Implementation of FM-Index Search

Ali Abbas

Abstract

To deal with the increasing amount of data, text indexes are indispensable tools that allow fast searches. In this work, search using FM-index and wavelet tree is implemented in Java and evaluated on a suite of texts with varying text and alphabet sizes.

1- Introduction

We live in the information age where huge amounts of data is produced every day. The ability to efficiently search through the data to find the relevant information is vital to many areas of scientific research, for example in DNA sequencing. In many cases the same text will be searched many times to find different patterns, and processing the text for every single pattern search is inefficient. A better approach is to preprocess the text by creating an index over the text that will be done once and can be used to search for many patterns. One example of such an index is FM-index, a self index on a text that allows for compression and search queries. Self index means that we don't need to carry the original text. In this work, I implement a pattern search functionality using FM-index and wavelet tree but will not explore the compression properties of FM-index.

The remainder of the paper is structured as follows. In section 2, the method used is detailed. Section 3 presents the experimental results. Finally, section 4 concludes the paper.

2- Method

To implement the search functionality using FM-index and wavelet tree in Java, the following steps are needed:

1. Create the suffix array (SA) using Manber and Myers algorithm
2. Get Burrows-Wheeler Transform (BWT) using SA
3. Create Wavelet Tree of BWT
4. Each node of the Wavelet Tree has preprocessed data structures to enable occurrence queries
5. Use backward search to find all the instances of the pattern in text

2.1 - Suffix Array

To create the suffix array (SA), I used the Manber and Myers algorithm [1].

Create SAH and SA2H arrays where each one uses 16 * m bytes (m is text size) and will be used to store partial suffix arrays for H and 2H length prefix of suffixes:

```
SAH = new Integer[m];
SA2H = new Integer[m];
```

Create the following arrays to be used for bucket management where each one uses 4 * m bytes:

bucketID = new int[m];	keep track of the bucket ID of a suffix in SAH
newBucketID = new int[m];	keep track of the bucket ID of a suffix in SA2H
bucketStartIndex = new int[m];	starting index of a bucket
bucketCounter = new int[m];	number of items in a bucket
bucketTracking = new int[m];	keep track of bucket Number of suffix A_{j+1} in SAH where A_j denotes the suffix S[j,m]

These are not used for SA but can be calculated during SA creation and used later:

C = new HashMap<Character, Integer>();
C.get(c) returns the number of occurrences in S of characters alphabetically smaller than c. C uses about 36 * |Σ| bytes.

sortedAlphabet = new ArrayList<Character>();
sortedAlphabet is a sorted list of the alphabet that will be used later for creating the Wavelet Tree and uses about 16 * |Σ| bytes.

Next, initialize SAH to be indexes of S, then let SAH hold the sorted indexes of S. This will take O(m log(m)) time using the Timsort algorithm implemented in Collections Java library's default sort method.

Loop over all characters of S. Since we have the sorted Indexes, we can compare S[SAH[i]] with S[SAH[i - 1]] to determine if they are in the same bucket or we need to add new buckets. We can fill C, sortedAlphabet, bucketID, and bucketStartIndex as well:

```
numberOfBuckets = 1;
bucketTracking[0] = -1;
sortedAlphabet.add(S[SAH[0]]);
for (int i = 1; i < m; i++) {
        if (S[SAH[i]] == S[SAH[i - 1]]) {
                // still in the same bucket
                bucketID[SAH[i]] = bucketID[SAH[i - 1]];
        } else {
                // start of new bucket
                C.put(S[SAH[i]], i);
                sortedAlphabet.add(S[SAH[i]]);
                bucketID[SAH[i]] = bucketID[SAH[i - 1]] + 1;
                bucketStartIndex[bucketID[SAH[i]]] = i;
                numberOfBuckets++;
        }
        bucketTracking[i] = -1;
}
```

We start with H = 1, which means length-1 prefix of the suffixes and loop until the number of buckets is equal to m. In each iteration, we loop over all items of SAH to fill SA2H. Let's call this loop update_SA2H. For each item j = SAH[i], A_j is in a singleton bucket if the start index of the buket is the last index of SAH, or if the start index of the bucket only differs by one from the starting index of the next bucket. In that case SA2H[i] = j which means the suffix keeps its

position in the next partial suffix array SA2H. If j - H >= 0, we put A_{j-H} in a location in SA2H that is determined by the sum of the starting index of the bucket which contains A_{j-H} and number of items in that bucket. The following shows the update_SA2H loop:

```
for (int i = 0; i < m; i++) {
	j = SAH[i];
	if (bucketStartIndex[bucketID[j]] == m - 1 || bucketStartIndex[bucketID[j]] == bucketStartIndex[bucketID[j] + 1] - 1) {
		// update only if it's the first time
		if (bucketTracking[j] == -1) {
			SA2H[i] = j;
			bucketTracking[j] = bucketID[j];
		}
	}
	if (j - H >= 0) {
		// update only if it's the first time
		if (bucketTracking[j - H] == -1) {
			p = bucketStartIndex[bucketID[j - H]] + bucketCounter[bucketID[j - H]];
			SA2H[p] = j - H;
			bucketCounter[bucketID[j - H]] ++;
			bucketTracking[j - H] = bucketID[j];
		}
	}
}
```

Now that SA2H is filled, we need to fill the new bucket ID array. To do that, we loop over all elements of SAH and SA2H, if the bucket ID of an element in SAH is the same as the bucket ID of the previous element in SAH and the bucket tracking of the same elements in SA2H is the same, then we assign the same bucket ID as the previous elements's bucket ID, otherwise we mark the start of a new bucket ID. We call this loop, update_newBucketID:

```
bucketTracking[SA2H[0]] = -1; // tracking of null character
numberOfBuckets = 1;
for (int i = 1; i < m; i++) {
	if ((bucketID[SAH[i]] == bucketID[SAH[i - 1]]) && bucketTracking[SA2H[i]] == bucketTracking[SA2H[i - 1]]) {
		// still in the same bucket
		newBucketID[SA2H[i]] = newBucketID[SA2H[i - 1]];
	} else {
		// start of new bucket
		newBucketID[SA2H[i]] = newBucketID[SA2H[i - 1]] + 1;
		bucketStartIndex[newBucketID[SA2H[i]]] = i;
		numberOfBuckets++;
	}
}
```

So the complete loop for H is as follows:

```
H = 1;
while (numberOfBuckets < m) {
	update_SA2H();
```

```
        update_newBucketID();
        // reset and copy to prepare for next loop
        for (int i = 0; i < m; i++) {
                // copy newBucketID to bucketID
                bucketID[SA2H[i]] = newBucketID[SA2H[i]];
                // reset bucket tracking
                bucketTracking[SA2H[i]] = -1;
                // copy new SA2H to SAH
                SAH[i] = SA2H[i];
                // reset bucket counter
                bucketCounter[i] = 0;
        }
        H = H * 2;
}
SA = SAH;
```

At the end SAH will contain the suffix array SA. This will take O(m log(m)) time and total space needed is 52 * m bytes which is O(m).

2.2 - Burrows-Wheeler Transform (BWT)

Burrows-Wheeler Transform (BWT) of a string S can be obtained by looping over the suffix array and concatenating the character that is before each suffix [2]:

```
BWT = new ArrayList<Character>(m);
for (int i = 0; i < m; i++) {
        if (SA[i] == 0) {
                BWT.add(S[m - 1]);
        } else {
                BWT.add(S[SA[i] - 1]);
        }
}
```

This will take O(m) time, and total space used is 16 * m bytes, which is O(m).

2.3 - Wavelet Tree

To create the Wavelet Tree, we start from the root node and recursively add left and right child nodes to it [3]. The root starts with the BWT Character array and the complete sorted alphabet:

```
waveletTreeRoot = new WaveletTreeNode(BWT, sortedAlphabet);
```

The general structure of the is as follows:

```
public class WaveletTreeNode {
   private PreprocessRank preprocessRank;
   private WaveletTreeNode leftNode;
   private WaveletTreeNode rightNode;
   private HashMap<Character, Boolean> characterBitMap;
   public WaveletTreeNode(ArrayList<Character> S, ArrayList<Character> sortedAlphabet) {   }
   public int Occ(int i, char c) {    }
}
```

PreprocessRank is the object that holds the bit vector associated with this node, in addition to boundaryRank and smallRank arrays obtained by preprocessing the bit vector to allow for fast rank queries.
Occ(i, c) returns the number of occurrences of character c in S[1,i], where S is the character array associated with this node. In practice we don't keep S but instead it's encoded in PreprocessRank object.
To create a WaveletTreeNode we first check if the length of the sorted alphabet is less than 2. If it is less than 2, it means that it's a leaf node and we don't need to do anything:

```
Public WaveletTreeNode(ArrayList<Character> S, ArrayList<Character> sortedAlphabet){
        int n = sortedAlphabet.size();
        if (n < 2) return;
        characterBitMap = new HashMap<Character, Boolean>();
```

characterBitMap allows us to quickly find if a given character belongs to the left or right child and uses 36 * |Σ| bytes.

the following data structures are temporarily used to create the node and are not needed when using the wavelet tree:

```
sortedLeftAlphabet = new ArrayList<Character>();          use at most 16 * |Σ| bytes
sortedRightAlphabet = new ArrayList<Character>();         use  at most 16 * |Σ| bytes
leftS = new ArrayList<Character>();                       use at most 16 * m bytes
rightS = new ArrayList<Character>();                      use at most 16 * m bytes
```

We divide the sortedAlphabet into left and right parts, while keeping track of where each character belongs by updating characterBitMap (true for right, and false for left):

```
for (int i = 0; i < (n + 1) / 2; i++) {
        sortedLeftAlphabet.add(sortedAlphabet.get(i));
        characterBitMap.put(sortedAlphabet.get(i), false);
}
for (int i = (n + 1) / 2; i < n; i++) {
        sortedRightAlphabet.add(sortedAlphabet.get(i));
        characterBitMap.put(sortedAlphabet.get(i), true);
}
```

Then we create a boolean array B, which serves as a bit vector, with true representing 1 and false representing 0. If a character is in the left child, assign false, otherwise true. In addition, we add the character to the left string or the right string based on the value of B:

```
boolean[] B = new boolean[S.size()];          uses at most m bytes
for (int i = 0; i < S.size(); i++) {
        B[i] = characterBitMap.get(S.get(i));
        if (B[i]) {
                rightS.add(S.get(i));
        } else {
                leftS.add(S.get(i));
        }
}
```

Finally we preprocess B and create left and right child nodes:

```
        preprocessRank = new PreprocessRank(B);
        leftNode = new WaveletTreeNode(leftS, sortedLeftAlphabet);
        rightNode = new WaveletTreeNode(rightS, sortedRightAlphabet);
} // end of WaveletTreeNode constructor
```

Because Wavelet Tree is a balanced binary tree, and we have log(Σ) levels, and we halve the size of string, starting from initial m at each level, we need O(m log(Σ)) time for creating the tree. Temporary space needed for creating each node is at most 32 * (|Σ| + m) which is O(|Σ| + m).

Occ function for each WaveletTreeNode is defined as follows:

```
public int Occ(int i, char c){
        if (leftNode == null) {
                return i;
        }
        if (characterBitMap.get(c)) {
                return rightNode.Occ(preprocessRank.rank1(i - 1), c);
        } else {
                return leftNode.Occ(i - preprocessRank.rank1(i - 1), c);
        }
}
```

It recursively searches the left or right node depending on if the character is in the left or right node using the constant time rank1 queries, made possible by preprocessing B.

2.4 - Preprocessing Rank Queries

Given a boolean array with n element, we can answer rank1(i) queries, which tell us how many true values are in B[1,i] (equivalent to how many 1s are in a bit vector), in constant time by preprocessing B [4]. The outline of the PreprocessRank class is as follows:

```
public class PreprocessRank {
   public boolean[] B;
   private int n;
   private int[] boundaryRank;
   private int[][] smallRank;
   private int t;
   public PreprocessRank(boolean[] B) {    }
   public int rank1(int i) {    }
}
```

In the constructor PreprocessRank(boolean[] B), we divide B into t = $\log_2(n) / 2$ blocks. To get boundaryRank values, we loop over B and count the true values, when we reach the boundary of a block, we store the number of true values encountered so far in boundaryRank[boundaryRankIndex] and increment boundaryRankIndex by one:

```
public PreprocessRank(boolean[] B){
        this.B = B;
```

```
        n = B.length;
        t = (int) Math.ceil((Math.log(n) / Math.log(2)) / 2);
        int numberOfBlocks = (int) Math.ceil((double) n / t);
        boundaryRank = new int[numberOfBlocks];
        boundaryRank[0] = 0;
        int numberOfOnes = 0;
        int boundaryRankIndex = 1;
        for (int i = 0; i < (numberOfBlocks - 1) * t; i++) {
                if (B[i]) {
                        numberOfOnes ++;
                }
                if ((i + 1) % t == 0) {
                        boundaryRank[boundaryRankIndex] = numberOfOnes;
                        boundaryRankIndex ++;
                }
        }
```

For smallRank, we create a 2D array of size 2^t by t, then we loop over all elements of smallRank with i for row index and j for column index. For each smallRank[i][j] is the number of 1s in the binary representation of i from the most significant bit to the bit at (t - j - 1) position. To do this we isolate the bit at position (t - j - 1) by right shifting i by (t - j - 1) and performing binary AND with 1 [(i >> (t - j - 1)) & 1]. The isolated bit value is accumulated in numberOfOnes, which tells us how many ones we encountered so far starting from the most significant bit. Then for each iteration of j, smallRank[i][j] = numberOfOnes and we set numberOfOnes to zero before each j loop:

```
        int smallRankRows = (int)Math.pow(2, t);
        smallRank = new int[smallRankRows][t];
        for (int i = 0; i < smallRankRows; i++) {
                numberOfOnes = 0;
                for (int j = 0; j < t; j++) {
                        numberOfOnes += (i >> (t - j - 1)) & 1;
                        smallRank[i][j] = numberOfOnes;
                }
        }
} // end of PreprocessRank constructor
```

Now the answer to rank1(i) queries is simply boundaryRank[j] + smallRank[y][k], where j is $\lfloor i/t \rfloor$, k is $\lfloor i \bmod t \rfloor$, and y is the integer representation of bits from j * t to (j + 1) * t - 1 position in B:

```
public int rank1(int i){
        if (i < 0) {
                return 0;
        }
        int j = Math.floorDiv(i, t);
        int y = 0;
        for (int p = 0; p < t && (j * t + p) < n; p++) {
                if (B[j * t + p]) {
                        y |= 1 << (t - p - 1);
                }
        }
        int k = Math.floorMod(i, t);
```

```
        return boundaryRank[j] + smallRank[y][k];
}
```

2.5 - Backward Search

At this point, the index is created and we are ready to use it for search. We only keep S, SA and C and the wavelet tree, everything else can be discarded. In each node of the wavelet tree we are also keeping characterBitMap, B, boundaryRank, and smallRank. Now we can use Backward Search to find all the starting and ending positions of a pattern in text. It starts from the last character of pattern, and using C and Occ queries to narrow down the starting and ending position of pattern in SA [4].

First, we see how can we use the wavelet tree to perform Occ:

```
private int Occ(int i, char c){
        return waveletTreeRoot.Occ(i + 1, c);
}
```

Then the backward search is:

```
public SearchInterval backwardSearch(char[] P){
        int i = P.length - 1;
        int sp = 0;
        int ep = m - 1;
        while (sp <= ep && i >= 0) {
                if (!C.containsKey(P[i])) {
                    return null;
                }
                sp = C.get(P[i]) + Occ(sp - 1, P[i]);
                ep = C.get(P[i]) + Occ(ep, P[i]) - 1;
                i --;
        }

        if (ep < sp) {
                return null;
        } else {
                return new SearchInterval(sp, ep);
        }
}
```

2.6 - Search Using FM-Index

First we create the FM-index over text:

```
FMIndex fmIndex = new FMIndex(text);
```

Then we use the backward search to find the interval in SA that contains the pattern:

```
SearchInterval searchInterval = fmIndex.backwardSearch(P);
```

Now we can get ranges of the pattern in text:

```
count = 0;
for (int i = searchInterval.sp; i <= searchInterval.ep; i++) {
        firstIndexValues[count] = fmIndex.SA[i];
        secondIndexValues[count] = fmIndex.SA[i] + P.length - 1;
        count++;
}
```

Finally, I sort the firstIndexValues and record the line number for each index and output the line that contains the pattern, with pattern enclosed in "[]". An example of the result of searching the pattern "heaven" in bible.txt is shown here:

Text Size: 4047393
Pattern Size: 6
Alphabet Size: 63
Total Space Used in Bytes: 100090470
Index Creation Execution Time In Seconds: 19.9554173
Search Execution Time In Seconds: 7.972E-4
Total Execution Time In Seconds: 19.9562146

718 Pattern(s) was found in the following range(s) [begin index, end index] (zero-based index):

1 Line: 1 [33, 38] [heaven] and the earth. And the earth was without form, and void; and darkness was upon the face of t
2 Line: 8 [849, 854] under the [heaven] be gathered together unto one place, and let the dry land appear: and it was so.

3- Experimental Results and Evaluation

To evaluate the implementation, from Canterbury corpus [5], I used the bible.txt (containing the King James version of the bible) with alphabet size 63 to represent a natural language text, and E.coli (containing the Complete genome of the E. Coli bacterium) with alphabet size 4 to represent genomic sequence data. I also generated a random.txt file with alphabet size 255, that contains random characters with values between 1 and 255.
I ran the experiments on a Windows 10 Laptop with Intel Core i7-4720HQ CPU @ 2.6 GHz with 4 physical cores and hyperthreading, 16 GB RAM, and Nvidia GeForce GTX 970M Graphic Card. Java 8 was used to code the implementation.
For the experiments, I varied the text size from 200,000 to 4,000,000 in 200,000 increments, and also varied the pattern size from 2 to 200,000, and then in increments of 200,000 to less than the text size.

Figure 1 shows the index creation time vs text size. As we can see the index creation time increases linearly with respect to text size. The effect of alphabet size seems to be the larger the alphabet size, the shorter the time it takes to create the index. This seems at first counterintuitive, but one reason could be that in the suffix array creation, having a small alphabet, we have many suffixes in the same bucket which will take more iterations to split those buckets until we have as many buckets as the text size.

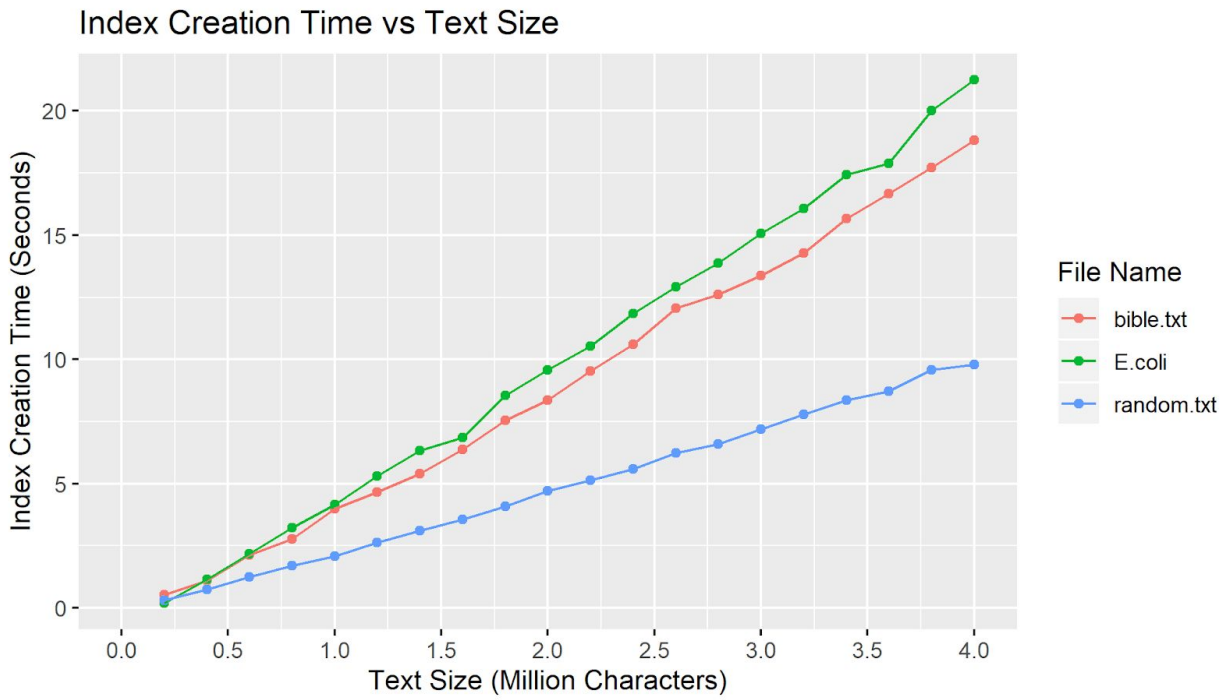

Figure 1. Index Creation Time vs Text Size for Different Texts

Figure 2. shows the space required for creating the index vs text size. As we can see the space increases linearly with respect to text size. Also we notice that increasing the alphabet size from lowest in E.coli to highest in random.txt, increases the space required.

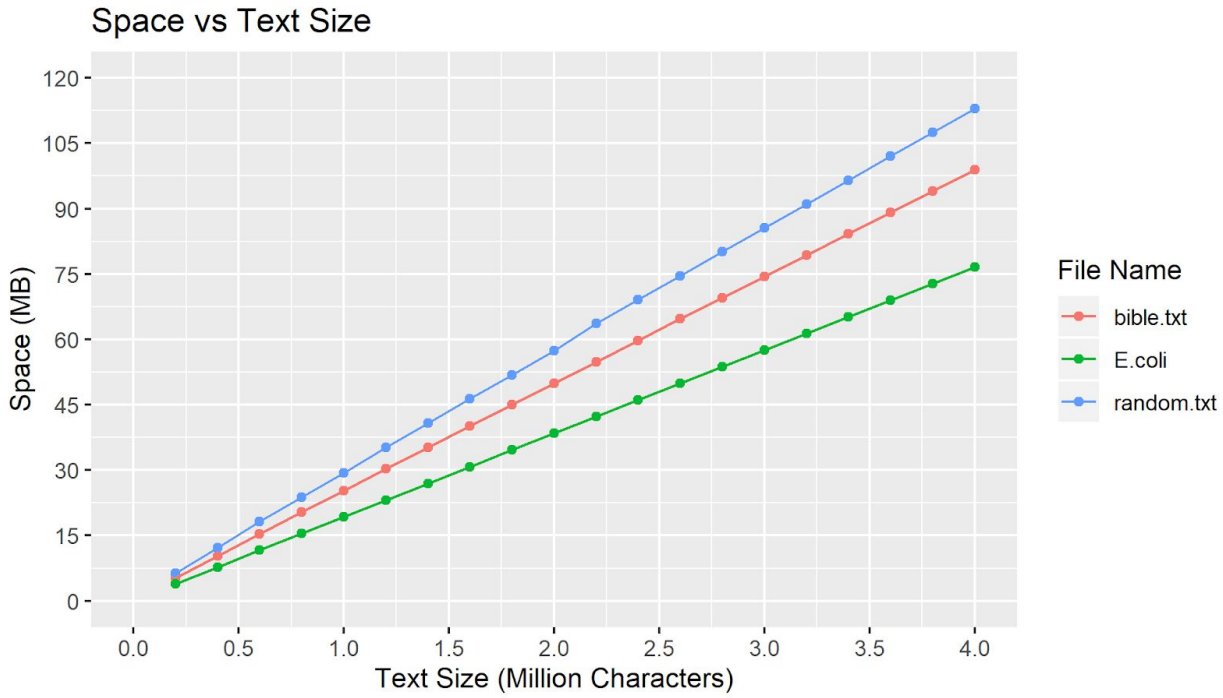

Figure 2. Space vs Text Size for Different Texts

Figure 3. shows the search time vs pattern size when text size is held constant at 4 million characters. As we can see, the search time linearly increases with pattern size but with different rates for different text. The difference can be explained in terms of different alphabet sizes,

where having a larger alphabet size, makes the search more time consuming. random.txt with the largest alphabet size, takes the longest to search, and E.coli with smallest alphabet size has the lowest search time.

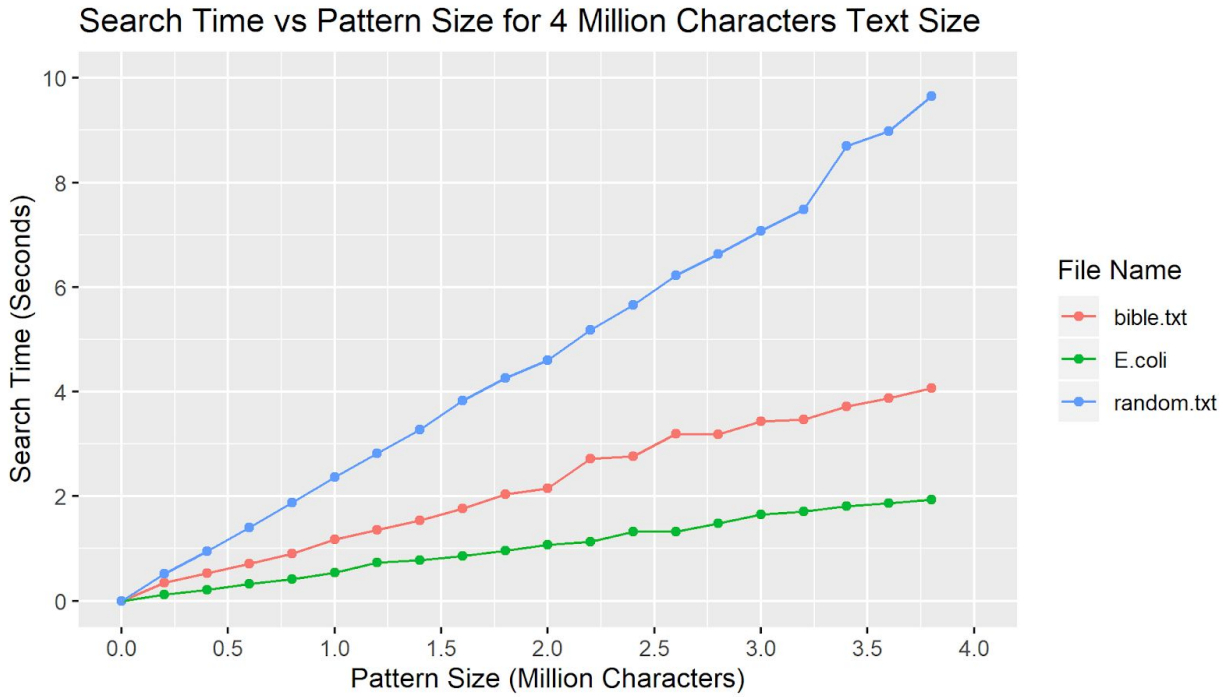

Figure 3. Search Time vs Pattern Size for 4 Million Characters Text Size

4- Conclusion

Searching large text efficiently has many applications. We implemented FM-index in Java and after the evaluation on different text files with varying alphabet sizes, we conclude that time and space required to create an FM-index and search it with respect to text size is linear. We also found that larger alphabet size, increases the space required for the index and time required for pattern search, but reduces the index creation time, compared to smaller alphabet size.

References

1- S. Yooseph, 2020, Lecture 7, lecture notes, Algorithms on Strings and Sequences COT 6417, University of Central Florida.

2- S. Yooseph, 2020, Lecture 10, lecture notes, Algorithms on Strings and Sequences COT 6417, University of Central Florida.

3- S. Yooseph, 2020, Lecture 12, lecture notes, Algorithms on Strings and Sequences COT 6417, University of Central Florida.

4- S. Yooseph, 2020, Lecture 11, lecture notes, Algorithms on Strings and Sequences COT 6417, University of Central Florida.

5- The Canterbury Corpus. (2020, November 14). Canterbury Corpus Website. Retrieved from https://corpus.canterbury.ac.nz/descriptions/