Software Engineering Department
Ort Braude College
Course 61766: Extended Project in Software Engineering

Enhanced K-Mismatch Search Engine with Insertions/Deletions(Indels)

Karmiel – February 2019

Authors:

Yaron Mamane - 301542643
Dima Spektor - 319286738

Supervisor:

Dr. Zakharia Frenkel

Abstract. *As years pass, we are adding more information and data bases are become larger. The search might take long time, what rises the need for efficient and fast search algorithms. In addition we want the algorithm will support finding a word similar to the word of search, including insertion or deletion of characters (word with different length).*

Goals of the new project:

1. *Adaptation of the idea of gapped q-gram filters for the Levenshtein distance case.*
2. *Creation of the correspondent program and comparison of its power with the theoretical prediction.*
3. *Comparison of our Levenshtein distance similarity search tools with the existent programs.*

List of keywords*: Levenshtein distance, K-mismatch problem, q-gram filters.*

CONTENTS

1. INTRODUCTION

The k-mismatch search problem is related with development of an algorithm for quick finding of 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, i.e. amount of matches or mismatches. There is a wide range of conditions such as similarity threshold, sizes of text, alphabet and the searched words, when only the naïve algorithm can be applied for search. That means, 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 we 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 that is practically impossible.

Recently, the idea of gapped q-gram filters was generalized for quick solution of k-mismatch problem under a wide range of conditions. The algorithm allows the best use of the available computer resources for achieving of the maximal speed of search. The algorithm is very adaptable and can be applied for a wide range of circumstances, where the k-mismatch search is used.

In the current project, we are going to make an extension of the "k-mismatch" method for the task of quick search of the similar terms, where insertions and deletions of letters are also permitted (Levenshtein distance). Although, there is an exhaustive approach providing the Levenshtein distance similarity search based on the Hamming distance similarity searches, our approach is suggesting a 'direct' calculation of filters for the Levenshtein distance case. We expect, it will be much quicker, but requiring a little bit more RAM. This new tool will be also compared with the existing programs.

2. BACKGROUND AND RELATED WORK

2.1. Naïve Algorithm Of K Mismatch

The basic algorithm is got input of searched word S=s1s2…sm and text T=t1t2…tm and using hamming distance checks all the number of locations j where Sj ≠ Tj and also the hamming distance smaller or equal than number k.

For example of the hamming distance , let's take the word S=ABCABC and the text T = ABBAAC.

The hamming distance between them is 2 (Marked on red the locations j where Sj ≠ Tj).

Now let's see the running of the Naïve Algorithm – we will take for input pattern P=p1p2….pm and text T=t1t2….tm and we will check for each i in T the Ham(P, $t_i t_{i+1} \ldots t_{i+m-1}$) that smaller or equal than k.

For example, let's take pattern P = ABBAAC and text T=ABCAABCAC…. and k = 2.

First we will check the hamming distance between P and first 6 letters of T:

P = ABBAAC

T = **ABCAAB**CAC

We will got that Ham(P,T1) is 2(T1 is bold, Marked on red the locations j where Pj ≠ Tj).

next we will check the hamming distance between P and next 6 letters of T:

P =   ABBAAC

T = A**BCAABC**AC

We will got that Ham(P,T2) is 4(T2 is bold, Marked on red the locations j where Pj ≠ Tj).

next we will check the hamming distance between P and next 6 letters of T:

P =   ABBAAC

T = AB**CAABCA**C

We will got that Ham(P,T3) is 6(T3 is bold, Marked on red the locations j where Pj ≠ Tj).

Finally we will check the hamming distance between P and next 6 letters of T:

P =   ABBAAC

T = ABC**AABCAC**

We will got that Ham(P,T4) is 2(T4 is bold, Marked on red the locations j where Pj ≠ Tj).

We will check now which hamming distance of T1,T2,T3,T4 is smaller or equal than k = 2.

P = ABBAAC

T1 = ABCAABCAC

T4 = ABCAABCAC

We got that Ham(P,T1) and Ham(P,T4) is equal to 2 and this places on text will be are output(marked on red and green).

The compare using hamming distance between searched word with the text will be searched on every place of the text.

The running time is O(nm) when n is the size of text T and m is the size of pattern P.

The running time is high can not be accepted on some circumstances.

2.2. Basic Algorithm Of K Mismatch

The basic algorithm is creates a Minimal Configuration Set(MCS). Configuration are strings that have 'X' or '-'.

An 'X' that means a match at that place and wild card '-' means that there is no match and we can ignore.

Example: one three-letter word with wild card in the third position will be marked as "XX-X".

We need for generating the MCS – s to select a size of the word and select the similarity threshold – t.

On the MCS s there has to be a one-word correspondent to a form from the set s [1].

Here we describe the algorithm for building of such set:

1. We will tart from the empty set S;
2. Here we generate all combinations of positions of matches/mismatches for given s and t. Every combination will begin from match.
   The amount of such combinations can be estimated as $\binom{s-1}{t}$.
3. For each combination we will check the presence of a configuration from the set. If such configuration does not present, we will take some configuration from the combination and add it to the set S.
4. Only after finishing of sections 2-3, we have a set of configurations, so that in each combination presents at least one of the configurations from the set. We will also check if there is a possibility that this set can be reduced.
5. For every configuration C_i from the set S:
   If there no such combinations, where C_i is a single conformation from the set S, delete this configuration from the set [1].

The generation of creating MCS is depending on the order of the generated combinations and configurations that we got after building the MCS. We can see a flowchart of the algorithm as shown at **figure 1**.

The 'basic' approach of application of the MCS to the word search is follow:

1. Do the pre-proceeding, map all words in all configurations from the set S in text T.

2. To search a word: extract all words in all configurations from the set S set S_w.

3. For each word we extracted from S_w: compare the word for search with corresponding places in the text , we will check that on the map we created. Select the places, where the similarity meets the requirements [1].

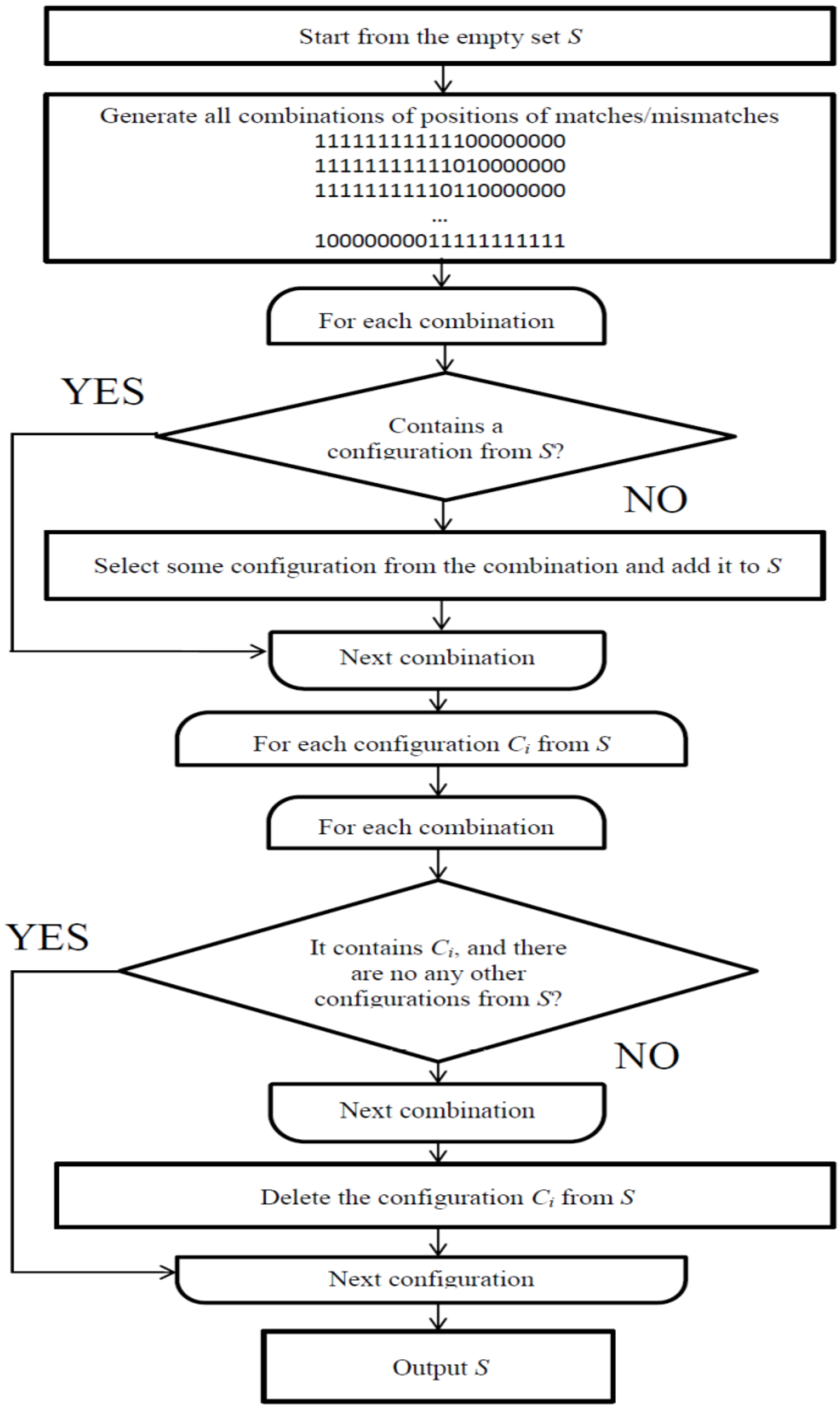

Figure 1. Flowchart of the basic algorithm of k-mismatch

For example, the query is a word that had a 10-letter size. We want to find similar words that containing not more than four mismatches.

If we had insertion/deletion we called it indel,on the basic algorithm of k mismatch we don't use indels.

To implement the algorithm, we need to check all combinations of the matches and mismatches to create a MCS – minimal set of forms.

Each combination contains at least one form from the set.

For example. we will take the word "ABCDEFGHIJ" and the text "ABCDEFQQQQ". We will mark the match as "1", mismatch as "0". We will get the combination "1111110000".

To get the MCS we will check all combinations of matches.

We will use the condition that the form should be selected from the combination – on every combination we will take the first form containing four matches:

Combination of matches	MCS
	{}
1111110000	{"1111"}
1111101000	{"1111"}
1111011000	{"1111"}
1110111000	{"1111","11101"}
1101111000	{"1111","11101"}
1011111000	{"1111","11101"}
1011110100	{"1111","11101"}
1011101100	{"1111","11101"}
1011011100	{"1111","11101","101101"}
1010111100	{"1111","11101","101101"}

Figure 2. Example of creating MCS

2.3. Improvement Of Basic Algorithm Of K Mismatch

This is a new approach to solve the k-mismatch problem, more specifically this approach deals with finding positions of all words that less or equal to given threshold as quickly as possible using as little operative memory as possible. This approach's algorithm is much faster and not required very large amount of RAM.

The goal of this invention is to find the best set of configurations for a given word to search. The search also depends on the following parameters: text, alphabet, string size, similarity threshold and RAM.

The probability to find a word in the text, it is depends on the number of matches ('X') in the configuration. We will mark Σ as the alphabet size and t – number of matches. The probability can be calculated as $\left(\frac{1}{\Sigma}\right)^t$.On our approach the larger number of matches will be between searched word and text, will be the larger size of MCS |S|, and also $|S_w|$.

Nevertheless, in some types of search the larger size of the configuration (including wild cards), the less amount of words is produces. This amount can be calculated 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

The improvement of the basic algorithm should be directed on the finding of the best (for given size of word for search |W| and similarity threshold T) MCS, minimizing the product of search complexity $|Sw| \cdot P_{s_w}$ (P_{s_w}- probability to find a word in the text) and also to available computer memory.

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.2 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 filer 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 3. Filters example table

2.4. Levenshtein Distance

The Levenshtein distance (or edit distance) between two words is a string metric for measuring the smallest number of substitutions, insertions, and deletions of characters to transform one of the words into the other. In this project, we consider the problem of computing the edit distance of a regular language (the set of words accepted by a given finite automaton).

We will use a fixed, but arbitrary, alphabet Σ of ordinary symbols, and the alphabet E of the

(basic) edit operations that depends on Σ.

The empty word over Σ is denoted by λ, that is, $\lambda w = w\lambda = w$, for all words w. The alphabet E consists of all symbols (x/y) such that $x, y \in \Sigma \cup \{\lambda\}$ and at least one of x and y is in Σ.

If (x/y) is in E and x is not equal to y then we call (x/y) an error. We write (λ/λ) for the empty word over the alphabet E. We note that λ is used as a formal symbol in the elements of E. For example, if a and b are in Σ then $(a/\lambda)(a/b) \neq (a/a)(\lambda/b)$

The elements of E* is the edit strings we used on the levenshtein distance.

The input and output parts of edit string $h = (x1/y1),\cdot\cdot,(xn/yn)$, the words from alphabet Σ is

x1, · · , xn and y1, · · , yn, respectively.

We will write inp(h) for input part and out(h) for output part of h.

We also will use weight(h) to check the number of errors in h.

The levenshtein (or edit) between two words u and v , is denoted by dist(u,v) – smallest number of errors(substitutions, insertions, and deletions of symbols)that can transform u to v.

It can be written by the formula:

$$dist(u,v) = min\{weight(h) |\ h \in E*, inp(h) = u, out(h) = v\}$$

For example, for $\Sigma = \{a, b\}$, we have that $dist(ababa, babbb) = 3$ and the edit string

$h = (a/\lambda)(b/b)(a/a)(b/b)(a/b)(\lambda/b)$, is a minimum weight edit string that transforms ababa to babbb. In words, h says that we can use the deletion (a/λ), the substitution (a/b), and the insertion (λ/b) to transform ababa to babbb[2].

The Levenshtein distance has several simple upper and lower bounds. These include:

- It is at most the length of the longer string.
- It is zero if and only if the strings are equal (no insertions, deletions or replaces).
    - If the strings are the same size, the Hamming distance is an upper bound on the edit distance: $ed(a, b) \leq hammingDis(a, b)$.
    - The edit distance between two strings is not lower than the difference between their length:

      $ed(a, b) \geq ||a| - |b||$

2.5. Filtering - Spaced Q-Grams(MCS)

Given a text string of length *n*, a pattern string of length *m*, and a distance *k*, the approximate string matching problem is to find all substrings of the text with a distance *k* of the pattern[3]. The most common distance measure is the *Levenshtein* or *edit distance*, the minimum number of single character insertions, deletions and replacements required to transform one string into the other[3]. This is a widely studied problem with numerous applications in text processing, computational biology, and other areas involving sequential data. In this project we do not cover the indexed version of the problem, which allows the use of a precomputed index of the text[3]. Nevertheless, we still had a problem of finding of the "best" gapped q-grams of filtering because most approaches are not practical for many applications - the search time is too large for long patterns and high distance limit *k*. There are several ways of "mathematical papers" such as [4][5] and [6]. The best practical methods for high *n*, *m* and *k* are based on filtering-type algorithm - Spaced (also known as gapped)Q-Grams(MCS).

A filter is an algorithm that quickly discards large parts of the text using a filter criterion, leaving the interesting parts, the potential match areas, to be checked with a proper (non-indexed) approximate string matching algorithm. These two phases are the filtration phase (MCS in our project) and the verification phase (dynamic programing in our project). A filter is lossless if it never discards a true match, it means it keeps all patterns that are potential with match.

2.6. Dynamic Programming

Dynamic programming is a powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach and simple thinking and the coding part is very easy. The idea is very simple, let's say you have solved a problem with some input, then save the result for using it later, to avoid solving the same problem again. If you can divide the problem into smaller sub-problems and these smaller sub-problems are in turn divided in to smaller ones, and in this process, if you observe some over-lapping sub-problems, then it's probably a case for dynamic programing[7].

We present now an algorithm based on dynamic programing to solve the k-mismatch problem which described above. Although the algorithm is not very efficient, it is one of the most flexible ones to adapt to different distance functions. We present the version that computes the Levenshtein distance[7].

The algorithm based on dynamic programing, if we need to compute *ed*(x,y) (edit distance between x and y), we create matrix $C_{|x|,|y|}$ where $C_{i,j}$ represent the minimum number of operations needed to match $x_{1..i}$ to $y_{1..j}$, this computed as follows:

$$C_{i,0} = i$$
$$C_{0,j} = j$$
$$C_{i,j} = if\left(x_{i=}y_j\right)\, then\; C_{i-1,j-1}$$
$$else\; 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$$

where at the end $C_{|x|,|y|} = ed(x,y)$.

The rational of the formula is: first, $C_{i,0}$ and $C_{0,j}$ represents the edit distance between a string of length i or j and the empty string, clearly i (respectively j) deletions are needed on the long string. For two non-empty strings of length i and j, we assume inductively that all the edit distances between shorter strings have already been computed, and try to convert $x_{1..i}$ into $y_{1..j-1}$[7].

Check the last characters x_i and y_j, if they are equal, we don't need to consider them and the conversion proceeds in the best way we can convert $x_{1..i-1}$ into $y_{1..j}$. If they are not equal, we must deal with them in some way. Following the three allowed operations, we can delete x_i and convert in the best way $x_{1..i-1}$ into $y_{1..j}$, insert y_j at the end of x and convert in the best way $x_{1..i}$ into $y_{1..j-1}$, or replace x_i by y_j and convert in the best way $x_{1..i-1}$ into $y_{1..j-1}$. In all cases the cost is one plus the cost for the rest process (already computed). Notice that the insertions in one string are equivalent to deletions in the other[7].

Here is an example of computing *ed*("survey", "surgery"):

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	0	1	2	3	4	5	6
u	2	1	0	1	2	3	4	5
r	3	2	1	0	1	2	3	4
v	4	3	2	1	1	2	3	4
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	**2**

Figure 4. Example for computing ed("survey", "surgery") through dynamic programming algorithm

The bold entry is the final result.

The algorithm must fill the matrix in such a way that the upper, left, and upper-left neighbors of a cell are computed prior to computing that cell. This can be achieved by row-wise left-to-right traversal or a column-wise top-to-bottom traversal.

The algorithm run in O(|x||y|) in the worst and average case. The space required is O(min(|x||y|), because, in case of column-wise processing, only the previous column must be stored in order to compute the new one, and therefore, we can keep only on column and update it. We can process the matrix row-wise or column-wise so that the space requirement is minimized.

We also can recover the sequence of operations performed to transform x into y, by proceeding from the cell $C_{|x|,|y|}$ to $C_{0,0}$ following the path that matches the update formula, in this case we need to store the complete matrix[7].

3. DESCRIPTION OF OUR METHOD

3.1. Improvement Of Basic Algorithm By Our Method

On our algorithm we will use levenshtein distance in addition to the mismatches with the indels(insertions/deletions) are also permitted. The algorithm will be similar to the hamming distance search with only difference that insertions and deletions also be used.

When we got the indels(insertion/deletion), the algorithm of the MCS building will be very similar as we saw on the explanation of basic algorithm of k mismatch with a difference that for each combination of the matches/mismatches for the insertions/deletions should be also considered.

For example, for the combination of matches considered above:.

ABCDEFGHIJ

1111110000

ABCDEFQQQQ

We will consider the words from the text – Here with insertions:

A-BCDEFGHIJ 10111110000 AQBCDEFQQQQ	AB-CDEFGHIJ 11011110000 ABQCDEFQQQQ	ABC-DEFGHIJ 11101110000 ABCQDEFQQQQ	ABCD-EFGHIJ 11110110000 ABCDQEFQQQQ

Figure 5. Combination of insertions between pattern and text

and here with deletions:

ABCDEFGHIJ 111110000 -BCDEFQQQQ	ABCDEFGHIJ 1 11110000 A-CDEFQQQQ	ABCDEFGHIJ 11 1110000 AB-DEFQQQQ	ABCDEFGHIJ 111 110000 ABC-EFQQQQ

Figure 6. Combination of deletions between pattern and text

We will get for each combination of the matches/mismatches:

matches/ mismatches	Insertion	Deletion
1111110000	10222220000 11022220000 11102220000 11110220000 11111020000	333330000 1 33330000 11 3330000 111 330000 1111 30000 11111 0000
1111101000	10222202000 11022202000 11102202000 11110202000 11111002000	333303000 1 33303000 11 3303000 111 303000 1111 03000 11111 3000
1111011000	10222022000	333033000

	11022022000 11102022000 11110022000 11110102000	1 33033000 11 3033000 111 033000 1111 33000 11110 3000 111101 000

Figure 7. Combination of indels between pattern and text

In this table we use the new rules:

"0" – Mismatch.

"1" – The match between the letters at the same positions in the text and query, relatively to start of the combination.

"2" – The match between the position x in the text with the position x-1 in query - because of insertion.

"3" – The match between the position x in the text with the position x+1 in query – because of deletion.

Summary the new implementation for the Levenshtein distance can be presented as follow:

matches/ mismatches	MCS
	{}
1111110000	{"1111"}
1022222000	{"1111"}
1102222000	{"1111"}
1110222000	{"1111", "11101"}
1111022000	{"1111", "11101"}
1111102000	{"1111", "11101"}
333330000	{"1111", "11101"}
1 3333000	{"1111", "11101"}
11 333000	{"1111", "11101"}
111 33000	{"1111", "11101"}
1111 30000	{"1111", "11101"}
11111 0000	{"1111", "11101"}
1111101000	{"1111", "11101"}
1022220200	{"1111", "11101"}
1102220200	{"1111", "11101"}
1110220200	{"1111", "11101"}
1111020200	{"1111", "11101"}
1111100200	{"1111", "11101"}
333303000	{"1111", "11101"}
1 3330300	{"1111", "11101"}
11 330300	{"1111", "11101"}
111 30300	{"1111", "11101"}
1111 03000	{"1111", "11101"}
11111 3000	{"1111", "11101"}
1111011000	{"1111", "11101"}
1022202200	{"1111", "11101"}

11022022000 11102022000 11110022000 11110102000 333033000 1 33033000 11 3033000 111 033000 1111 33000 11110 3000 111101 000 …	{"1111", "11101", "11011"} {"1111", "11101", "11011"} {"1111", "11101", "11011"} {"1111", "11101", "11011"} {"1111", "11101", "11011"} {"1111", "11101", "11011"} {"1111", "11101", "11011"} {"1111", "11101", "11011"} {"1111", "11101", "11011"} {"1111", "11101", "11011"} {"1111", "11101", "11011"} …

Figure 8. New implementation of MCS

The amount of the forms in the MCS is comparable with the k-mismatch case. Because the form like "11103" is not different from "11101" from implementation. The difference will be in amount of subsequences taken from the query. For example, for the form "11101" the subsequences in the case of k-mismatch will be:

ABCDEFGHIJ ABC-E BCD-F CDE-G DEF-H EFG-I FGH-J

Figure 9. Example to amount of subsequences on "11101"

In the case of the Levenshtein distance for the case of one permitted indel, we should also consider the follow forms: "11102", "11103", "11303" and "13303". We will got the combinations:

"11101"	"11102"	"11103"
ABCDEFGHIJ ABC-E BCD-F CDE-G DEF-H EFG-I FGH-J	ABCDEFGHIJ ABC-D BCD-E CDE-F DEF-G EFG-H FGH-I GHI-J	ABCDEFGHIJ ABC-F BCD-G CDE-H DEF-I EFG-J

Figure 10. Example to subsequences to one permitted indel

"11101"	"11303"	"13303"
ABCDEFGHIJ ABC-E BCD-F CDE-G DEF-H EFG-I FGH-J	ABCDEFGHIJ ABD-F BCE-G CDF-H DEG-I EFH-J	ABCDEFGHIJ ACD-F BDE-G CEF-H DFG-I EGH-J

Figure 11. Example to subsequences to one permitted indel

We consider form of four matches with the k-mismatch, we take six forms in the case of the levenshtein distance with one indel.

After we find the subsequences that we added on the query, we will use the dynamic programing to compute levenshtein distance on for every subsequence we find on query, on the relevant letters from right of the subsequence and relevant letters from left of the subsequence with relevant letters from right of the text and relevant letters from right of the text, we will compute through *ed*(X,Y) – the edit distance between X and Y.

For example: We had the subsequence “CDF-H” on the query “11303”.

So we will computing levenshtein distance through dynamic programming in the pattern word “ABCDEFGHIJ” from relevant letters from right of the subsequence and relevant letters from left of the subsequence with the text “AECDQFQHWR” (Marked on red).

Here is example computing levenshtein distance through dynamic programming on the relevant letters from left of the subsequence with relevant letters from left of text , we will compute through ed(AB, AE) – the edit distance between “AB” and “AE”(the result marked on bold):

		A	B
	0	1	2
A	1	0	1
E	2	1	**1**

Figure 12. Example to computing ed(AB, AE) through dynamic programming

Here is example computing levenshtein distance through dynamic programming on the relevant letters from right of the subsequence with relevant letters from right of text , we will compute through ed(IJ, WR) – the edit distance between “AB” and “AE”(the result marked on bold):

		I	J
	0	1	2
W	1	1	2
R	2	2	**2**

Figure 13. Example to computing ed(IJ, WR) through dynamic programming

We will compute the levenshtein distance between pattern word and text using dynamic programming and through that method we will find the best match between pattern word and text.

4. PLANE OF WORK

We will check a couple of proofs to prove that our new algorithm Enhanced K-Mismatch Search Engine with Insertions/deletions is better than k-mismatch search algorithm:

1. We will develop our algorithm Enhanced K-Mismatch Search Engine with Insertions/deletions on a software platform.

2. We will check if our algorithm Enhanced K-Mismatch Search Engine with Insertions/deletions is working correctly.

3. We will check if our algorithm Enhanced K-Mismatch Search Engine with Insertions/deletions have a faster running time than k-mismatch search algorithm.

5. PRELIMINARY SOFTWARE ENGINEERING DOCUMENTS

5.1. Requirements (Use Case)

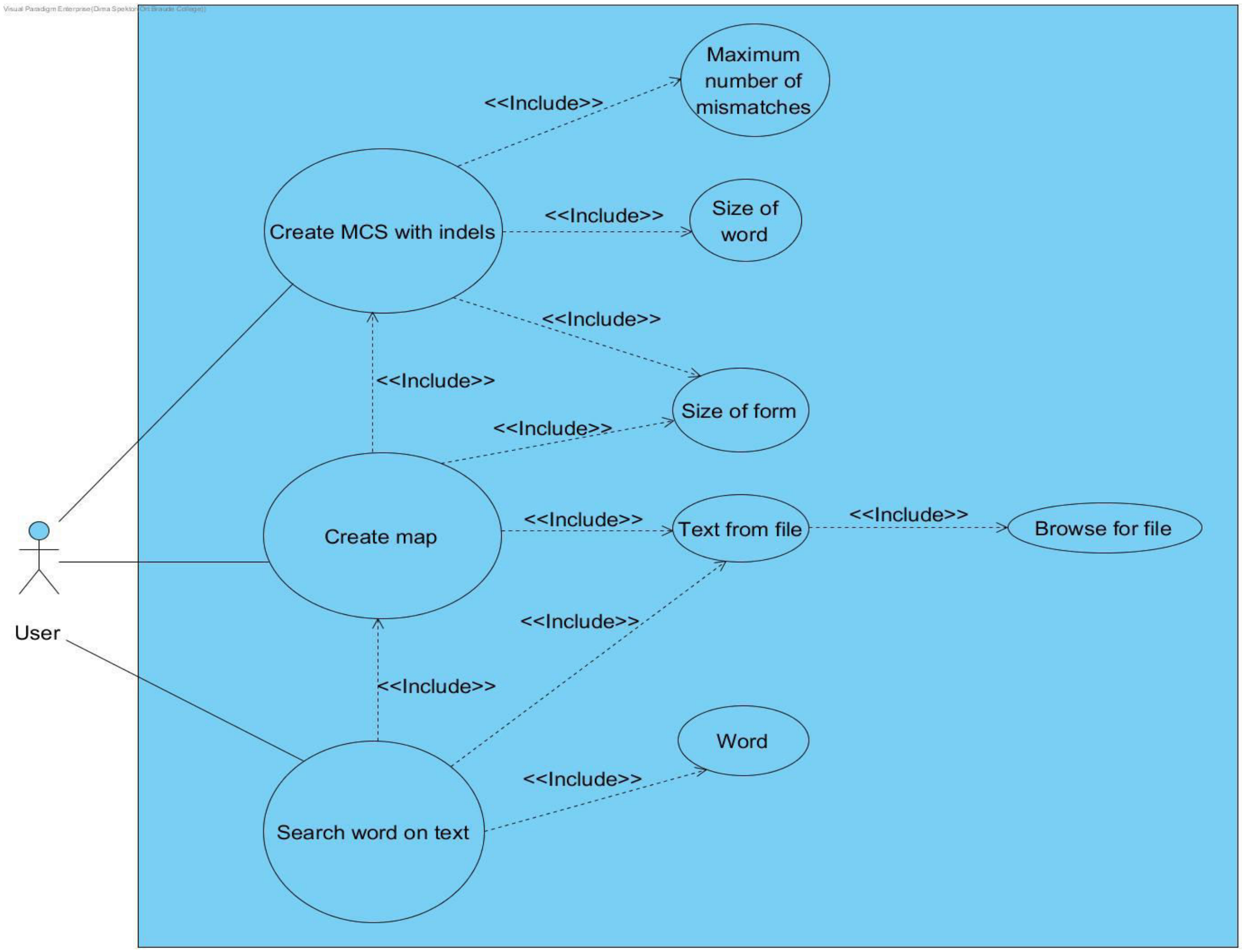

Figure 14. Use case diagram

The program will create MCS with indels by selected maximum number of mismatches , size of word and size of form , after that will create map that will using size of form , text from file and MCS with indels.

The user will search the word on the text using map, text from file and a word.

5.2. Design - GUI

Now we show how to use the main screen.
The user had 5 options:

Define the search parameters:

- Browse a File:

  Here we choose a txt file to get the text - we will search the word in that text through our algorithm.

Define the MCS:

- Select size of form:

  Here we will choose the size of form that will define the size of each form based on mismatch.
- Select size of word:

  Here we will choose the size of word that will define the size of form based on match and mismatch.
- Select size of mismatches:

  Here we will choose the size of mismatches that will be on one form.

Button:

- Run MCS:

  Here we execute the creation of MCS with indels using the size of form , size of word and size of mismatches. It will create the MCS with indels.

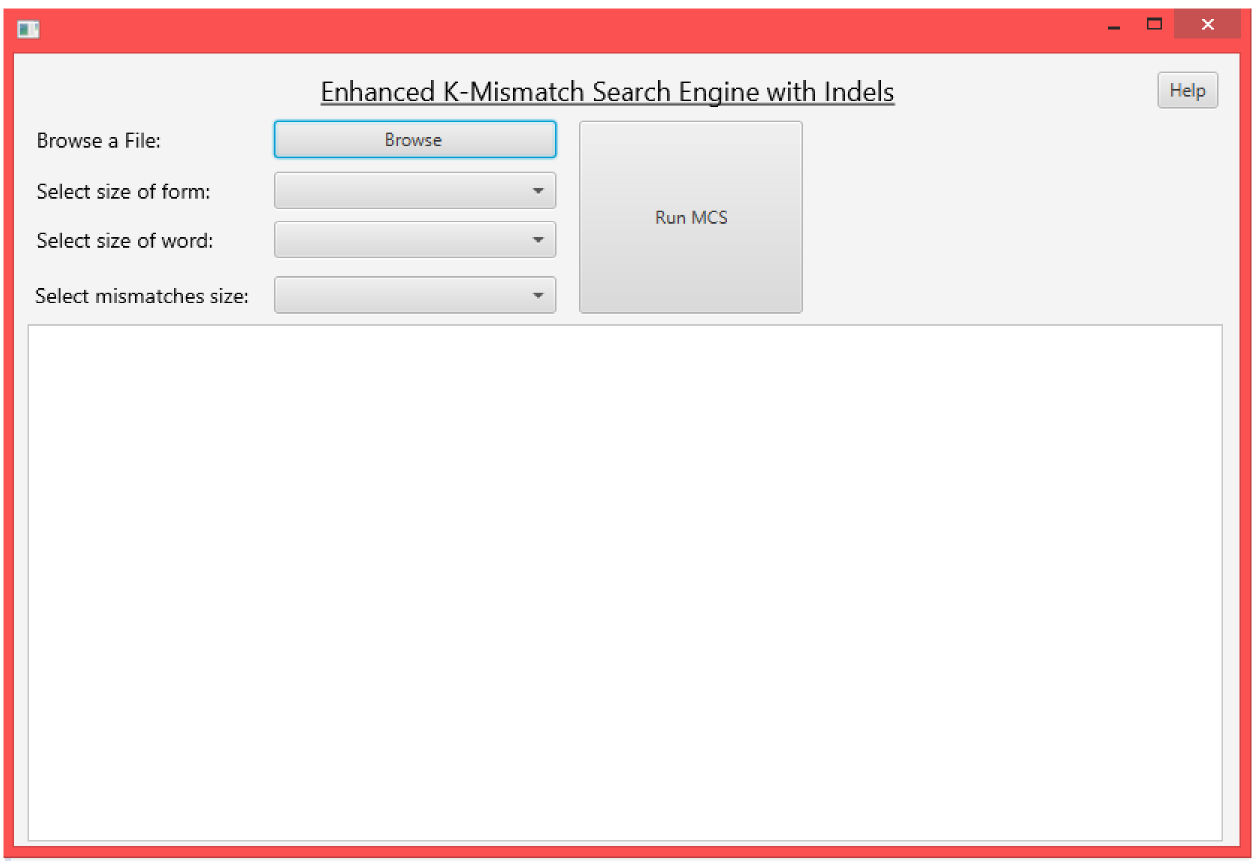

Figure 15. Main screen

Now we show how to use the enhanced k-mismatch search engine with indels screen.
The user had 3 options:

<u>Define the search parameters:</u>

- Add a word for search in text:

  Here we add a word for search in text - we will search that word on the text through our algorithm.

<u>Button:</u>

- Run our algorithm:

  Here we execute the creation of map that we will use to search the word through MCS , text and size form. Next we will search the word on text using the map , word and text. We will show the results of searching in the text area.

- Return to main:

  Here we return to main.

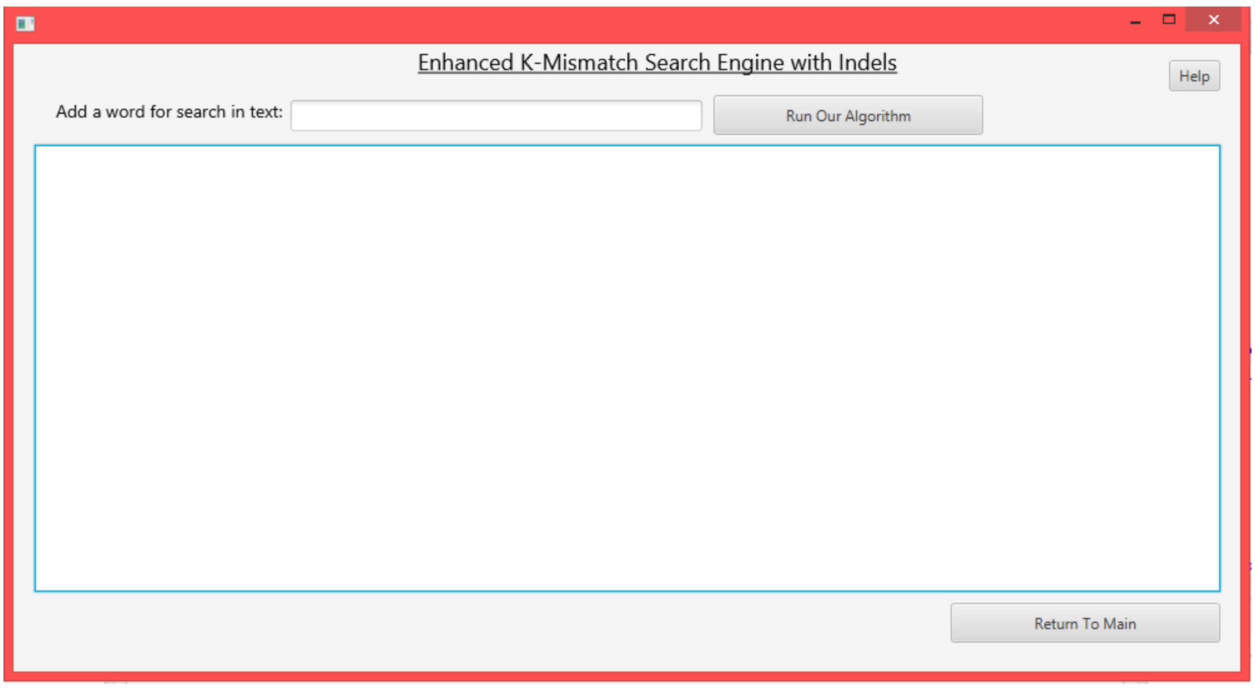

Figure 16. Enhanced k-mismatch search engine with indels screen.

5.3. Design – UML Diagrams

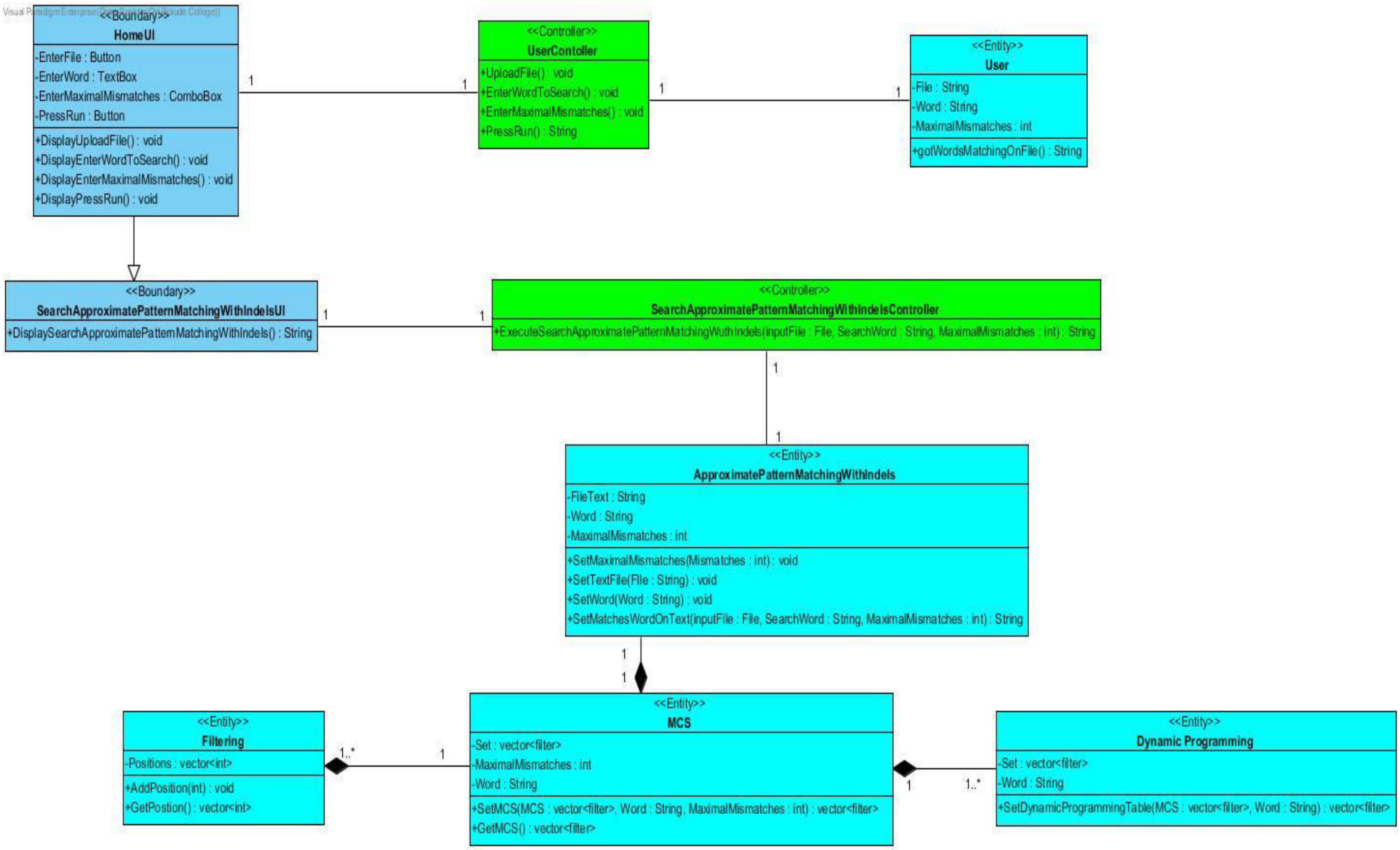

Figure 18. Class diagram

5.4. Testing Plan

Test ID	Description	Expected results	Comments
1	Press “Browse” button	Open browse file window	
2	Select text	The words on text shown on text area	The user select regular text
3	Select empty text and press “Run MCS” button	Show error message: “The text is empty - Please add another text”	The user select text file that it is empty
4	Text is not selected and press “Run MCS” button	Show error message: “Please check missing”	The user did not select text
5	Select size of form and press “Run MCS” button	Show error message: “Please check missing”	The user did not select size of word and mismatches
6	Select size of form and word and press “Run MCS” button”	Show error message: “Please check missing”	The user did not select size of mismatches
7	Select size of form, word and mismatches and press “Run MCS” button	The program move to new window	System creates MCS and map

Table 1. Testing plan – Menu window

Test ID	Description	Expected results	Comments
8	Press "Run Our Algorithm" button	Show results of searching on the text area – the text has no bold letters	System runs search and did not find forms of the word at the text
9	Press "Run Our Algorithm" button	Show results of searching on the text area – the text has bold letters	System runs search and did find forms of the word at the text
10	Press "Run Our Algorithm" button	Show error message: “Please enter word”	The user did not enter a word
11	Press “Return To Main” button	The program returns to main window	

Table 2. Testing plan – Enhanced k-mismatch search engine with indels screen window

5.5. Implementation

The implementation consists of two main parts:

- A front-end part implemented in javaFx.
- The algorithm of enhanced k-mismatch search engine with Indels w implemented by java.

6. RESULTS AND CONCLUSIONS

6.1. Results

We check the running time of large text that his size is 1 MB.

We will define the search parameters: Text that had more than 1000 words and his size is 1 MB and word search “disappoint”.

Lets add parameters and check the running time of the two algorithms we used to search word on a text: K-mismatch search algorithm and enhanced K-Mismatch Search Engine with Indels.

We will first check the running time of enhanced K-Mismatch Search Engine with Indels.

Enhanced K-Mismatch Search Engine with Indels:

We will check first for the parameters that shown on the GUI for creating the MCS and Map and the text from file: (form size = 5,size word = 10 , size mismatch = 0)

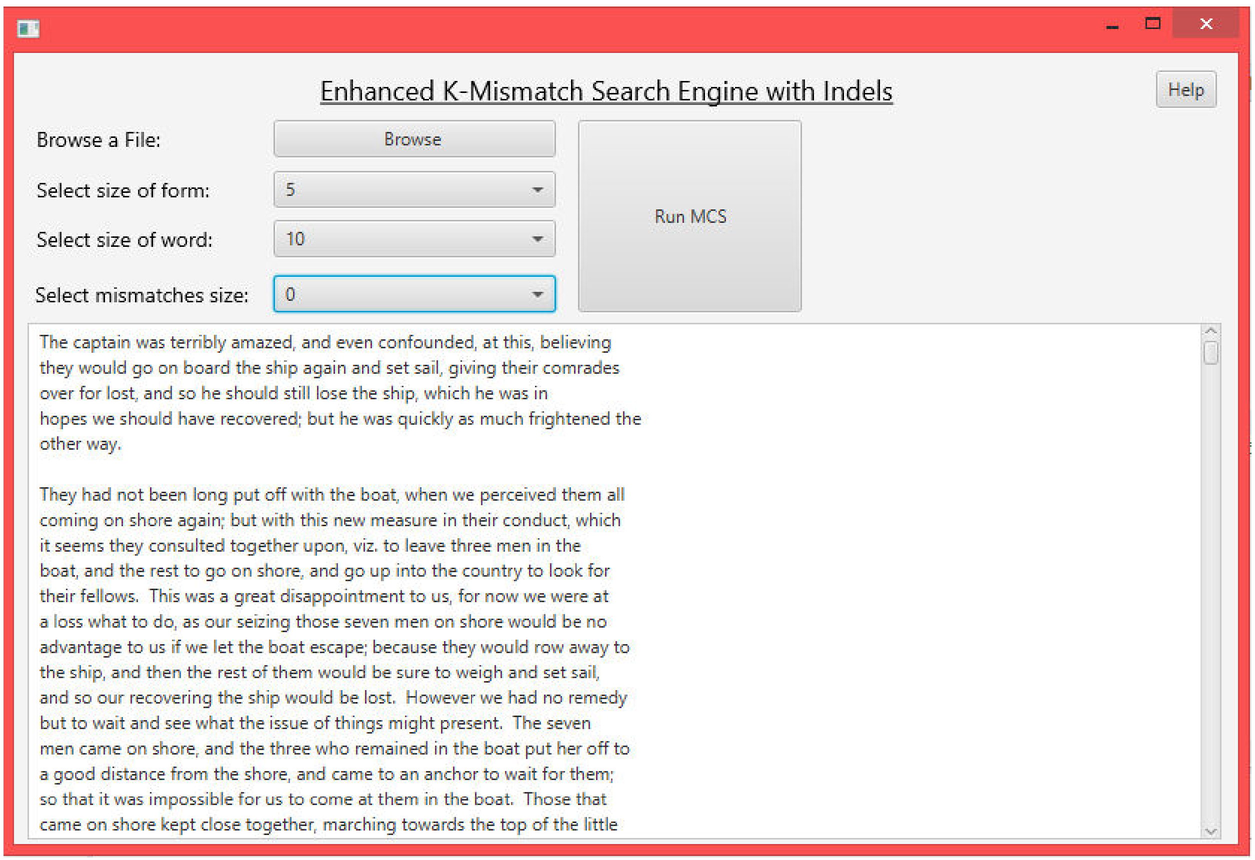

Figure 19. Menu Screen

Finally , we will add a searched word to search on the text through enhanced K-Mismatch Search Engine with Indels:

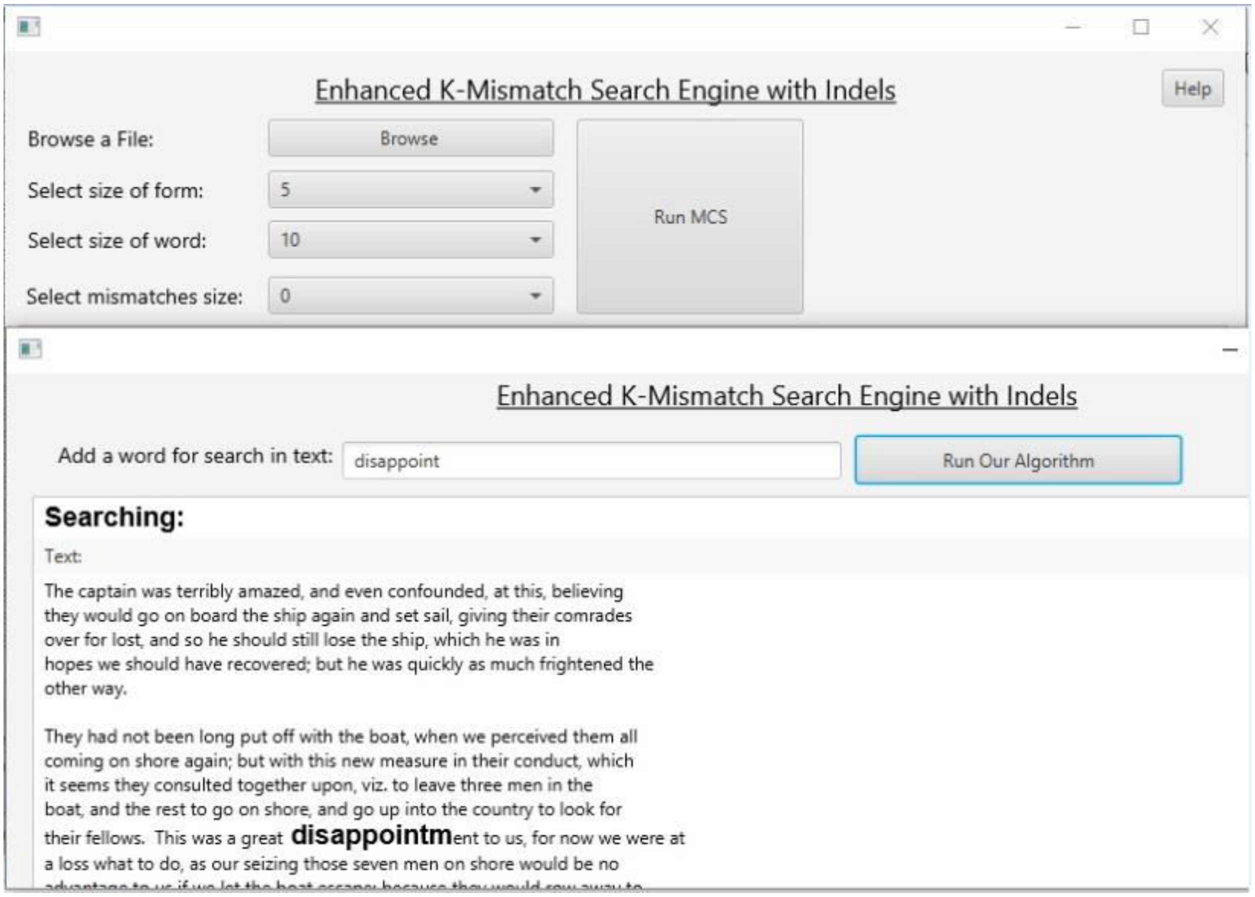

Figure 20. Enhanced K-Mismatch Search Engine with Indels Screen

The running time is: 428 Milli-second on average.

k-mismatch search algorithm:

For the k-mismatch search algorithm we will check for the same parameters.

First we add parameters for creating the MCS and Map and add the text from file:

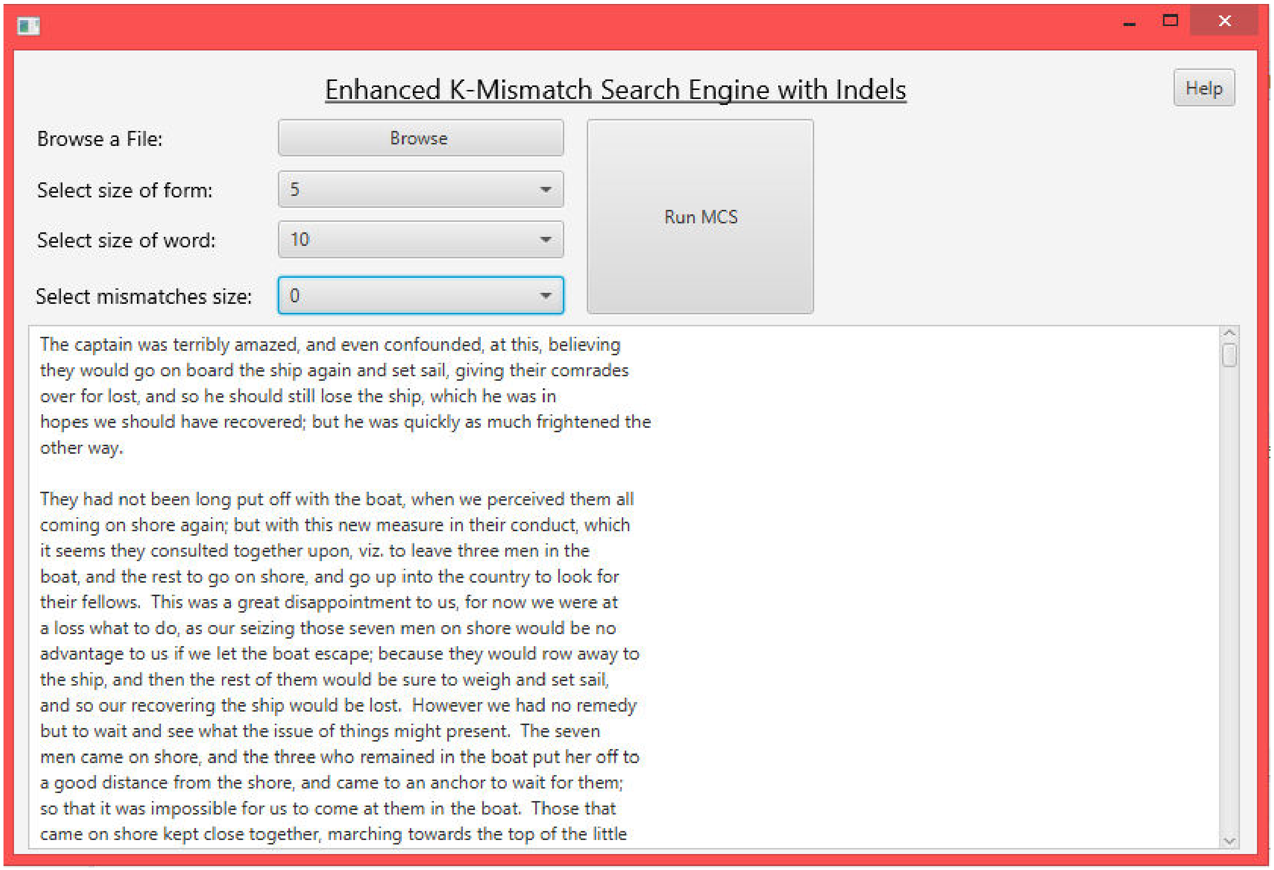

Figure 21. Menu Screen

Finally , we will add a searched word to search on the text through k-mismatch search algorithm:

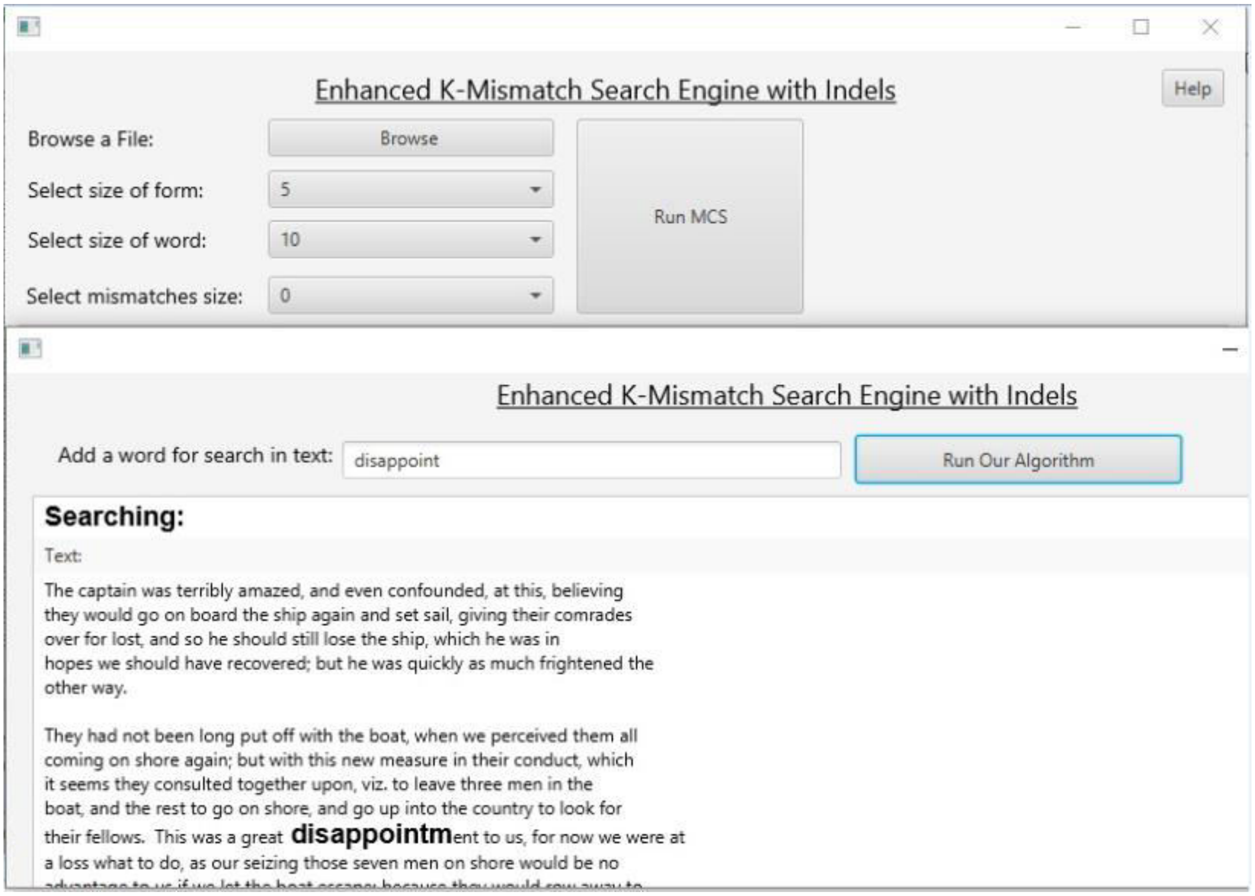

Figure 22. Enhanced K-Mismatch Search Engine with Indels Screen

The running time is: 512 Milli-second on average.

Enhanced K-Mismatch Search Engine with Indels:

First we add parameters for creating the MCS and Map and the text from file: (form size = 5,size word = 10 , size mismatch = 2)

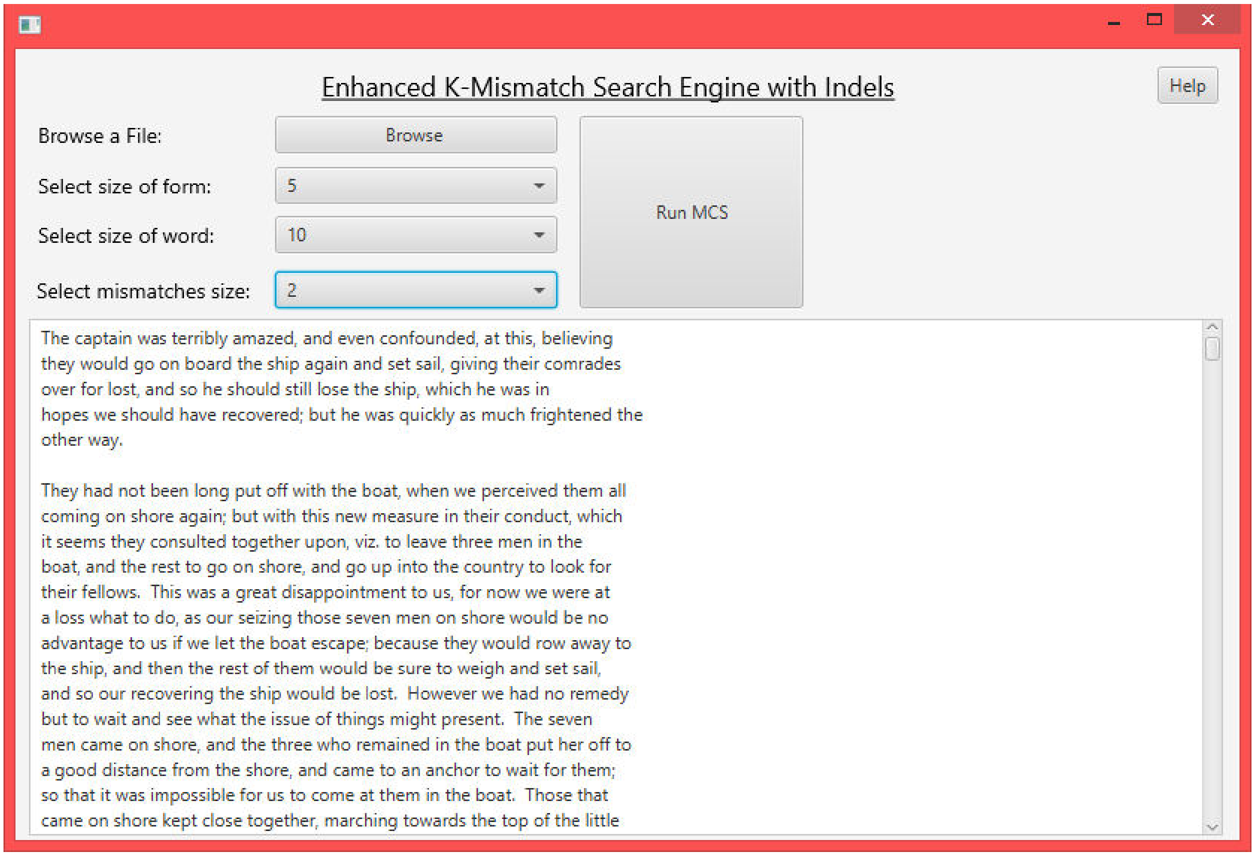

Figure 23. Menu Screen

Finally , we will add a searched word to search on the text through enhanced K-Mismatch Search Engine with Indels:

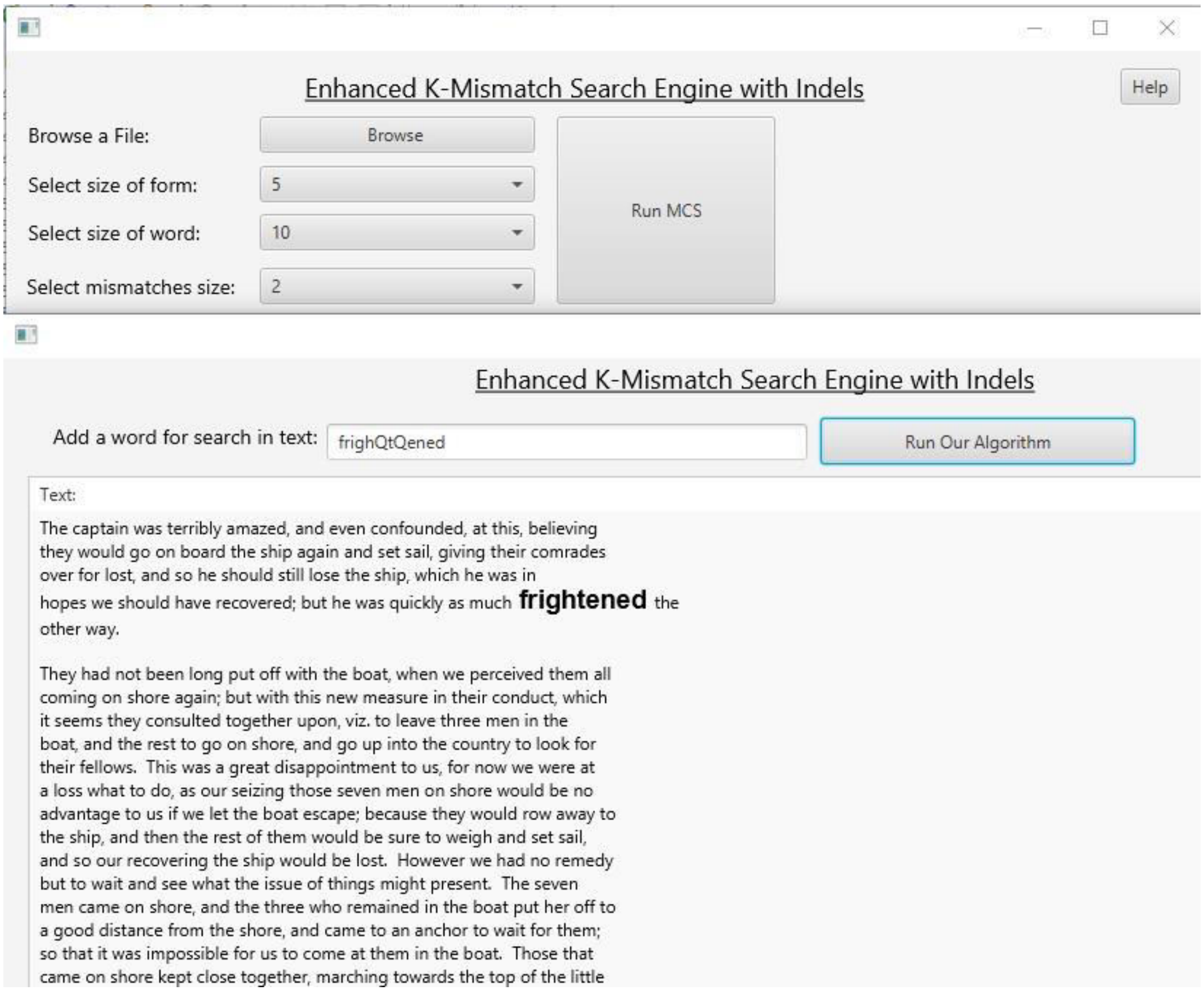

Figure 24. Enhanced K-Mismatch Search Engine with Indels Screen

The running time is: 3620 Milli-second on average.

k-mismatch search algorithm:

For the k-mismatch search algorithm we will check for the same parameters.

First we add parameters for creating the MCS and Map and add the text from file:

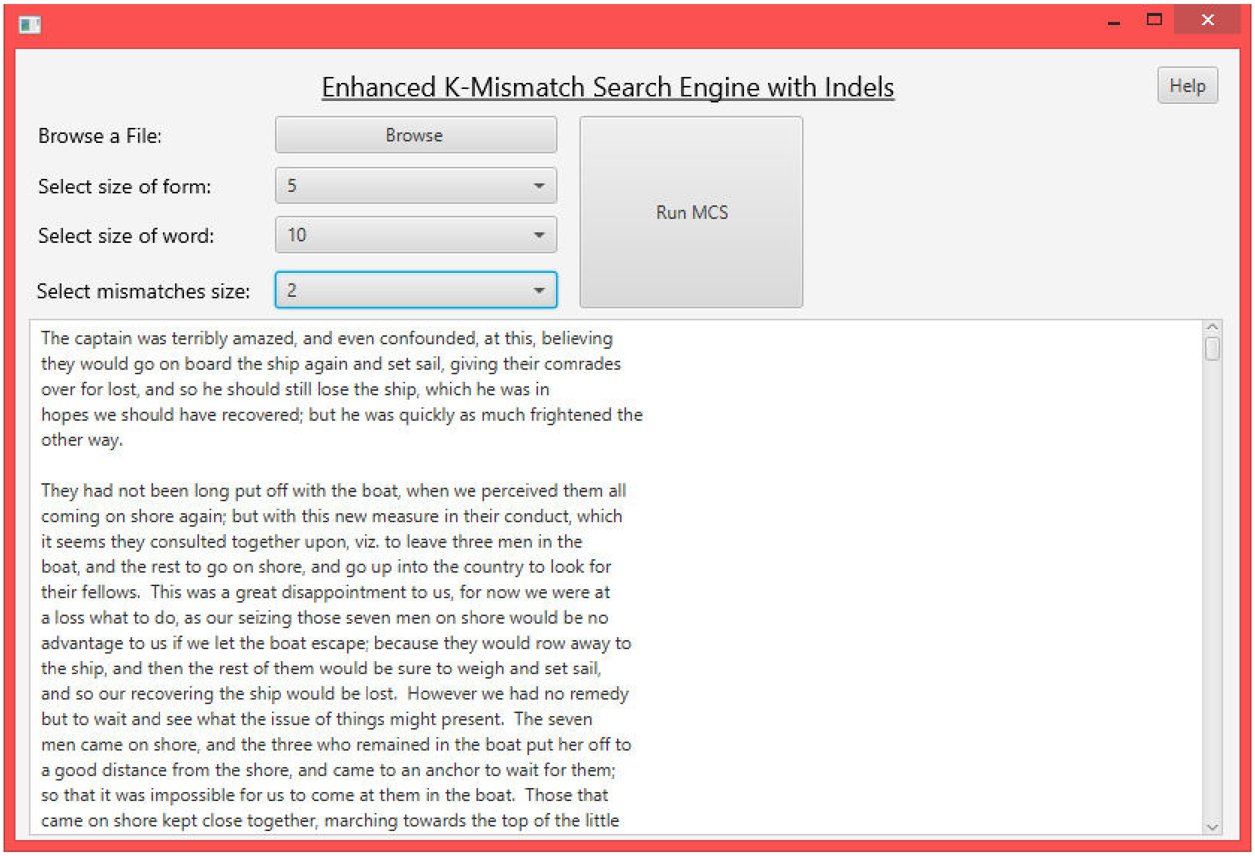

Figure 25. Menu Screen

Finally , we will add a searched word to search on the text through k-mismatch search algorithm:

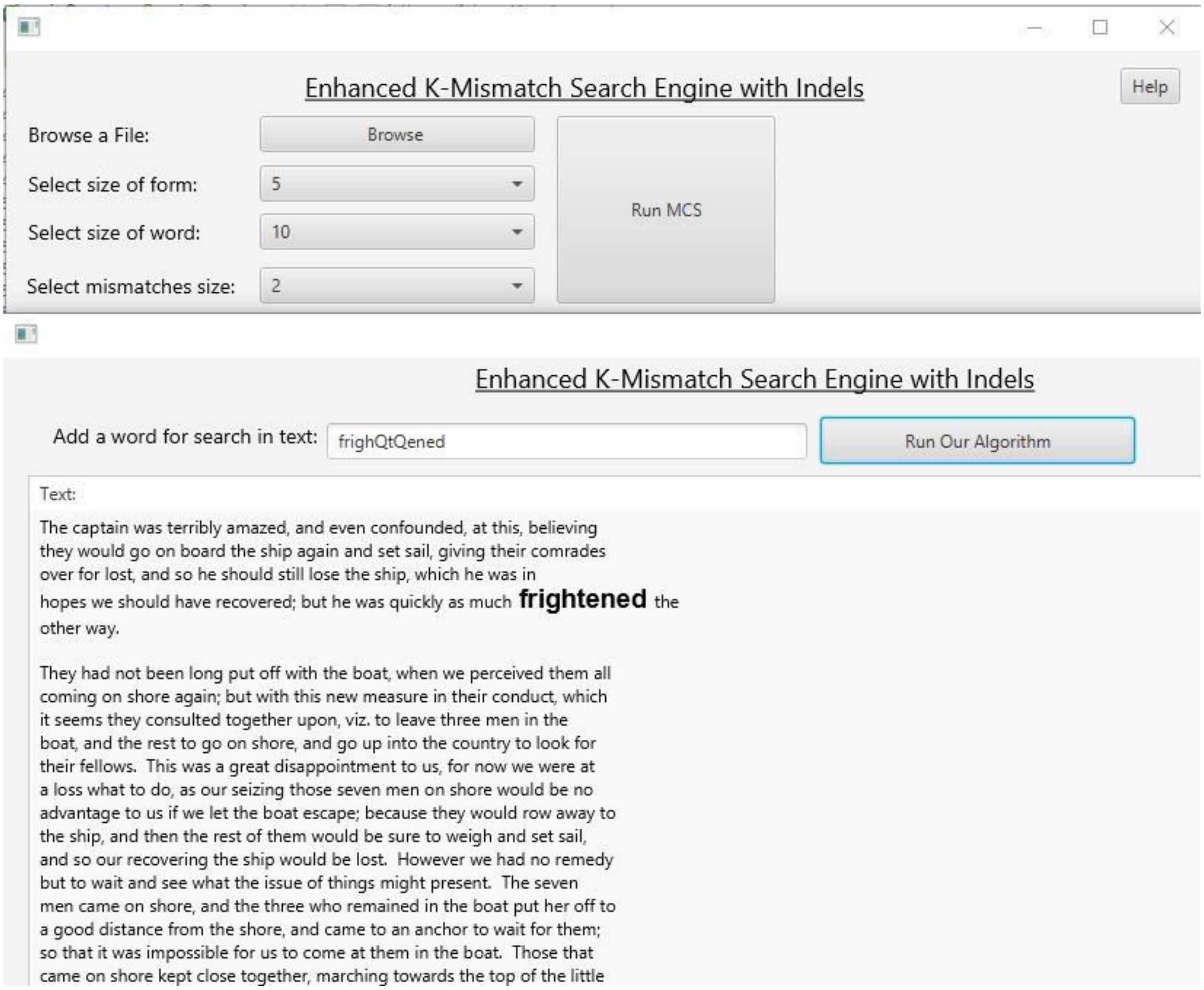

Figure 26. Enhanced K-Mismatch Search Engine with Indels Screen

The running time is: 3688 Milli-second on average.

6.2. Conclusions

The running time is the same complexity as the regular algorithm.

This is because the memory is more larger the complexity time on the enhanced K-Mismatch Search Engine with Indels.

We can see that the MCS of our algorithm is bigger than the regular algorithm , so the memory is important for the running time of our algorithm.

Parameters: form size = 5,size word = 10 , size mismatch = 0

MCS of Enhanced K-Mismatch Search Engine with Indels:

```
1 1 1 1
1 1 1 1
1 0 2 2 2
1 1 0 2 2
1 1 1 0 2
3 3 3 3
1 3 3 3
1 1 3 3
1 1 1 3
```

Final MCS Enhanced K-Mismatch Search Engine with Indels:

```
[111, 133, 333, 113, 1102, 1022]
```

MCS of enhanced k-mismatch search algorithm:

```
1 1 1 1 1 1 1 1 1 1
```

Final MCS k-mismatch search algorithm:

```
[11111]
```

Parameters: form size = 5,size word = 10 , size mismatch = 2

MCS of enhanced k-mismatch search algorithm:

```
1 1 1 1
1 0 0 1
1 0 0 0 2
3 0 0 3
1 0 0 3
1 0 1 0
1 0 0 2 0
3 0 3 0
1 0 3 0
1 1 0 0
1 0 2 0 0
3 3 0 0
1 3 0 0
```

Final MCS Enhanced K-Mismatch Search Engine with Indels:

```
[10020, 1010, 1030, 3030, 111, 1003, 1300, 1001, 3003, 1100, 3300, 10002,
10200]
```

MCS k-mismatch search algorithm:

```
1 0 0 1 1 1 1 1 1 1
1 0 1 0 1 1 1 1 1 1
1 0 1 1 0 1 1 1 1 1
1 0 1 1 1 0 1 1 1 1
1 0 1 1 1 1 0 1 1 1
1 0 1 1 1 1 1 0 1 1
1 0 1 1 1 1 1 1 0 1
1 0 1 1 1 1 1 1 1 0
1 1 0 0 1 1 1 1 1 1
1 1 0 1 0 1 1 1 1 1
1 1 0 1 1 0 1 1 1 1
1 1 0 1 1 1 0 1 1 1
1 1 0 1 1 1 1 0 1 1
1 1 0 1 1 1 1 1 0 1
1 1 0 1 1 1 1 1 1 0
1 1 1 0 0 1 1 1 1 1
1 1 1 0 1 0 1 1 1 1
1 1 1 0 1 1 0 1 1 1
1 1 1 0 1 1 1 0 1 1
1 1 1 0 1 1 1 1 0 1
1 1 1 0 1 1 1 1 1 0
1 1 1 1 0 0 1 1 1 1
1 1 1 1 0 1 0 1 1 1
1 1 1 1 0 1 1 0 1 1
1 1 1 1 0 1 1 1 0 1
1 1 1 1 0 1 1 1 1 0
1 1 1 1 1 0 0 1 1 1
1 1 1 1 1 0 1 0 1 1
1 1 1 1 1 0 1 1 0 1
1 1 1 1 1 0 1 1 1 0
1 1 1 1 1 1 0 0 1 1
1 1 1 1 1 1 0 1 0 1
1 1 1 1 1 1 0 1 1 0
1 1 1 1 1 1 1 0 0 1
1 1 1 1 1 1 1 0 1 0
1 1 1 1 1 1 1 1 0 0
```

Final MCS k-mismatch search algorithm:

```
[1010111, 1100111, 11111, 111011, 1101101]
```

We can see that the MCS of our algorithm is more bigger than the regular algorithm , so the memory of the algorithm has big condition of the running time.

7. REFERENCES

[1] Z. Frenkel and Z. Volkovich, "A new approach for solution of the k-mismatch search problem”.

[2] S. Konstantinidis, “2.3. Ordinary word, edit string, weight, and edit distance”, *Computing the edit distance of a regular language.* Information and Computation 205 (2007) 1307–1316: p. 3-4, 2007.

[3] S. Burkhardt. and J. Kärkkäinen, “1. Introduction”, *Better filtering with gapped q-grams.* Fundamenta informaticae,56(1-2): p. 1-2, 2003.

[4] M. Farach-Colton, et al., *Optimal spaced seeds for faster approximate string matching.* Journal of Computer and System Sciences,73(7): p. 1035-1044, 2007.

[5] F. Nicolas and E. Rivals, *Hardness of optimal spaced seed design.* Journal of Computer and System Sciences,74(5): p. 831-849, 2008.

[6] L. Egidi and G. Manzini, *Better spaced seeds using quadratic residues.* Journal of Computer and System Sciences,79(7): p. 1144-1155, 2013.

[7] R.Clifford and E.Porat, “A filtering algorithm for k-mismatch with don’t cares”, *Information Processing Letters*, 110 (4), 1023, 2010.