Implementation Structure

Jochen Singer

June 15, 2011

Abstract

This document describes the structure of the classes chosen to implement the FM-Index by Ferragina and Manzini in SeqAn. In addition the steps necessary to implement the index will be shown and described. This document is based on [3]. A very short description of the Index can be found here: FM-Index wiki.

This document will describe several different implementation of the FM-Index. Even though they all differ in one or more aspects they have the following properties in common:

- They are based on the Burrows-Wheeler-Transform (BWT) to rearrange the characters of a given text T. The Burrows-Wheller transformed text (T*) is usually more efficient compressible then the original text T, since equal substring tend to group together.
- T* is compressed to save memory.
- Usually some auxiliary data are stored to use the index as a fast searching tool.

The next section will introduce the BWT. Afterwards we will describe the original FM-index created by Ferragina and Manzini [1]. The description will be followed by sections in which other implementations are introduced and compared.

1 The Burrows-Wheeler-Transform

Text

2 The FM-Index

2.1 The Original FM-Index

- Opportunistic Data Structures with Applications [1]
- Indexing compressed text [3]

As can be seen in the previous chapter the BWT is a rearrangement of characters of a given text. The rearrangement is based on a data structure which sorts suffices of the original string and groups them together according to their similarity. In doing so the BWT creates a new string T* in which equal characters tend to appear together. This observation can be used to efficiently compress T*. In addition the underlying suffix array structure can be used to efficiently search for patterns P in T.

We will now first introduce the searching procedure before focusing on the compression.

2.1.1 Searching based on the BWT

2.1.2 Compression of the BWT

1. move-to-front encoding (MTF)
2. run-length encoder
3. variable-length-prefix code

2.1.3 Counting occurrences in the compressed BWT

The previous two sections described how we can efficiently search for a pattern P in text T and how we can compress T. However, so far we are only fast or space efficient, but not both at the same time. This section will describe how Ferragina and Manzini overcame this shortcoming.

In order to answer how many times a given character appeared before a given position in T* in constant time they partition T* into buckets. To be precise, they introduce two layers of partitioning. The first layer consists of buckets which span a substring of T* of length l, while the second layers consists of super buckets which cover a substring of length l^2. The following information are then stored and associated with the buckets, respectively super buckets:

- Super buckets
  - $NO_j[|\Sigma|]$ - Stores the number of occurrences of c in NOj[c] in T*[1,jl^2]. In other words: $NO_j[|\Sigma|]$ stores the number of occurrences of a given character in T* until the given super bucket.
  - W[n/l^2] - Stores the sum of the compressed buckets until the given super bucket.
- Buckets
  - $NO'_j[|\Sigma|]$ - Stores the number of occurrences of a given character in T* until the given bucket starting from the preceding super bucket.
  - W[n/l^2] - Stores the sum of the compressed buckets until the given super bucket starting from the preceding super bucket.
- Substrings of the buckets
  - MTF[n/l] - Stores in MTF[j] a picture of the state of the MTF-list at the beginning of the encoding of T* in the bucket j.

$\|\Sigma\|$	l	l'	dimension 3.	4 .	number of entries	memory consumption [MB]
5	8	40	1099511627776	3125	1.37E+017	51539607552
5	16	80	1.21E+024	3125	4.68E+044	1.51E+023

Table 1: Space consumption of S for several chosen settings.

- S - Table S is the most space consuming and complicated table. It can be interpreted as a four dimensional array which stores the number of occurrences of a given character in a given substring. In more detail, S[c, h, BZ_j, MTF[j]] stores for **every** character c for **every** length h (h ¡= l) for **every possible** compressed string BZ_j and **every possible** picture of the MTF-list the number of occurrences of c.

  On first view this four dimensional array does not seem to enable a space efficient storage of the FM-index at all. **However, the key observation is that several positions in T* may be covered by a single entry in S.**

Using the data structure introduced above one can determine the number of occurrences of a given character at a specific position in constant time. This is done by adding the information of the proceeding super bucket and bucket to the value in the lookup table S. However this approach has an obvious bottleneck which is the space consumption of table S. The following section will analyze the amount of memory needed in total and especially for S.

Memory Consumption of Table S As described in 2.1.3 table S can be referred to as an four dimensional array. We will now state the number of entries of each dimension depending on the different parameters.

- The key for the first dimension is a character c. Since the are five different ones we have five different entries.
- The key for the second dimension is the length of a substring in a bucket.
- BZ_j is used as a key for the third dimension and represents a binary string of size l'. l' depends on the used encoding scheme. In this case $l' = (1 + 2\lfloor log\Sigma \rfloor) * l$. Note that [3] states that each possible compressed bucket BZ_j will be represented.
- The last dimension represents all possible MTF list states, which are $|\Sigma|^{|\Sigma|}$ many.

Table 1 summarizes the observation mentioned above and states explicit memory consumptions for different scenarios.

Memory Consumption Excluding Table S The previous section showed that the memory consumption of table S is too large to be efficiently handled. Nevertheless we will now analyse the memory consumption of the remaining FM-index since we can trade the storage of S with a small speed trade off (see section 2.2 for details).

text length	bucket size	number of buckets	number of super buckets	memory consumption [MB] without W	with W
3000	8	375	46.88	0.0032	0.0038
3000000	8	375000	46875	5.8008	6.9609
3000000000	8	375000000	46875000	8437.5000	10125.0000
3000	16	187.5	11.72	0.0015	0.0018
3000000	16	187500	11718.75	2.7393	3.2871
3000000000	16	187500000	11718750	3984.3750	4781.2500
3000	32	93.75	2.93	0.0007	0.0009
3000000	32	93750	2929.69	1.3293	1.5952
3000000000	32	93750000	2929687.5	1933.5938	2320.3125
3000	64	46.88	0.73	0.0004	0.0004
3000000	64	46875	732.42	0.6546	0.7855
3000000000	64	46875000	732421.88	952.1484	1142.5781

Table 2: Memory consumption for the bucket structure for several chosen bucket sizes and text length.

Table 2 shows the memory consumption for the described bucket structure without table S, the compressed String and suffix array for several bucket sizes and text length.

Table 2 showed that it is feasible to store the bucket structure in main memory and therefore guarantee fast access.

2.1.4 Searching the compressed BWT

In order to return the exact pattern positions in the text Ferragina and Manzini marked every *nth* position, with $\eta = \theta(log^2 n)$ and n being the length of the text. At query time it is tested whether s, with sp <= s <= ep, is a marked row. If this is the case there is nothing to be done and the correct text position can be returned. Otherwise the LF-mapping is used to find a marked row and the text position can be computed. Note that sp and ep were determined in 2.1.3.

2.2 A Practical Implementation of the FM-Index

- An experimental study of an opportunistic index [2]

In [2] Ferragina and Manzini present a practical solution of the FM-index described above. In contrast to original FM-index this practical solution does not store the table S. In doing so they are able to reduce the memory consumption of the index at the cost of speed at query time. In order to compute the occurrence of a given character c at a given location *loc* the substring of the bucket containing *loc* is decompressed and the number of occurrences of c from the beginning of the bucket to *loc* are counted.

2.3 An alphabet friendly FM-index

- An alphabet friendly FM-index [4]

The original FM-index is not particular alphabet friendly. The reason for this is a dependence on the alphabet size when accessing an element of the compressed text. Instead of decompressing the text in the corresponding bucket one uses the occurrence function. This is possible since *occ(i,c)* and *occ(i+1,c)* are only different if c appears at position i in T*. Using this method one has to go through the whole alphabet in order to determine c in the worst case. However, Ferragina and Manzini developed an alphabet friendly version of the index which will be described in the following.

In contrast to the already described implementations of the index in section 2.1 and 2.2 the compression scheme of the alphabet friendly FM-index is based on wavelet trees. Instead of using MTF encoding, run-length encoding and a variable-length-prefix code a wavelet tree is constructed for every bucket. The wavelet tree can then be used to search in O(m $log|\Sigma|$) for occurrences of a given pattern.

In order to decide which structure to use one needs to estimate the memory consumption of the compressed text. While the compression of the original proposed compression scheme greatly depends on the entropy of the text the compressed text using wavelet tress needs exactly $n * log|Sigma|$ bits plus some auxiliary memory for the tree structure.

The two compression schemes result in 11.7 MB and 12.8 MB respectively usage of memory for the human chromosome 22. Note that the auxiliary memory consumption for the tree structure is not included in the 12.8 MB. However, the result shows that the wavelet tree does not use much more memory than the original compression scheme. In fact, the wavelet tree does not need a hash table to calculate the number of occurrences of a given character at a given location in constant time. One can use the bucket structure described above and use constantly many bit operations[1] to achieve a O(1) runtime.

References

[1] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.

[2] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, pages 269–278, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[3] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52:552–581, July 2005.

[4] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly fm-index. In Alberto Apostolico and Massimo Melucci, editors, *String Processing and Information Retrieval*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

[1] http://graphics.stanford.edu/ seander/bithacks.html#CountBitsSetParallel