FM-Index and Backwards Search

Adam Groves
ADAM.GROVES@CS.MSU.EDU
Department of Computer Science
Montana State University
Bozeman, MT 59717

Abstract

This paper describes a technique know as FM-Index that can be used to compress text while also allowing for efficient retrieval of the number of occurrences of a pattern as well as the location of the pattern in the original uncompressed text. FM-Index is used widely in bioinformatics to provide space and time efficient searching of genes within large DNA sequences. Using a search method called *backwards search*, the search can be performed in O(1) time.

1. Introduction

An FM-Index is a compressed full-text substring index that is used to efficiently find the number of occurrences of a pattern inside the compressed text as well as the locations of the pattern(s).[1] The motivation behind this data structure and supporting algorithms is the need to search large strings such as DNA in order to find particular genes which are composed of nucleotide sequences. These sequences act as the search query and the entire DNA sequence is the search space. This space is often times very large and the storage space can be reduced by compressing the string. In this paper, a technique known as Burrows-Wheeler Transformation will be executed on a string of characters to produce the Burrows-Wheeler text (BW text). The original text can be reconstructed from the BW text which allows us to discard the original, and much larger string providing compression. The BW text will be further compressed using a wavelet tree which will also provide O(1) time computation of a rank queries. There are other auxiliary data structures that will be used as well and they will be described in the next chapter.

2. Data Structures Implemented

The subsections below discuss each of the four training algorithms in a little more detail in an effort to show the functional differences between each. The three evolutionary algorithms are extremely similar in the structure of their generic algorithms, but subtle differences in the mutation and crossover operators lead to varying results when applying each to the same training problems.

2.1. Burrows-Wheeler Text

The Burrows-Wheeler text is a string containing all the same characters as the original string *S* used to generate it. The BW text can be generated by taking |*S*| rotations of *S* and storing them in an array. The array is then lexicographically sorted. The last column of the sorted rotations is the BW text as shown in Figure 1. Another way to generate the BW text is to first create a suffix tree and traverse it to provide a list of suffixes which closely resemble the first table shown in Figure 1. The only difference is that all characters occurring after the terminating symbol ($) will not be included. Therefore, the BW text cannot be found by getting the last column using this method but instead, Equation 1, shown below, can be used to generate the BW text.

ACGTACGT$
CGTACGT$A
GTACGT$AC
TACGT$ACG
ACGT$ACGT
CGT$ACGTA
GT$ACGTAC
T$ACGTACG
$ACGTACGT

Rotate

$ACGTACG **T**
ACGT$ACG **T**
ACGTACGT **$**
CGT$ACGT **A**
CGTACGT$ **A**
GT$ACGTA **C**
GTACGT$A **C**
T$ACGTAC **G**
TACGT$AC **G**

BW Text

Figure 1. The first table shows all rotations of the string *S* = ACGTACGT$. The second table shows the lexicographically sorted rows. As you can see, the last column of the table is the Burrows-Wheeler text = TT$AACCGG.

$$BW[\,i\,] = \begin{cases} S[SA[\,i\,] - 1] & \text{if } SA[\,i\,] \mathrel{!}= 1 \\ S[n] & \text{if } SA[\,i\,] = 1 \end{cases}$$

Equation 1

2.2. Occurrence Table *C*

The second data structure that is required is the *C Lookup Table*. This lookup table stores the occurrences of all lexicographically smaller characters occurring in the string *S*. The table was implemented using a hash table which provides O(1) time retrieval of the occurrences. Figure 2 shows the C lookup table for the string *S* = ACGTACGT$.

C Lookup

$	A	C	G	T
0	1	3	5	7

Figure 2. C lookup table that sores the occurrences of all lexicographically smaller characters occurring in the string *S*=ACGTACGT$.

2.3. Wavelet Tree as Occ(·) look-up

This is the bottleneck of the FM-Index. Many different implementations of the FM-Index are used and they mainly differ in the data structure used for occ(·) look-up. Occ(*x, i*) returns the number of occurrences of the character *x* in the range(1..*i*). The original string *S* is first transformed into the BW text. The BW text is then used to generate the wavelet tree which stores the text as a tree of individual succinct data structures as nodes. This means that not only is the BW text compressed further, but rank queries can be performed in O(log n) time. When representing the original string as binary data, a rank query can be performed in O(log 2) = O(1) time. To construct the wavelet tree, we first take the alphabet of the string to compress, in our case, the BW text, and encode the first half as 0, and the second half as 1. e.g. {A,C,G,T} would be encoded as {0,0,1,1}.[4] Next, each 0 encoded character is grouped as a sub-tree as well as the 1 encoded characters. For example, {A,C} would be used to create the left node of the root, and {G,T} would be used to create the right node of the root.[4] This is applied recursively until each character can only be represented as a 0 or a 1 and as a result, all ambiguity of the the representations is resolved.[4] An example of this process is shown below in Figure 3. The rank queries, in the context of FM-Indices, are know as occ(·). From now on, the rank query will be used as such. As an example, let's compute occ(A, 5). As you can see in

Figure 4, ‘A’ is initially encoded as 0. Therefore, all 0’s are counted up until position 5. There are 3 occurrences of 0’s in the range(1..5) for the root bit vector. Also, since ‘A’ was encoded as a 0 at the root level, the next step will be to take the path of the left node which groups all 0 encoded characters from the root node. The previous result of 3 will be used as the ending index for the left node of the root. However, as you can see, ‘A’ is encoded as 1 for the roots left node. Therefore, we count the number of occurrences of 1’s in range(1..3). This results in 2, which is our final answer for the rank query occ(A, 5) and can be seen checked by simply looking at the BW text that was used to construct the tree.[4]

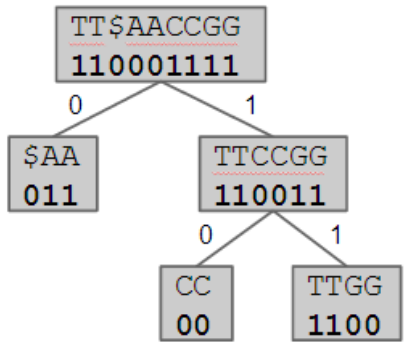

Figure 3. The BW text TT$AACCGG is compressed using a wavelet tree. The alphabet {$,A,C,G,T} is encoded as {0,0,1,1,1}. From the root, the left node is created form all 0 encoded characters while the right node is created from all 1 encoded characters. The nodes bit vectors are re-encoded for each node and is recursively applied until one character corresponds to a 0 or a 1. Note that the strings aren’t actually stored in the wavelet tree nodes, but are only shown here to guide the encoding process. Only the bit vector is stored.[4]

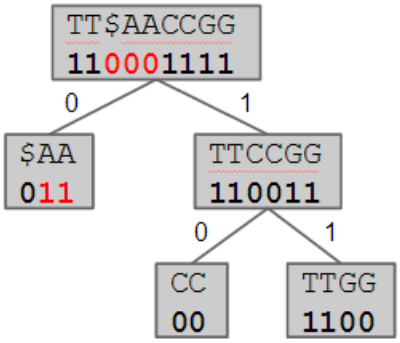

Figure 4. Occ(A, 5) rank query. First step is to count occurrences of 0’s in range(1..5) which equals 3. Then count the number of 1’s in the left sub-tree in range(1..3) since A was encoded as 1 in this node. The result is 2.

3. Reconstruction of *S* from the Wavelet Tree Compressed BW-Text

Just as the original string could be transformed into its BW Text, then compressed into a tree of bit vectors as wavelet tree, decompression of the string *S* is also possible in order to retrieve the original string. This way the original string *S* can be discarded and only the wavelet tree, along with C lookup table can be used to count the occurrences of, and the positions of search patterns. The first step to reconstructing the string *S* is to retrieve the BW text from the wavelet

tree; that is, to decompress it. The BW text will be used as the last column as previously shown in table *M*. To generate L, pairs of rank queries are used to find where L[i] would differ for a given i. Using this approach, we can find L[i] by comparing rank(σ, i) and rank(σ, i+1) for all σ∈Σ, where Σ is the alphabet. If the two ranks result in different values, then L[i] is found. This calculation is done for the range(0, |*S*|), thus generating the whole BW text from just the wavelet tree. The reconstruction of *S* is described in Figure 5, given the last column has been generated.

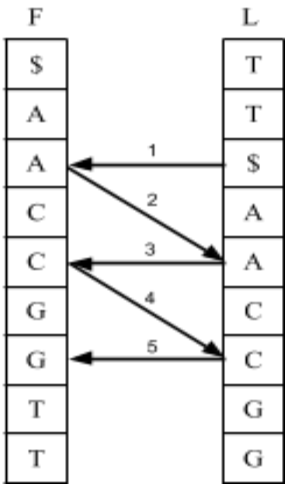

Figure 5. After the Last column (BW text) is generated, the first column can be generated by sorting the BW text in lexicographical order. The original string *S* can then be generated by following the steps shown in this graphic. First, the terminating symbol $ is found in the last column. If *i* is the index of the terminating symbol, then F[i] is the first symbol of the string *S*. In this case, A. since this is the second a in F, the second step finds the second A in L. The next symbol in *S* is found by step 3 in the same way step one is determined. This process continues for the lenth of *S,* and eventually, the original string *S* is found.

4. Backwards Search

With the data structures in place, searching the string *S* now requires a search algorithm that ensures O(1) time retrieval of the number of occurrences of the search pattern. FM-Index allows us to use count() and locate() methods that count the number of occurrences of a pattern in *S* and locate the positions of the pattern in *S*, respectively. In order to locate the position(s) of the search query, backward search has to be executed which retrieves the range of indices that the pattern corresponds to in the sorted suffix array, *M*. The backwards search algorithm is shown below in Figure 6. The algorithm is named this because it iterates over the search pattern in reverse and computes a set of indices that corresponds to start and end pointers of all occurrences of the pattern in *M*. The start and end indices (*st* and *ed*) are initialized to 1 and |*Q*|, respectively. *Q* is the pattern to search for. The backward search method is the same thing as count(). It simply returns the a pair of indices. These indices can be used to find the positions of *Q* in the original text *S* by by passing each index in the range to locate(*i*), where *i* is an index in the range returned by count().

```
Backwards_Search(Q[1..m])
1.  x = Q[m] ; st = C[x] +1;  ed = C[x+1];
2.  i = m- 1;
3.  while st <= ed and i >= 1 do
4.        x = Q[i];
5.        st  = C[x] + occ(x, st - 1) + 1;
6.        ed = C[x] + occ(x, ed);
7.        i = i -1;
8.  end while;
9.  if st > ed, then pattern not found else report [st..ed].
```

Figure 6. Backwards-Search algorithm[3]

5. Results

A string *S* of size 1000 was generated and backwards search was used to obtain a set of indices representing the start and end of the range where the search pattern could be found in the table *M*. A search pattern of Q=”ACC” was used and the results were a pair of indices 68 and 79. This represents the range(68..79) of the occurrences of the pattern in the table *M*. The output is shown in Figure 7.

```
run:
Enter the search query
ACC
The result range is: 68..79
```

Figure 7. The output of the program after entering Q=”ACC”. The result is the pair of indices 68 and 79 representing a range of the occurrences of the search query in table *M*.

Figure 8 shows the result of the reconstruction algorithm that extracts *S* from the FM-Index.

```
Enter text
mississippi$
Original text: mississippi$
```

Figure 8. After entering the text “mississippi$”, the text is compressed as an FM-Index with wavelet tree. Then immediately, the reconstruction algorithm is ran that extracts the original text from the FM-Index and displays it. The code for this text is in FMText.java.

6. Conclusion

Using less than 1.5 gigabytes, the entire human genome can be stored by the use of the FM-Index.[3] By using an FM-Index, not only space is conserved, but search time is also reduced due to the implementation of an efficient succinct occ(·) data structure. In this project, a wavelet tree was used to provide O(1) time computation of occ(·). Other data structures are proposed and implemented throughout the life of this technique and each one delivers a trade-off between space and time efficiency. For example, The wavelet tree structure sacrifices optimal query time for space conservation.[2]

7. Summary

An FM-Index is a method to not only compress large text, but to also provide efficient querying on the text in O(1) time. This is primarily used in the field of bioinformatics to query protein and DNA sequences that take up unreasonable amounts of space if left uncompressed. The compressed text can be retrieved from the compressed text using a reconstruction algorithm. This allows the original text to be discarded. By using a small collection of data structures and accompanying algorithms, the FM-Index has become one of the most elegant and efficient search methods at the time of this paper.

References

1. FM-index. (2013, September 18). In *Wikipedia, The Free Encyclopedia*. Retrieved December 5, 2013, from http://en.wikipedia.org/w/index.php?title=FM-index&oldid=573557526

2. Compressing Occ with Wavelet Trees. Retrieved December 5, 2013, from http://www.mi.fu-berlin.de/wiki/pub/ABI/SS13Lecture8Materials/bwt2.pdf

3. Sung, W. (2010). Suffix Tree. *Algorithms in Bioinformatics* (pp.76-79). Boca Raton: Chapman & Hall/CRC.

4. Bowe, A. (2011, Aug 24). FM-Indexes and Backwards Search [Web log post]. Retrieved December 5, 2013, from http://alexbowe.com/fm-index/.