11.1 Compressing the FM Index

This exposition has been developed by David Weese. It is based on the following sources, which are all recommended reading:

1. P. Ferragina, G. Manzini (2000) *Opportunistic data structures with applications*, Proceedings of the 41st IEEE Symposium on Foundations of Computer Science
2. P. Ferragina, G. Manzini (2001) *An experimental study of an opportunistic index*, Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms, pp. 269-278
3. Johannes Fischer (2010), *Skriptum VL Text-Indexierung*, SoSe 2010, KIT
4. A. Andersson (1996) *Sorting and searching revisited*, Proceedings of the 5th Scandinavian Workshop on Algorithm Theory, pp. 185-197

11.2 RAM Model

From now on we assume the *RAM model* in which we model a computer with a CPU that has registers of w bits which can be modified with logical and arithmetical operations in $O(1)$ time. The CPU can directly access a memory of at most 2^w words.

In the following we assume $n \leq 2^w$ so that it is possible to address the whole input. To have a more precise measure, we count memory consumptions in bits. The uncompressed suffix array then does not require $O(n)$ memory but $O(n \log n)$ bits, as $\lceil \log_2 n \rceil$ bits are required to represent any number in $[1..n]$.

11.3 Tables of the FM Index

Let T be a text of length n over the alphabet Σ and $\sigma = |\Sigma|$ be the alphabet size. We have seen, that for the algorithms **count** and **locate** we need L and the tables C and Occ. Without compression their memory consumption is as follows:

- $L = T^{\text{bwt}}$ is a string of length n over Σ and requires $O(n \log \sigma)$ bits
- C is an array of length σ over $[0..n]$ and requires $O(\sigma \log n)$ bits
- Occ is an array of length $\sigma \times n$ over $[0..n]$ and requires $O(\sigma \cdot n \log n)$ bits
- *pos* (if every row is marked) is a suffix array of length n over $[1..n]$ and requires $O(n \log n)$ bits

We will present approaches to compress L, Occ and *pos*, but omit to compress C assuming that σ and $\log n$ are tolerably small.

11.4 Compressing L

Burrows and Wheeler proposed a move-to-front coding in combination with Huffman or arithmetic coding. In the context of the move-to-front encoding each character is encoded by its index in a list, which changes over the course of the algorithm. It works as follows:

1. Initialize a list Y of characters to contain each character in Σ exactly once
2. Scan L with $i = 1, \ldots, n$
    (a) Set $R[i]$ to the number of characters preceding character $L[i]$ in the list Y
    (b) Move character $L[i]$ to the front of Y

R is the MTF encoding of L. R can again be decoded to L in a similar way (Exercise).

Algorithm **move_to_front(L)** shows the pseudo-code of the move-to-front encoding. The array M maintains for every alphabet character the number preceding characters in Y instead of using Y directly.

```
// move_to_front(L)
for $j = 1$ to $\sigma$ do
    $M[j] = j - 1$
od
for $i = 1$ to $n$ do
    // ord maps a character to its rank in the alphabet
    $x = \text{ord}(L[i])$
    $R[i] = M[x]$;
    for $j = 1$ to $\sigma$ do
        if $M[j] < M[x]$ then $M[j] = M[j] + 1$; fi
    od
    $M[x] = 0$;
od
return $R$;
```

Observation 1. The BWT tends to group characters together so that the probability of finding a character close to another instance of the same character is increased substantially:

final char (L)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set $L[i]$ to be the
i	n turn, set $R[i]$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector $R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with ch appear in the {\em same order
i	n with ch. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell~\cite{bell}.

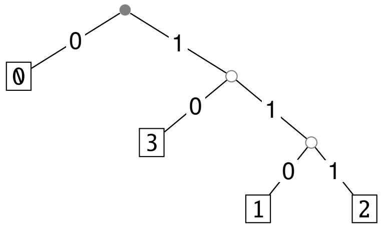
Observation 2. The move-to-front encoding replaces equal characters that in L are "close together" by "small values" in R. In practice, the most important effect is that zeroes tend to occur in runs in R. These can be compressed using an order-0 compressor, e.g. the Huffman encoding.

i	$L[i]$	$R[i]$	Y_{next}
			aeio
1	a	0	aeio
2	o	3	oaei
3	o	0	oaei
4	o	0	oaei
5	o	0	oaei
6	a	1	aoei
7	a	0	aoei
8	i	3	iaoe
9	i	0	iaoe
10	o	2	oiae
11	a	2	aoie
12	e	3	eaoi
13	i	3	ieao
14	e	1	eiao
15	e	0	eiao
16	i	1	ieao
17	i	0	ieao

...

The Huffman encoding builds a binary tree where leaves are alphabet characters. The tree is balanced such that for every node the leaves in the left and right subtree have a similar sum of occurrences.

character	0	1	2	3
occurrences in R	10	3	2	5



x	bit code of x
0	0
1	110
2	111
3	10

Left and right childs are labeled with 0 and 1. The labels on the paths to each leaf define its bit code. The more frequent a character the shorter its bit code. The final sequence H is the bitwise concatenation of bit codes of characters from left to right in R.

The final sequence of bits H is:

$$\begin{aligned} L &= \texttt{aooooaaii...} \\ R &= \texttt{030001030...} \\ H &= \texttt{0100001100100...} \end{aligned}$$

One property of the MTF coding is that the whole prefix $R[1..i-1]$ is required to decode character $R[i]$, the same holds for H. Thus the random accesses to L in algorithm **locate** would take $O(n)$ time. To avoid decompressing from the beginning of H we divide L into blocks of equal length ℓ and compress each block separately.

However, this approach still takes $O(n/\ell)$ time to access L. By a simple trick we can determine $L[i]$ using the Occ function. Clearly, the values $\mathsf{Occ}(c,i)$ and $\mathsf{Occ}(c,i-1)$ differ only for $c = L[i]$. Thus we can determine both $L[i]$ and $\mathsf{Occ}(L[i],i)$ using σ Occ-qeries, which we will see take in sum $O(\sigma)$ time. Using wavelet trees this time can even be reduced to $O(\log\sigma)$.

11.5 Compressing Occ

We reduce the problem of counting the occurrences of a character in a prefix of L to counting 1's in a prefix of a bitvector. Therefore we construct a bitvector B_c of length n for each $c \in \Sigma$ such that:

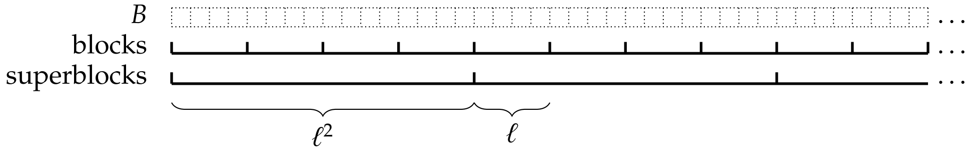
$$B_c[i] = \begin{cases} 1 & \text{if } L[i] = c \\ 0 & \text{else} \end{cases} \quad .$$

Definition 3. For a bitvector B we define $rank_1(B,i)$ to be the number of 1's in the prefix $B[1..i]$. $rank_0(B,i)$ is defined analogously.

As each 1 in the bitvector B_c indicates an occurrence of c in L, it holds:

$$\mathsf{Occ}(c,i) = rank_1(B_c,i) \quad .$$

We will see that it is possible to answer a rank query of a bitvector of length n in constant time using additional tables of $o(n)$ bits. Hence the σ bitvectors are an implementation of Occ that allows to answer Occ queries in constant time with an overall memory consumption of $O(\sigma n + o(\sigma n))$ bits. Given a bitvector $B = B[1..n]$. We compute the length $\ell = \left\lfloor \frac{\log n}{2} \right\rfloor$ and divide B into blocks of length ℓ and superblocks of length ℓ^2.



1. For the i-th superblock we count the number of 1's from the beginning of B to the end of the superblock in $M'[i] = rank_1(B, i\cdot\ell^2)$. As there are $\left\lfloor \frac{n}{\ell^2} \right\rfloor$ superblocks, M' can be stored in $O\left(\frac{n}{\ell^2}\cdot\log n\right) = O\left(\frac{n}{\log n}\right) = o(n)$ bits.
2. For the i-th block we count the number of 1's from the beginning of the overlapping superblock to the end of the block in $M[i] = rank_1\big(B[1+k\ell..n], (i-k)\ell\big)$ where $k = \left\lfloor \frac{i-1}{\ell} \right\rfloor \ell$ is the number of blocks left of the overlapping superblock. M has $\left\lfloor \frac{n}{\ell} \right\rfloor$ entries and can be stored in $O\left(\frac{n}{\ell}\cdot\log\ell^2\right) = O\left(\frac{n\log\log n}{\log n}\right) = o(n)$ bits.

3. Let P be a precomputed lookup table such that for each possible bitvector V of length ℓ and $i \in [1..\ell]$ holds $P[V][i] = rank_1(V, i)$. V has $2^\ell \times \ell$ entries of values at most ℓ and thus can be stored in
$$O\left(2^\ell \cdot \ell \cdot \log \ell\right) = O\left(2^{\frac{\log n}{2}} \cdot \log n \cdot \log \log n\right) = O\left(\sqrt{n} \log n \log \log n\right) = o(n)$$
bits.

We now decompose a rank-query into 3 subqueries using the precomputed tables. For a position i we determine the index $p = \left\lfloor \frac{i-1}{\ell} \right\rfloor$ of next block left of i and the index $q = \left\lfloor \frac{p-1}{\ell} \right\rfloor$ of the next superblock left of block p. Then it holds:
$$rank_1(B, i) = M'[q] + M[p] + P\Big[B[1 + p\ell..(p+1)\ell]\Big]\Big[i - p\ell\Big] \quad .$$

Note that $B[1 + p\ell..(p+1)\ell]$ fits into a single CPU register and can therefore be determined in $O(1)$ time. Thus a rank-query can be answered in $O(1)$ time.

11.6 Compressing *pos*

To compress *pos* we mark only a subset of rows in the matrix $\mathcal{M}$ and store their text positions. Therefore we need a data structure that efficiently decides wether a row $\mathcal{M}_i = T[j]$ is marked and that retrieves j for a marked row i.

If we would mark every η-th row in the matrix ($\eta > 1$) we could easily decide whether row i is marked, e. g. iff $i \equiv 1 \pmod{\eta}$. Unfortunately this approach still has worst-cases where a single *pos*-query takes $O\left(\frac{\eta-1}{\eta} n\right)$ time (excercise).

Instead we mark the matrix row for every η-th text position, i. e. for all $j \in \left[0..\lceil \frac{n}{\eta} \rceil\right)$ row i with $\mathcal{M}_i = T^{(1+j\eta)}$ is marked with the text position $pos(i) = 1 + j\eta$. To determine whether a row is marked we could store all marked pairs $(i, 1 + j\eta)$ in a hash map or a binary search tree with key i. Ferragina and Manzini proposed a different approach. They marked every η-th text position for $\eta = \Theta(\log^2 n)$ and divided the matrix in buckets of η adjacent rows. For each marked row they recorded the row offset to the first row of the bucket. This offset takes $O(\log \eta) = O(\log \log n)$ bits.

As each bucket has at most η marked rows they use a *packet B-tree* (Appendix) of $u = O(\log^2 n)$ keys of size $k = O(\log \log n)$ bits. This B-tree supports membership queries in $O(\log_{w/k} u) = O\left(\frac{\log \log n}{\log \log n - \log \log \log n}\right) = O(1)$ time.

Each packet B-tree uses space proportional to the number of stored keys. Hence the *pos* data structure has an overall space consumption of $O\left(\frac{n}{\eta}(\log \log n + \log n)\right)$ bits since with each marked row $\mathcal{M}_i$ they also keep the value $pos(i)$ using $O(\log n)$ bits.

11.7 Appendix: Packed B-tree

A *packed B-tree* (Andersson 1996) is a balanced search tree whose nodes store keys of k bits length. Inner nodes store $t = \left\lfloor \frac{w}{k+1} \right\rfloor$ keys $y_1 < y_2 < \ldots < y_t$ in sorted order and have $t + 1$ children. It is easy to see that searching a node in a tree of u leaves then takes $O(\log t \cdot \log_{t+1} u)$ as the tree height is $O(\log_{t+1} u)$ and at each level, the search chooses the child pointer (subtree) whose separation values are on either side of the search value in $O(\log t)$ (binary search on node keys).

The main trick of the packed B-tree is to replace the binary search of the child pointer by a bit-parallel comparison of all the keys in constant time. Let the sorted keys $y_1 < \ldots < y_t$ of a node be stored in a register Y ascending from left to right each separated by a 1 bit. Let X be a register with a similar layout storing t copies of the search key x each separated by a 0 bit. For the difference $Y - X$ holds that the bit at the separating position left of node key y_i is 0 iff $y_i < x$. If we clear all but the separating bits by logically AND'ing $(Y - X)$ with a corresponding mask M, the result contains a sequence of r 0 bits followed by $t - r$ 1 bits, where r is the rank of x among the keys $y_1, \ldots, y_t$.

Y	1	00010	1	00111	1	01001	1	01110	1	10101	1	11000	1	11011	1	11110
X	0	01011	0	01011	0	01011	0	01011	0	01011	0	01011	0	01011	0	01011
$Y - X$	0	10111	0	11100	0	11110	1	00011	1	01010	1	01101	1	10000	0	10011
M	1	00000	1	00000	1	00000	1	00000	1	00000	1	00000	1	00000	1	00000
$(Y - X)$ AND M	0	00000	0	00000	0	00000	1	00000	1	00000	1	00000	1	00000	1	00000

Lemma 4. *The rank of x among the keys of a node can be determined in $\mathcal{O}(1)$.*

Proof: M can be precomputed. X can be computed from the search key x by $X = x * (M >> t)$. We compute $(Y - X)$ AND M in $\mathcal{O}(1)$. The rank of x can then be determined in $\mathcal{O}(1)$ by looking up the result in a precomputed lookup table.

Hence, the whole packed B-tree search takes $\mathcal{O}(\log_{t+1} u) = \mathcal{O}(\log_{w/k} u)$ time.