FM-index

From Wikipedia, the free encyclopedia

In computer science, an **FM-index** is a compressed full-text substring index based on the Burrows-Wheeler transform, with some similarities to the suffix array. It was created by Paolo Ferragina and Giovanni Manzini,[1] who describe it as an opportunistic data structure as it allows compression of the input text while still permitting fast substring queries. The name stands for Full-text index in Minute space.[2]

It can be used to efficiently find the number of occurrences of a pattern within the compressed text, as well as locate the position of each occurrence. Both the query time and storage space requirements are sublinear with respect to the size of the input data.

The original authors have devised improvements to their original approach and dubbed it "FM-Index version 2".[3] A further improvement, the alphabet-friendly FM-index, combines the use of compression boosting and wavelet trees [4] to significantly reduce the space usage for large alphabets.

The FM-index has found use in, among other places, bioinformatics.[5]

Contents [hide]

Background [edit]

Using an index is a common strategy to efficiently search a large body of text. When the text is larger than what reasonably fits within a computer's main memory, there is a need to compress not only the text but also the index. When the FM-index was introduced, there were several suggested solutions that were based on traditional compression methods and tried to solve the compressed matching problem. In contrast, the FM-index is a compressed self-index, which means that it compresses the data and indexes it at the same time.

FM-index data structure [edit]

An FM-index is created by first taking the Burrows-Wheeler transform (BWT) of the input text. For example, the BWT of the string $T =$ "abracadabra" is "ard$rcaaaabb", and here it is represented by the matrix M where each row is a rotation of the text, and the rows have been sorted lexicographically. The transform corresponds to the last column labeled L.

I	F		L
1	$	abracadabr	a
2	a	$abracadab	r
3	a	bra$abraca	d
4	a	bracadabra	$
5	a	cadabra$ab	r
6	a	dabra$abra	c
7	b	ra$abracad	a
8	b	racadabra$	a
9	c	adabra$abr	a
10	d	abra$abrac	a
11	r	a$abracada	b
12	r	acadabra$a	b

The BWT in itself allows for some compression with, for instance, move to front and Huffman encoding, but the transform has even more uses. The rows in the matrix are essentially the sorted suffixes of the text and the first column F of the matrix shares similarities with suffix arrays. How the suffix array relates to the BWT lies at the heart of the FM-index.

It is possible to make a last-to-first column mapping $\mathrm{LF}(i)$ from a an index i to an index j, such that $F[j] = L[i]$, with the help of a table $C[c]$ and a function $\mathrm{Occ}(c, k)$.

- $C[c]$ is a table that, for each character c in the alphabet, contains the number of occurrences of lexically smaller characters in the text.
- The function $\mathrm{Occ}(c, k)$ is the number of occurrences of character c in the prefix $L[1..k]$. Ferragina and Manzini showed[1] that it is possible to compute $\mathrm{Occ}(c, k)$ in constant time.

$C[c]$ of "ard$rcaaaabb"

c	$	a	b	c	d	r
$C[c]$	0	1	6	8	9	10

The last-to-first mapping can now be defined as $\mathrm{LF}(i) = C[L[i]] + \mathrm{Occ}(L[i], i)$. For instance, on row 9, L is a and the same a can be found on row 5 in the first column F, so $\mathrm{LF}(9)$ should be 5 and $\mathrm{LF}(9) = C[a] + \mathrm{Occ}(a, 9) = 5$. For any row i of the matrix, the character in the last column $L[i]$ precedes the character in the first column $F[i]$ also in T. Finally, if $L[i] = T[k]$, then $L[\mathrm{LF}(i)] = T[k - 1]$, and using the equality it is possible to extract a string of T from L.

The FM-index itself is a compression of the string L together with C and Occ in some form, as well as information that maps a selection of indices in L to positions in the original string T.

$\mathrm{Occ}(c, k)$ of "ard$rcaaaabb"

	a	**r**	**d**	**$**	**r**	**c**	**a**	**a**	**a**	**a**	**b**	**b**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
$	0	0	0	1	1	1	1	1	1	1	1	1
a	1	1	1	1	1	1	2	3	4	5	5	5
b	0	0	0	0	0	0	0	0	0	0	1	2
c	0	0	0	0	0	1	1	1	1	1	1	1
d	0	0	1	1	1	1	1	1	1	1	1	1
r	0	1	1	1	2	2	2	2	2	2	2	2

Count [edit]

The operation *count* takes a pattern $P[1..p]$ and returns the number of occurrences of that pattern in the original text T. Since the rows of matrix M are sorted, and it contains every suffix of T, the occurrences of pattern P will be next to each other in a single continuous range. The operation iterates backwards over the pattern. For every character in the pattern, the range that has the character as a suffix is found. For example, the count of the pattern "bra" in "abracadabra" follows these steps:

1. The first character we look for is a, the last character in the pattern. The initial range is set to $[C[a] + 1..C[a{+}1]] = [2..6]$. This range over L represents every character of T that has a suffix beginning with *a*.
2. The next character to look for is r. The new range is $[C[r] + \mathrm{Occ}(r, start\text{-}1) + 1..C[r] + \mathrm{Occ}(r, end)] = [10 + 0 + 1..10 + 2] = [11..12]$, if start is the index of the beginning of the range and end is the end. This range over L is all the characters of T that have suffixes beginning with *ra*.
3. The last character to look at is b. The new range is $[C[b] + \mathrm{Occ}(b, start\text{-}1) + 1..C[b] + \mathrm{Occ}(b, end)] = [6 + 0 + 1..6 + 2] = [7..8]$. This range over L is all the characters that have a suffix that begins with *bra*. Now that the whole pattern has been processed, the count is the same as the size of the range: $8 - 7 + 1 = 2$.

If the range becomes empty or the range boundaries cross each other before the whole pattern has been looked up, the pattern does not occur in T. Because $\mathrm{Occ}(c, k)$ can be performed in constant time, count can complete in linear time in the length of the pattern: $O(p)$ time.

Locate [edit]

The operation *locate* takes as input an index of a character in L and returns its position i in T. For instance $\mathrm{locate}(7) = 8$. To locate every occurrence of a pattern, first the range of character is found whose suffix is the pattern in the same way the *count* operation found the range. Then the position of every character in the range can be located.

To map an index in L to one in T, a subset of the indices in L are associated with a position in T. If $L[j]$ has a position associated with it, $\mathrm{locate}(j)$ is trivial. If it's not associated, the string is followed with $\mathrm{LF}(i)$ until an associated index is found. By associating a suitable number of indices, an upper bound can be found. *Locate* can be implemented to find *occ* occurrences of a pattern $P[1..p]$ in a text $T[1..u]$ in $O(p + occ \log^{\varepsilon} u)$ time with $O(H_k(T) + \frac{\log \log u}{\log^{\epsilon} u})$ bits per input symbol for any $k \geq 0$.[1]

Applications [edit]

DNA read mapping [edit]

FM index with Backtracking has been successfully (>2000 citations) applied to approximate string matching/sequence alignment, See Bowtie http://bowtie-bio.sourceforge.net/index.shtml

See also [edit]

- Burrows–Wheeler transform
- Suffix array
- Compressed suffix array
- Sequence alignment

References [edit]

1. ^ a b c Paolo Ferragina and Giovanni Manzini (2000). "Opportunistic Data Structures with Applications". Proceedings of the 41st Annual Symposium on Foundations of Computer Science. p.390.
2. ^ Paolo Ferragina and Giovanni Manzini (2005). "Indexing Compressed Text". Journal of the ACM (JACM), 52, 4 (Jul. 2005). p. 553
3. ^ Paolo Ferragina and Rossano Venturini "FM-Index version 2"
4. ^ P. Ferragina, G. Manzini, V. Mäkinen and G. Navarro. An Alphabet-Friendly FM-index. *In Proc. SPIRE'04*, pages 150-160. LNCS 3246.
5. ^ Simpson, Jared T. and Durbin, Richard (2010). "Efficient construction of an assembly string graph using the FM-index". Bioinformatics, 26, 12 (Jun. 17). p. i367

Categories: Substring indices | String data structures