)*

Test name	Description	Expected results	Comments
Search1	Press "Search" button	Move to "Positive result of search screen"	The program found the search word in the text file, and move to the positive result of search screen
Search2	Press "Search" button	Move to "Negative result of search screen"	The program doesn't find the search word in the text file, and move to the negative result of search screen
Search3	Press "Search" button	Show message: "Please Enter a search word"	"search word" field in EMPTY
Search4	Press "Search" button	Show message: "Please Enter a legal search word"	"search word" field in include illegal input (numbers, or any kind of sign …)
Back1	Press "Back" button	Move to "Main GUI screen"	The Program will go back to the main page ..
Exit1	Press " X " button	The program will ended and the window closed	

Table 4: *Positive result of search GUI (that shown in fig.24)*

Test name	Description	Expected results	Comments
Search another word	Press "Search another word" button	Move to *"Search screen"*	The program found the search word in the text file, and the user want to search another word, the program move to the *"Search screen* "
BackToHomePage	Press " Back To Home Page " button	Move to "Main GUI screen"	The Program will go back to the main page ..
Exit1	Press " X " button	The program will ended and the window closed	

Table 5: *Negative result of search GUI (that shown in fig.25)*

Test name	Description	Expected results	Comments
Search another word	Press "Search another word" button	Move to *"Search screen"*	The program doesn't find the search word in the text file, and the user want to search another word, the program move to the *"Search screen* "
BackToHomePage	Press " Back To Home Page " button	Move to "Main GUI screen"	The Program will go back to the main page ..
Exit1	Press " X " button	The program will ended and the window closed	

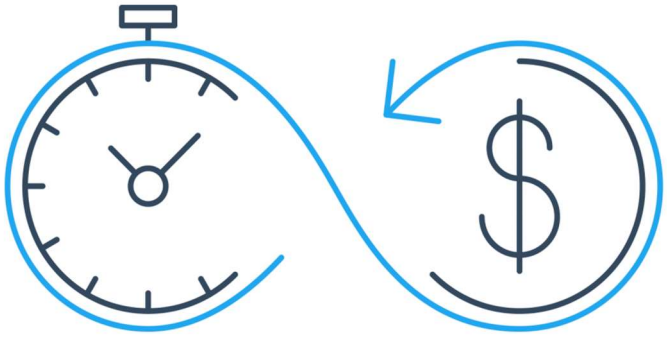
B. Basic tests for our improvement :

Table 6: *Basic Tests for our improvement*

Test name	Description	Expected results
Successful Search	User Search a word found in the text	Show message "word found in index i"
Unsuccessful Search	User Search a word not found in the text	Show message "word NOT found!"
Evaluate search speed	compare search speeds between basic algorithm and our algorithm	Our algorithm is faster by more than 1000 times (we predict that will be 10,000 times)

C. Divide the algorithm to small functions and check each separately. In other word, we will take a function and identify different results between expected and actual results :

- We will test the selected MCSs are correct and respond to the user's input.
- We will test the legality of the input entered by the user and check it's correctness.
- To test the time of mapping and search time , we will run the algorithm few times on the same input and check if the output of algorithm is similar in all cases.
- We will test that the final prediction time is relevant.



6. *REFERENCES*

1. https://en.wikipedia.org/wiki/Approximate_string_matching

2. Leonid Boytsov. "*Indexing Methods for Approximate Dictionary Searching:Comparative Analysis*".

3. https://www.vdube.com/articles/read/what-is-trie-what-are-its-properties_891.html

4. Johannes Krugel. "*Approximate Pattern Matching with Index Structures*".

5. HISATOSHI MOCHIZUKI and YOSHITAKA HAYASHI . "*An Efficient Retrieval Algorithm of Compound Words Using Extendible Hashing*".

6. Chen Li, Bin Wang and Xiaochun Yang. "*VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams*".

7. Esko Ukkonen. "*Approximate string-matching with q-grams and maximal matches*".

8. Z. Frenkel and Z. Volkovich, "A new approach for solution of the k-mismatch search problem".

9. LEENA SALMELA, JORMA TARHIO and JARI KYTOJOKI. "*Multi-Pattern String Matching with q-Grams*".