First Huffman, then Burrows-Wheeler: A Simple Alphabet-Independent FM-Index

Szymon Grabowski[1], Veli Mäkinen[2], and Gonzalo Navarro[3]

[1] Computer Engineering Dept., Tech. Univ. of Łódź, Poland.
[2] Dept. of Computer Science, Univ. of Helsinki, Finland.
[3] Dept. of Computer Science, Univ. of Chile, Chile.

Abstract. We design a succinct full-text index based on the idea of Huffman-compressing the text and then applying the Burrows-Wheeler transform over it. The resulting structure can be searched as an FM-index, with the benefit of removing the sharp dependence on the alphabet size, σ, present in that structure. On a text of length n with zero-order entropy H_0, our index needs $O(n(H_0+1))$ bits of space, without any dependence on σ. The average search time for a pattern of length m is $O(m(H_0+1))$, under reasonable assumptions. Each position of a text occurrence can be reported in worst case time $O((H_0+1)\log n)$, while any text substring of length L can be retrieved in $O((H_0+1)L)$ average time in addition to the previous worst case time. By paying $2n$ additional bits of space, it is possible to maintain the average complexities and ensure that H_0+1 becomes $\log\sigma$ in the worst case for all the time complexities. Our index provides a relevant space/time tradeoff between existing succinct data structures, with the additional interest of being easy to implement.

1 Introduction

The classical problem in string matching is to determine the *occ* occurrences of a short pattern $P = p_1p_2\ldots p_m$ in a large text $T = t_1t_2\ldots t_n$. Text and pattern are sequences of characters over an alphabet Σ of size σ. In practice one wants to know the text positions of those *occ* occurrences, and usually also a text context around them. Usually the same text is queried several times with different patterns, and therefore it is worthwhile to preprocess the text in order to speed up the searches. Preprocessing builds an index structure for the text.

To allow fast searches for patterns of any size, the index must allow access to all suffixes of the text (the ith suffix of T is $t_it_{i+1}\ldots t_n$). These kind of indexes are called *full-text indexes*. Optimal query time, which is $O(m+occ)$ as every character of P must be examined and the *occ* occurrences must be reported, can be achieved by using the *suffix tree* [21, 12, 20] as the index.

The suffix tree takes much more memory than the text. In general, it takes $O(n\log n)$ bits, as the text takes $n\log\sigma$ bits[4]. A smaller constant factor is achieved by the *suffix array* [11]. Still, the space complexity does not change. Moreover, the searches take $O(m\log n+occ)$ time with the suffix array (this can be improved to $O(m+\log n+occ)$ using twice the original amount of space [11]).

The large space requirement of full-text indexes has raised a natural interest for an index that occupies the same amount of space as the text itself. There are several so-called *succinct* full-text indexes that achieve good tradeoffs between search time and space complexity [10, 3, 7, 18, 19, 5, 14, 17, 15, 6]. Most of these are *opportunistic* as they take less space than the text itself, and also *self-indexes* as they contain enough information to reproduce the text: A self-index does not need the text to operate.

[4] By log we mean $\log_2$ in this paper.

For example, the FM-index of Ferragina and Manzini [3] is a self-index that takes in practice the same amount of space as the *compressed* text. The size of the index is in fact close to the best compression ratios achievable. The space complexity, however, contains an exponential dependence on the alphabet size. A practical implementation [4] avoids this by using heuristics with the side effect of not achieving the optimal search time anymore.

In this paper we concentrate on improving the FM-index, in particular its large alphabet dependence. This dependence shows up not only in the space usage, but also in the time to show an occurrence position and display text substrings. The FM-index needs up to $5H_k n + O\left((\sigma \log \sigma + \log\log n)\frac{n}{\log n} + n^\gamma \sigma^{\sigma+1}\right)$ bits of space, where $0 < \gamma < 1$. The time to search for a pattern and obtain the number of its occurrences in the text is the optimal $O(m)$. The text position of each occurrence can be found in $O\left(\sigma \ \log^{1+\varepsilon} n\right)$ time, for some $\varepsilon > 0$ that appears in the sublinear terms of the space complexity. Finally, the time to display a text substring of length L is $O\left(\sigma \ (L + \log^{1+\varepsilon} n)\right)$. The latter operation is important not only to show a text context around each occurrence, but also because a self-index replaces the text and hence it must provide the functionality of retrieving any desired text substring.

The compressed suffix array (CSA) of Sadakane [18] can be seen as a tradeoff with larger search time but much milder dependence on the alphabet size in all the other aspects. The CSA needs $(H_0/\varepsilon + O(\log\log\sigma))n$ bits of space. Its search time (finding the number of occurrences of a pattern) is $O(m \log n)$. Each occurrence can be reported in $O\left(\log^\varepsilon n\right)$ time, and a text substring of length L can be displayed in $O\left(L + \log^\varepsilon n\right)$ time.

In this paper we present a simple structure based on the FM-index concept. We Huffman-compress the text and then apply the Burrows-Wheeler transform over it, as in the FM-index. The obtained structure can be regarded as an FM-index built over a binary sequence. As a result, we remove any dependence on the alphabet size. We show that our index can operate using $n(2H_0 + 3 + \varepsilon)(1 + o(1))$ bits, for any $\varepsilon > 0$. No alphabet dependence is hidden in the sublinear terms.

At search time, our index finds the number of occurrences of the pattern in $O(m(H_0+1))$ average time. The text position of each occurrence can be reported in worst case time $O\left(\frac{1}{\varepsilon}(H_0 + 1) \ \log n\right)$. Any text substring of length L can be displayed in $O\left(\frac{1}{\varepsilon}(H_0 + 1)L)\right)$ average time, in addition to the mentioned worst case time to find a text position.

We show that, by using $n(2H_0 + 5 + \varepsilon)$ bits of space, we can retain the average time complexities while ensuring a reasonable worst case, obtained by replacing H_0+1 by $\log\sigma$ in all time complexities.

Compared to the CSA, we have similar space usage and better search complexity. Compared to the FM-index, we have better complexities to report occurrences and display text substrings.

Very recently [16], we have proposed a completely different index structure with similar space and time complexities. The present proposal has a larger constant factor in the space complexity. In exchange, it is considerably simpler to implement.

2 The FM-index Structure

The FM-index [3] is based on the *Burrows-Wheeler transform (BWT)* [1], which produces a permutation of the original text, denoted by $T^{bwt} = bwt(T)$. String T^{bwt} is a result of the following *forward* transformation: (1) Append to the end of T a special end marker \$, which is lexicographically smaller than any other character; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic

shifts of the string $T\$$, sorted in lexicographic order; (3) construct the transformed text L by taking the last column of $\mathcal{M}$. The first column is denoted by F.

The *suffix array (SA)* $\mathcal{A}$ of text $T\$$ is essentially the matrix $\mathcal{M}$: $\mathcal{A}[i] = j$ iff the ith row of $\mathcal{M}$ contains string $t_j t_{j+1} \cdots t_n \$ t_1 \cdots t_{j-1}$. Given the suffix array, the search for the occurrences of the pattern $P = p_1 p_2 \cdots p_m$ is trivial. The occurrences form an interval $[sp, ep]$ in $\mathcal{A}$ such that suffixes $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \cdots t_n$, $sp \leq i \leq ep$, contain the pattern as a prefix. This interval can be searched for using two binary searches in time $O(m \log n)$.

The suffix array of text T is represented implicitly by T^{bwt}. The novel idea of the FM-index is to store T^{bwt} in compressed form, and to simulate the search in the suffix array. To describe the search algorithm, we need to introduce the *backward* BWT that produces T given T^{bwt}:

1. Compute the array $C[1 \ldots \sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\$, 1, \ldots, c-1\}$ in the text T. Notice that $C[c] + 1$ is the position of the first occurrence of c in F (if any).
2. Define the *LF-mapping* $LF[1 \ldots n+1]$ as $LF[i] = C[L[i]] + Occ(L, L[i], i)$, where $Occ(X, c, i)$ equals the number of occurrences of character c in the prefix $X[1, i]$.
3. Reconstruct T backwards as follows: set $s = 1$ and $T[n] = L[1]$ (because $\mathcal{M}[1] = \$T$); then, for each $n-1, \ldots, 1$ do $s \leftarrow LF[s]$ and $T[i] \leftarrow L[s]$.

We are now ready to describe the search algorithm given in [3] (Fig. 1). It finds the interval of $\mathcal{A}$ containing the occurrences of the pattern P. It uses the array C and function $Occ(X, c, i)$ defined above. Using the properties of the backward BWT, it is easy to see that the algorithm maintains the following invariant [3]: *At the ith phase, the variable sp points to the first row of $\mathcal{M}$ prefixed by $P[i, m]$ and the variable ep points to the last row of $\mathcal{M}$ prefixed by $P[i, m]$*. The correctness of the algorithm follows from this observation.

Algorithm FM_Search(P,T^{bwt})
(1) $i = m$;
(2) $sp = 1$; $ep = n$;
(3) **while** (($sp \leq ep$) **and** ($i \geq 1$) **do**
(4) $\quad c = P[i-1]$;
(5) $\quad sp = C[c] + Occ(T^{bwt}, c, sp-1)+1$;
(6) $\quad ep = C[c] + Occ(T^{bwt}, c, ep)$;
(7) $\quad i = i - 1$;
(8) **if** ($ep < sp$) **then return** "not found" **else return** "found ($ep - sp + 1$) occs".

Fig. 1. Algorithm for counting the number of occurrences of $P[1 \ldots m]$ in $T[1 \ldots n]$.

Ferragina and Manzini [3] describe an implementation of $Occ(T^{bwt}, c, i)$ that uses a compressed form of T^{bwt}. They show how to compute $Occ(T^{bwt}, c, i)$ for any c and i in constant time. However, to achieve this they need exponential space (in the size of the alphabet). In a practical implementation [4] this was avoided, but the constant time guarantee for answering $Occ(T^{bwt}, c, i)$ was no longer valid.

The FM-index can also show the text positions where P occurs, and display any text substring. The details are deferred to Section 6.

3 Rank and Select Queries on Bit Arrays

A building block we use is a data structure to perform *rank* and *select* operations over a bit array. In this section we review the existing solution for *rank*, which is rather practical and easy to implement, and propose a practical implementation for a restricted version of *select*, the one we need in this paper.

3.1 Rank Queries

Given a bit sequence $B[1 \dots n]$, $rank(B, i)$ is the number of 1's in $B[1 \dots i]$, $rank(B, 0) = 0$. This function can be computed in constant time using only $o(n)$ extra bits [9, 13, 2]. We briefly describe such structure now.

We divide the bit array into blocks of length $b = \lfloor \log(n)/2 \rfloor$. Consecutive blocks are grouped into superblocks of length $s = b\lfloor \log n \rfloor$.

For each superblock j, $j = 0 \dots \lfloor n/s \rfloor$ we store a number $R_s[j] = rank(B, j \cdot s)$. Array R_s needs overall $O(n/\log n)$ bits since each $R_s[j]$ value needs $\log n$ bits and there are $n/s = O(n/\log^2 n)$ entries.

For each block k of superblock $j = k \text{ div } \lfloor \log n \rfloor$, $k = 0 \dots \lfloor n/b \rfloor$ we store a number $R_b[k] = rank(B, k \cdot b) - rank(B, j \cdot s)$. Array R_b needs $O(n \log\log n/\log n)$ bits since each $R_b[k]$ value needs $O(\log\log n)$ bits. The reason is that it represents the number of bits set inside a superblock of length $O(\log^2 n)$, and therefore $O(\log\log n)$ bits suffice for it. On the other hand, there are $n/b = O(n/\log n)$ blocks overall.

Finally, for every bit stream S of length b and for every position i inside S, we precompute $R_p[S, i] = rank(S, i)$. This requires $O(2^b \cdot b \cdot \log b) = O(\sqrt{n} \log n \log\log n)$ bits.

The above structures need $O(n/\log n + n \log\log n/\log n + \sqrt{n} \log n \log\log n) = o(n)$ bits. They permit computing *rank* in constant time as follows:

$$rank(B, i) = R_s[i \text{ div } s] + R_b[i \text{ div } b] + R_p[B[(i \text{ div } b) \cdot b + 1 \dots (i \text{ div } b) \cdot b + b], i \bmod b]$$

This structure is rather practical, and can be implemented with little effort.

3.2 Select and SelectNext Queries

The inverse function, $select(B, j)$, gives the position of the j-th bit set in B. It can also be implemented in constant time using $o(n)$ additional space [9, 13, 2]. However, it is much more complicated and less practical.

We propose now a simple and practical version of *select*, which is restricted to find the next bit set in B starting at some position. More precisely, given bit sequence B such that $n = |B|$, $selectnext(B, j)$ is the position of the first bit set in $B[j \dots n]$, and $n + 1$ if no such bit set exists.

We divide B as before, into blocks and superblocks of sizes b and s, respectively. For each superblock j, $j = 1 \dots \lfloor n/s \rfloor$ we store a number $N_s[j] = selectnext(B, j \cdot s + 1)$. Array N_s needs overall $O(n/\log n)$ bits since each $N_s[j]$ value needs $O(\log n)$ bits.

For each block k of superblock $j = k \text{ div } \lfloor \log n \rfloor$, $k = 1 \dots \lfloor n/b \rfloor$, we store a number $N_b[k] = selectnext(B[j \cdot s + 1 \dots (j + 1) \cdot s], k \cdot b - j \cdot s + 1)$. Array N_b needs overall $O(n \log\log n/\log n)$

bits since each $N_b[k]$ value needs $O(\log\log n)$ bits, as it represents a position inside a superblock of length $O(\log^2 n)$.

Finally, for every bit stream S of length b and for every position i inside S, we precompute $N_p[S,i] = \mathit{selectnext}(S,i)$. This requires $O(2^b \cdot b \cdot \log b) = O(\sqrt{n}\log n \log\log n)$ bits. Note that $N_p[S,i] = b+1$ if $S[i \dots b]$ contains all zeros.

As before, the structures require $o(n)$ bits. They can answer $\mathit{selectnext}(B,i)$ in $O(1)$ time as follows.

(1) Compute $i_b = (i \text{ div } b) \cdot b$ and then $pos = N_p[B[i_b+1 \dots i_b+b], i-i_b+1]$. If $pos \leq b$, then there is a bit set in $B[i \dots i_b+b-1]$ and we just return $i_b + pos$.
(2) Otherwise, $\mathit{selectnext}(B,i) = \mathit{selectnext}(B, i_b+b)$, so find the answer corresponding to the beginning of the next block. Compute $pos = N_b[(i \text{ div } b)+1]$. If $pos \leq s$, then return $((i_b + b) \text{ div } s) \cdot s + pos$.
(3) Otherwise, there are all zeros in $B[i \dots i_s+s-1]$ where $i_s = (i \text{ div } s) \cdot s$, so $\mathit{selectnext}(B,i) = \mathit{selectnext}(B, i_s+s)$. Return $N_s[i \text{ div } s+1]$.

4 First Huffman, then Burrows-Wheeler

We focus now on our index representation. Imagine that we compress our text $T\$$ using Huffman. The resulting bit stream will be of length $n' < (H_0+1)n$, since (binary) Huffman poses a maximum representation overhead of 1 bit per symbol[5]. Let us call T' this sequence. Let us also define a second bit array Th, of the same length of T', such that $Th[i] = 1$ iff i is the starting position of a Huffman codeword in T'. Th is also of length n'. (We will not, however, represent T' nor Th in our index.)

The idea is to search the binary text T' instead of the original text T. Let us apply the Burrows-Wheeler transform over text T', so as to obtain $B = (T')^{bwt}$. The terminator character, "$", is excluded from T' so as to have a binary alphabet.

More precisely, let $\mathcal{A}'[1 \dots n']$ be the suffix array for text T', that is, a permutation of the set $1 \dots n'$ such that $T'[A'[i] \dots n'] < T'[A'[i+1] \dots n']$ in lexicographic order, for all $1 \leq i < n'$. In a lexicographic comparison, if a string x is a prefix of y, assume $x < y$. Suffix array $\mathcal{A}'$ will not be explicitly represented. Rather, we represent bit array $B[1 \dots n']$, such that $B[i] = T'[\mathcal{A}'[i]-1]$ (except that $B[i] = T[n']$ if $\mathcal{A}'[i] = 1$). We also represent another bit array $Bh[1 \dots n']$, such that $Bh[i] = Th[\mathcal{A}'[i]]$. This tells whether position i in $\mathcal{A}'$ points to the beginning of a codeword.

Our goal is to search B exactly like the FM-index. For this sake we need array C and function Occ. Since the alphabet is binary, however, Occ can be easily computed: $Occ(B,1,i) = rank(B,i)$ and $Occ(B,0,i) = i - rank(B,i)$. Also, array C is so simple for the binary text that we can do without it: $C[0] = 0$ and $C[1] = n' - rank(B,n')$, that is, the number of zeros in B (of course value $n' - rank(B,n')$ should be precomputed in practice). Therefore, $C[c] + Occ(T^{bwt}, c, i)$ is replaced in our index by $i - rank(B,i)$ if $c = 0$ and $n' - rank(B,n') + rank(B,i)$ if $c = 1$.

There is a small twist, however, due to the fact that we are not putting a terminator to our binary sequence T' and hence no terminator appears in B. Let us call "#" the terminator of the binary sequence so that it is not confused with the terminator "$" of $T\$$. In the position $p_\#$ such

[5] Note that these n and H_0 refer to $T\$$, not T. However, the difference between both is only $O(\log n)$, and will be absorbed by the $o(n)$ terms that will appear later.

that $\mathcal{A}'[p_\#] = 1$, we should have $B[p_\#] = \#$. Instead, we set $B[p_\#]$ to the last bit of T'. This is the last bit of the Huffman codeword assigned to the terminator "\$" of $T\$$. Since we can freely switch left and right siblings in the Huffman code, let us assume this last bit will be zero. Hence the correct B sequence will be of length $n'+1$, starting with 0 (which corresponds to $\mathcal{A}[n'+1] = \#$, since $\# < 0 < 1$), and it would have $B[p_\#] = \#$. To obtain the right mapping to our binary B, we must correct $C[0] + Occ(B, 0, i) = i - rank(B, i) + 1 - [i \geq p_\#]$, that is, add 1 to the original value when $i < p_\#$. The computation of $C[1] + Occ(B, 1, i)$ remains unchanged.

Therefore, by preprocessing B to solve $rank$ queries, we can search B exactly as the FM-index. The extra space required by the $rank$ structure is $o(H_0 n)$, without any dependence on the alphabet size. Overall, we have used at most $n(2H_0 + 2)(1 + o(1))$ bits for our representation. This will grow slightly in the next sections due to additional requirements.

Our search pattern is not the original P, but its binary coding P' using the Huffman code we applied to T. Converting P to P' takes $O(m)$ time. If we assume that the characters in P have the same distribution of T, then the length of P' is $< m(H_0 + 1)$. This is the number of steps to search B using the FM-index search algorithm.

The answer to that search, however, is different from that of the search of T for P. The reason is that the search of T' for P' returns the number of suffixes of T' that start with P'. Certainly these include the suffixes of T that start with P, but also other superfluous occurrences may appear. These correspond to suffixes of T' that do not start a Huffman codeword, yet they start with P'.

This is the reason why we have marked the suffixes that start a Huffman codeword in Bh. In the range $[sp, ep]$ found by the search for P' in B, every bit set in $Bh[sp \ldots ep]$ represents a true occurrence. Hence the true number of occurrences can be computed as $rank(Bh, ep) - rank(Bh, sp - 1)$.

Figure 2 depicts the search algorithm. An example is shown in the Appendix.

Algorithm Huff-FM_Search(P',B,Bh)
(1) $i = m'$;
(2) $sp = 1$; $ep = n'$;
(3) **while** $((sp \leq ep)$ **and** $(i \geq 1)$ **do**
(4) **if** $P'[i] = 0$ **then**
$sp = (sp - 1) - rank(B, sp - 1) + 1 + 1 - [sp - 1 \geq p_\#]$;
$ep = ep - rank(B, ep) + 1 - [ep \geq p_\#]$;
else $sp = n' - rank(B, n') + rank(B, sp - 1) + 1$;
$ep = n' - rank(B, n') + rank(B, ep)$;
(7) $i = i - 1$;
(8) **if** $ep < sp$ **then** $occ = 0$ **else** $occ = rank(Bh, ep) - rank(Bh, sp - 1)$;
(9) **if** $occ = 0$ **then return** "not found" **else return** "found (occ) occs".

Fig. 2. Algorithm for counting the number of occurrences of $P'[1 \ldots m']$ in $T'[1 \ldots n']$.

Therefore, the search complexity is $O(m(H_0 + 1))$, assuming that the zero-order distributions of P and T are similar. For the worst case, note that any character that appears in T has frequency at least 1 and hence a probability $p \geq 1/n$. Huffman cannot assign it a codeword longer than

$1 + \log(1/p) \leq 1 + \log n$. Therefore, the encoded pattern P' cannot be longer than $O(m \log n)$ and this is also the worst case search cost. This matches the worst case search cost of the original CSA.

An exception to the above argument occurs when P contains a character not present in T. This is easier, however, as we immediately know that P does not occur in T.

5 Ensuring $O(m \log \sigma)$ Worst Case Search Time

Still, we can limit our worst case to $O(m \log \sigma)$ and retain both our $O(m(H_0+1))$ average complexity and $O(n(H_0 + 1))$ space. This technique has been used in [16] under a different scenario. The idea is to change the Huffman coding for T, while all the rest remains untouched.

Pick a constant t, so that the Huffman tree shape for T is followed up to depth $t \log \sigma$. All the subtrees at that depth are converted to perfectly balanced subtrees. It follows that the depth of the tree is at most $t \log \sigma + \log \sigma = (t + 1) \log \sigma$, and hence the search cost is $O(m \log \sigma)$ in the worst case.

The question is which is the space and the average search time now. Consider first the space, as the average search time will be similar. Take a leaf that in the original Huffman tree has depth $l \geq t \log \sigma$. This means that its empirical probability p must satisfy $\log(1/p) + 1 \geq l \geq t \log \sigma$, and therefore $p \leq 2/\sigma^t$. So the character can appear in the text at most $2n/\sigma^t$ times.

Adding these upper bounds on the probabilities over all the possible characters that might be so deep, we have that the overall number of occurrences of such characters in the text cannot exceed $2n\sigma/\sigma^t = 2n/\sigma^{t-1}$. Those characters can be at depth at most $(t + 1) \log \sigma$, and therefore the total number of bits allocated to them cannot exceed $(2n/\sigma^{t-1})(t + 1) \log \sigma$. On the other hand, all the other characters maintain their original depths, and we pessimistically assume that they add up $n(H_0 + 1)$ bits. Hence the point is to find out how large must t be so that $(2/\sigma^{t-1})(t + 1) \log \sigma = O(H_0 + 1)$. Solving for $(2/\sigma^{t-1})(t + 1) \log \sigma \leq x(H_0 + 1)$, one obtains that for any $x \geq 1/(H_0 + 1)$ it is enough that

$$t \quad \geq \quad 1 + \log_\sigma 2 + \log_\sigma \log \sigma + \log_\sigma(t + 1)$$

for which it is sufficient that

$$t \quad \geq \quad 1 + \log_\sigma 2 + \log_\sigma \log \sigma + t/\ln \sigma$$

that is,

$$t \quad \geq \quad (1 + \log_\sigma 2 + \log_\sigma \log \sigma)\frac{\ln \sigma}{\ln \sigma - 1}$$

which tends to 1 as σ grows. Using this t value, we have that $t \log \sigma = (\log \sigma + \log \log \sigma + 1) \ln \sigma/(\ln \sigma - 1) = \log \sigma(1 + o(1))$.

Overall, the number of bits needed for T' is $n(H_0 + 1 + x(H_0 + 1)) = n(H_0 + 2)$, and thus B and Bh together need $n(2H_0 + 4)(1 + o(1))$ bits, without large constants hidden in the sublinear parts. The space price of ensuring $O(m \log \sigma)$ worst case has been $2n$ extra bits.

The analysis of the modified average search time is similar. We have that there are at most $2m\sigma/\sigma^t$ times where our search cost will be $O(\log \sigma)$, while the others will cost $O(H_0 + 1)$ and we do not need to consider them further. Exactly as for the analysis of the space, we seek a constant t such that $2m\sigma/\sigma^t \leq xm(H_0 + 1)$ and find the same solution as for guaranteeing $O(n(H_0 + 1))$ extra space. This is fortunate, as we must use the same t for both purposes.

6 Reporting Occurrences and Displaying the Text

Up to now we have focused on the search time, that is, the time to determine the suffix array interval containing all the occurrences. In practice, one needs also the text positions where they appear, as well as a text context. Since self-indexes replace the text, in general one needs to extract any text substring from the index.

Given the suffix array interval that contains the *occ* occurrences found, the FM-index reports each such position in $O(\sigma \log^{1+\varepsilon} n)$ time, for any $\varepsilon > 0$ (which appears in the sublinear space component). The CSA can report each in $O(\log^{\varepsilon} n)$ time, where ε is paid in the nH_0/ε space. Similarly, a text substring of length L can be displayed in time $O(\sigma(L+\log^{1+\varepsilon} n))$ by the FM-index and $O(L + \log^{\varepsilon} n)$ by the CSA.

In this section we show that our index can do better than the FM-index, although not as well as the CSA. Using $(1+\varepsilon)n$ additional bits, we can report each occurrence position in $O(\frac{1}{\varepsilon}(H_0+1)\log n)$ time and display a text context in time $O(L \log \sigma + \log n)$ in addition to the time to find an occurrence position. On average, assuming that random text positions are involved, the overall complexity to display a text interval becomes $O((H_0+1)(L+\frac{1}{\varepsilon}\log n))$.

6.1 Reporting Occurrences

A first problem is how to extract, in $O(occ)$ time, the *occ* positions of the bits set in $Bh[sp \ldots ep]$. This is easy using *select*: Let $r = rank(Bh, sp-1)$. Then, the positions of the bits set in Bh are $select(Bh, r+1)$, $select(Bh, r+2)$, ..., $select(Bh, r+occ)$. We recall that $occ = rank(Bh, ep) - rank(Bh, sp-1)$. This can be expressed using our *selectnext* function: The positions $pos_1 \ldots pos_{occ}$ can be found as $pos_1 = selectnext(Bh, sp)$, and $pos_{i+1} = selectnext(Bh, pos_i + 1)$. We focus now on how to find the text position of a valid occurrence.

For the development that follows, we recall from Section 4 that no Huffman codeword can be longer than $1+\log n$ bits: No symbol probabiliy can be smaller than $p = 1/n$ and Huffman codeword lengths are at most $1 + \log(1/p)$.

We choose some $\varepsilon > 0$ and sample $\lfloor \frac{\varepsilon n}{2 \log n} \rfloor$ positions of T' at regular intervals, with the restriction that only codeword beginnings can be chosen. For this sake, pick positions in T' at regular intervals of length $\ell = \lceil \frac{2n'}{\varepsilon n} \log n \rceil$, and for each such position $1+\ell(i-1)$, choose the beginning of the codeword being represented at $1 + \ell(i-1)$.

Since no codeword can be longer than $1 + \log n$, the distance between two chosen positions in T', after the adjustment, cannot exceed

$$\ell + 1 + \log n \quad \leq \quad \frac{2}{\varepsilon}(H_0+1)\log n + 2 + \log n \quad = \quad O\left(\frac{1}{\varepsilon}(H_0+1)\log n\right)$$

Now, store an array TS with the $\lfloor \frac{\varepsilon n}{2 \log n} \rfloor$ positions of $\mathcal{A}'$ pointing to the chosen positions of T', in increasing text position order. More precisely, $TS[i]$ refers to position $1+\ell(i-1)$ in T' and hence $TS[i] = j$ such that $A'[j] = select(Th, rank(Th, 1+\ell(i-1)))$. Array TS requires $\frac{\varepsilon n}{2}(1+o(1))$ bits, since each entry needs $\log n' \leq \log(n \log \min(n, \sigma)) = \log n + O(\log \log \min(n, \sigma))$ bits.

The same $\mathcal{A}'$ positions are now sorted and the corresponding T positions (that is, $rank(Th, \mathcal{A}'[i])$) are stored in array ST, for other $\frac{\varepsilon n}{2}$ bits. Finally, we store an array S of n bits so that $S[i] = 1$ iff

$\mathcal{A}'[select(Bh, i)]$ is in the sampled set. That is, $S[i]$ tells whether the i-th entry of $\mathcal{A}'$ pointing to beginning of codewords, points to a sampled text position. S is further processed for $rank$ queries.

Overall, we spend $(1+\varepsilon)n(1+o(1))$ bits for these three arrays, raising our final space requirement to $n(2H_0 + 3 + \varepsilon)(1 + o(1))$, and $n(2H_0 + 5 + \varepsilon)(1 + o(1))$ to ensure $O(m \log \sigma)$ worst case search time.

Let us focus first in how to determine the text position corresponding to an entry $\mathcal{A}'[i]$ for which $Bh[i] = 1$. Use bit array $S[rank(Bh, i)]$ to determine whether $\mathcal{A}'[i]$ points or not to a codeword beginning in T' that has been sampled. If it does, then find the corresponding T position in $ST[rank(S, rank(B, i))]$ and we are done. Otherwise, just as done by the FM-index, determine position i' whose value is $\mathcal{A}'[i'] = \mathcal{A}'[i] - 1$. Repeat this process, which corresponds to moving backward bit by bit in T', until a new codeword beginning is found, that is, $Bh[i'] = 1$. Now determine again whether i' corresponds to a sampled character in T: Use $S[rank(Bh, i')]$ to determine whether $\mathcal{A}'[i']$ is present in ST. If it is, report text position $1 + ST[rank(S, rank(Bh, i'))]$ and finish. Otherwise, continue with i'' trying to report $2 + ST[rank(S, rank(Bh, i''))]$, and so on. The process must finish after $O\left(\frac{1}{\varepsilon}(H_0 + 1) \log n\right)$ backward steps in T' because we are considering consecutive positions of T' and that is the maximum distance among consecutive samples.

We have to specify how we determine i' from i. In the FM-index, this is done via the LF-mapping, $i' = C[T^{bwt}[i]] + Occ(T^{bwt}, T^{bwt}[i], i)$. In our index, the LF-mapping over $\mathcal{A}'$ is implemented as $i' = i - rank(B, i)$ if $B[i] = 0$ and $i' = n' - rank(B, n') + rank(B, i)$ if $B[i] = 1$. This LF-mapping moves us from position $T'[\mathcal{A}'[i]]$ to $T'[\mathcal{A}'[i] - 1]$.

Overall, an occurrence can be reported in worst case time $O(\frac{1}{\varepsilon}(H_0 + 1) \log n)$. Figure 3 gives the pseudocode. An example is given in the Appendix.

Algorithm Huff-FM_Position(i,B,Bh,ST)
(1) $d = 0$;
(2) **while** $S[rank(Bh, i)] = 0$ **do**
(3) **do if** $B[i] = 0$ **then** $i = i - rank(B, i) + 1 - [i \geq p_{\#}]$;
else $i = n' - rank(B, n') + rank(B, i)$;
(4) **while** $Bh[i] = 0$;
(5) $d = d + 1$;
(6) **return** $d + ST[rank(S, rank(Bh, i))]$;

Fig. 3. Algorithm for reporting the text position of the occurrence at $B[i]$. It is invoked for each $i = select(Bh, r+k)$, $1 \leq k \leq occ$, $r = rank(Bh, sp - 1)$.

6.2 Displaying Text

In order to display a text substring $T[l \ldots r]$ of length $L = r - l + 1$, we start by binary searching TS for the smallest sampled text position larger than r. Given value $TS[j]$, we know that $S[rank(Bh, TS[j])] = 1$ as it is a sampled $\mathcal{A}'$ entry, and the corresponding T position is simply $ST[rank(S, rank(Bh, TS[j]))]$. Once we find the first sampled text position that follows r, we have

its corresponding position $i = TS[j]$ in $\mathcal{A}'$. From there on, we perform at most $O\left(\frac{1}{\varepsilon}(H_0+1)\log n\right)$ steps going backward in T' (via the LF-mapping over $\mathcal{A}'$), position by position, until reaching the first bit of the codeword for $T[r+1]$. Then, we obtain the L preceding positions of T, by further traversing T' backwards, collecting all its bits until reaching the first bit of the codeword for $T[l]$. The reversed bit stream collected is Huffman-decoded to obtain $T[l \ldots r]$.

Each of those L characters costs us $O(H_0+1)$ on average because we obtain the codeword bits one by one. In the worst case they costs us $O(\log n)$, or $O(\log \sigma)$ by using the techniques of Section 5. The overall time complexity is $O((H_0+1)(L+\frac{1}{\varepsilon})\log n)$ on average and $O(L\log\sigma + (H_0+1)\frac{1}{\varepsilon}\log n)$ in the worst case.

Figure 4 shows the pseudocode. An example is given in the Appendix.

Algorithm Huff-FM_Display(l,r,B,Bh,TS)
(1) $j = \min\{k,\ ST[rank(S, rank(Bh, TS[k]))] > r\}$; // binary search
(2) $i = TS[j]$;
(3) $p = ST[rank(S, rank(Bh, i))]$;
(4) $L = \langle\ \rangle$;
(5) **while** $p \geq l$ **do**
(6) **do if** $B[i] = 0$ **t**hen $i = i - rank(B, i) + 1 - [i \geq p_{\#}]$;
else $i = n' - rank(B, n') + rank(B, i)$;
(7) $L = B[i] \cdot L$;
(8) **while** $Bh[i] = 0$;
(9) $p = p - 1$;
(10) Huffman-decode the first $r - l + 1$ characters from list L;

Fig. 4. Algorithm for extracting $T[l \ldots r]$.

7 Implementation and Experiments

We are currently implementing the index. The main part that supports counting queries (Section 4) is ready. In this section we report experimental results on counting queries and compare the efficiency to existing indexes. The implementation is available at
`http://www.cs.helsinki.fi/u/vmakinen/software/`.

Our experiments were run over 15 MB of text obtained from the "ZIFF-2" disk of the TREC-3 collection [8]. The tests ran on a Pentium IV processor at 2.6 GHz, 2 GB of RAM and 512 KB cache, running Red Hat Linux 3.2.2-5. We compiled the code with `gcc 3.2.2` using optimization option `-O3`. Times were summed over 10,000 search patterns. As we work only in main memory, we only consider CPU times. The search patterns were obtained by pruning text lines to their first m characters. We avoided lines containing tags and non-visible characters such as '&'.

Our index took 1.84 times the text size[6]. Indexes of similar size are the LZ-index variant of Navarro [17] (1.42 times the text size), and the Compressed Compact Suffix Array (CCSA) of

[6] This does not include the structures for reporting queries, which will add at least 0.13 to the 1.84. The other spaces reported in this paragraph do include structures for reporting.

Mäkinen and Navarro [15] (1.64 times the text size). The former is quite slow in counting queries [17], so we did not include it in this experiment. We used the implementation of the CCSA that can be found at the Web address given above. The Compressed Suffix Array (CSA) of Sadakane [18] has similar asymptotic space requirement as our index, but it is much smaller in practice (0.71 times the text size). The FM-index of Ferragina and Manzini [3, 4] is even smaller. Navarro has implemented a variant of the FM-index that takes more space (1.33 times the text size in this experiment), but searches faster than the original implementation. We used Sadakane's CSA implementation and the FM-index implementation of Navarro. Both can be downloaded from `http://www.dcc.uchile.cl/~gnavarro/software`.

Figure 5 shows the result for counting queries. As expected, our index is faster than CSA and CCSA, whose running times are $O(m \log n)$. Also, as expected, our index is slower than FM-index, whose running time is $O(m)$.

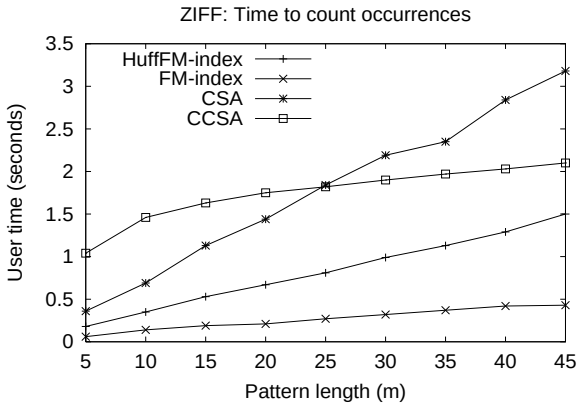


Fig. 5. Query times for our index (HuffFM-index) versus alternative succinct indexes. We report overall time for 10,000 queries.

Now, let us speculate about reporting queries. As confirmed by previous experiments [17, 15], the FM-index is the slowest in reporting queries compared to CSA, CCSA, and LZ-index. In theory, our index is faster than FM-index in reporting and somewhat slower than CSA and CCSA. If we can keep the same lead over CSA and CCSA as in counting queries, then our index will become an attractive tradeoff alternative. This will be the hypothesis to be tested once we have the full implementation.

8 Conclusions

We have focused in this paper on a practical data structure inspired by the FM-index [3], which removes its sharp dependence on the alphabet size σ. Our key idea is to Huffman-compress the text before applying the Burrows-Wheeler transform over it. Over a text of n characters, our structure needs $O(n(H_0+1))$ bits, being H_0 the zero-order entropy of the text. It can search for a pattern of length m in $O(m(H_0+1))$ average time. Our structure has the advantage over the FM-index of not

depending at all on the alphabet size, and of having better complexities to report text occurrences and displaying text substrings. It has the advantage over the CSA [18] of having better search time. Also, compared to the other structures, ours is simpler and easier to implement.

One possibility to reduce the $O(m(H_0+1))$ search cost is to use Huffman over a coding alphabet of $k > 1$ bits instead of binary Huffman ($k = 1$). In this case the number of bits in B is bounded by $n' < (H_0 + k)n$, and B_h needs only n'/k bits. Furthermore, the search pattern is of length $< m(H_0 + k)$ and the LF-mapping can be carried out in chunks of k bits, for a search complexity of $O(m(H_0/k + 1))$. The main problem, however, is that we cannot anymore use $rank$ over B to simulate Occ. Rather, we need one bit stream of length n'/k for each of the 2^k different coding symbols. More precisely, we need their sublinear structures to compute $rank$ and $selectnext$. The bit-streams themselves can be replaced by the single B for the whole text, provided we precompute the 2^k tables that count the number of occurrences of each symbol in a chunk of b bits from B.

Under this scheme, the space complexity becomes $n(1+1/k)(H_0+k)(1+o(2^k/k))$ and the search time becomes $O(m(H_0/k + 1))$. The complexities to report occurrences and display text positions get divided by k too. Hence parameter $k \in 1 \ldots \log \sigma$ yields different space/time tradeoffs.

References

1. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *DEC SRC Research Report 124*, 1994.
2. D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.
3. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pp. 390–398, 2000.
4. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. SODA'01*, pp. 269–278, 2001.
5. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pp. 841–850, 2003.
6. R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. SODA'04*, 2004.
7. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. STOC'00*, pp. 397–406, 2000.
8. D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. TREC-3*, pages 1–19, 1995. NIST Special Publication 500-207.
9. G. Jacobson. *Succinct Static Data Structures*. PhD thesis, CMU-CS-89-112, Carnegie Mellon University, 1989.
10. J. Kärkkäinen. *Repetition-Based Text Indexes*, PhD Thesis, Report A-1999-4, Department of Computer Science, University of Helsinki, Finland, 1999.
11. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22, pp. 935–948, 1993.
12. E. M. McCreight. A space economical suffix tree construction algorithm. *J. of the ACM*, 23, pp. 262–272, 1976.
13. I. Munro. Tables. In *Proc. FSTTCS'96*, pp. 37–42, 1996.
14. V. Mäkinen. Compact Suffix Array — A space-efficient full-text index. *Fundamenta Informaticae* 56(1-2), pp. 191–210, 2003.
15. V. Mäkinen and G. Navarro. Compressed compact suffix arrays. To appear in *Proc. CPM'04*, LNCS, 2004.
16. V. Mäkinen and G. Navarro. On succinct suffix arrays. Submitted, 2004.
17. G. Navarro. Indexing text using the Ziv-Lempel trie. *J. of Discrete Algorithms* 2(1):87–114, 2004.
18. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. ISAAC'00*, LNCS 1969, pp. 410–421, 2000.
19. S. Srinivasa Rao. Time-space trade-offs for compressed suffix arrays. *Inf. Proc. Lett.*, 82 (6), pp. 307-311, 2002.
20. E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14, pp. 249–260, 1995.
21. P. Weiner. Linear pattern matching algorithms. In *Proc. IEEE 14th Ann. Symp. on Switching and Automata Theory*, pp. 1–11, 1973.

Appendix — To be read at the discretion of the reviewer

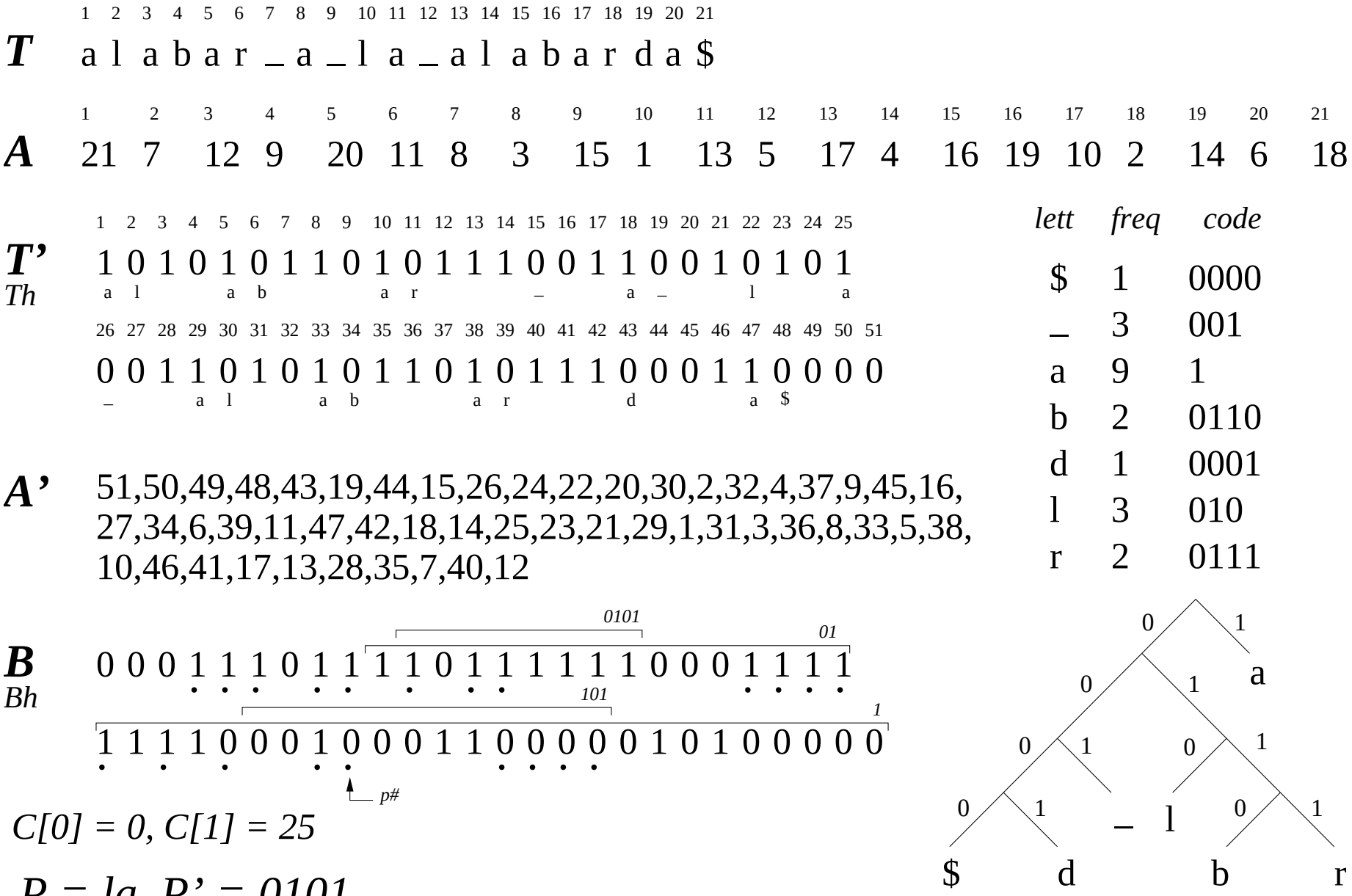
The purpose of this appendix is to illustrate the index structure and workings using a small example.

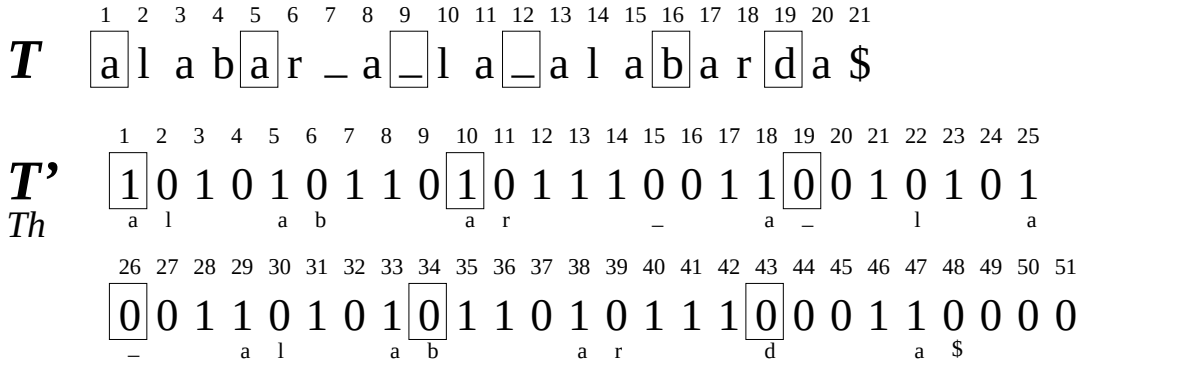
Figure 6 shows an example over the text *alabar_a_la_alabarda*. We display T and $\mathcal{A}$. On the right we show the Huffman codebook for T. Using this Huffman code, the resulting text T' is shown. We also display Th in the form of the corresponding letters in small font below the T' positions i where $Th[i] = 1$. We then show $\mathcal{A}'$. Finally, we show B and Bh, which are the only structures really stored. Bh is shown in the form of dots below the positions i of B such that $Bh[i] = 1$. Position $p_{\#}$ is also indicated. Finally, we show the search process for $P = la$, which translates into $P' = 0101$, indicating the ranges in B corresponding to 1, 01, 101, and finally 0101. As one can see, there are three dots in Bh in the resulting range $[sp, ep] = [11, 18]$, corresponding to the positions in T that match $P = la$. All the others are false matches of 0101 in T'.

Figure 7 continues our example, considering reporting occurrences. We have used a sampling step of $\ell = 9$ over T', choosing positions 1, 10, 19, 28, 37, and 46. Those positions are adjusted to their previous codeword beginning, obtaining 1, 10, 19, 26, 34, and 43. These are marked with a square in T', as well as their corresponding positions in T. The positions of $\mathcal{A}'$ pointing to sampled T' positions have been collected, in T' order, in array TS. For clarity, we have squared in B those sampled $\mathcal{A}'$ positions. The same $\mathcal{A}'$ positions have been sorted by their value, and the corresponding sampled T entries have been stored in array ST. Array S marks, for each position i such that $Bh[i]$, whether it is sampled or not. Then we illustrate the quest for the text position of the first occurrence of $P = la$, that is, $\mathcal{A}'[11]$, where $11 = selectnext(Bh, sp)$. Since $S[rank(Bh, 11)] = S[6] = 0$, position 11 is not sampled and we must traverse T' backwards in our quest for a sampled position. (Note that $\mathcal{A}'[11] = 22$ but we do not know that.) The LF-mapping over B leads us to position 32, since $\mathcal{A}'[32] = 21$. Since $Bh[32] = 0$, we have not yet reached the previous Huffman codeword in T'. We apply the LF-mapping again and are leaded to position 12, $\mathcal{A}'[12] = 20$ but still $Bh[12] = 0$. Finally, one further application of the LF-mapping leads us to position 6, $\mathcal{A}'[6] = 19$. Here we have $Bh[6] = 1$, so we have reached the previous Huffman codeword and increment d, $d = 1$. It also turns out that $S[rank(Bh, 6)] = S[3] = 1$, which means that the T position corresponding to $\mathcal{A}'[6]$ is sampled in $ST[rank(S, 3)] = ST[2] = 9$. Adding $d = 1$ we obtain the correct T position 10, where an occurrence of $P = la$ begins.

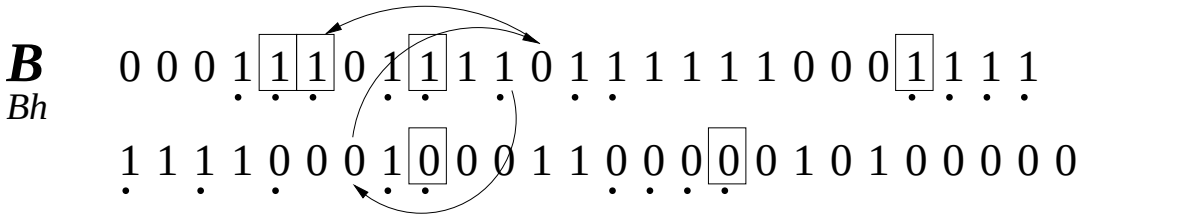
Finally, as an example of displaying text substrings, note in Figure 7 that the B values we have traversed, excluding the last $B[6]$, form the string $B[11]B[32]B[12] = 100$. Reversed, this is 001, the code for character "_" in $T[9]$. Say that we wish to display a context around our occurrence at $T[10]$, say $T[8 \ldots 13]$. Then we binary search TS to find that $TS[5] = 22$ corresponds to text position $ST[rank(S, rank(Bh, TS[5]))] = ST[rank(S, rank(Bh, 22))] = ST[rank(S, 9)] = ST[4] = 16$, which is the smallest one following 13. Now we apply the LF-mapping from $B[22]$, collecting all the Huffman codewords that give us, successively, $T[15]$, $T[14]$, ..., $T[8]$.

Fig. 6. Example of our structure for text $T = alabar\_a\_la\_alabarda$ and pattern $P = la$.





A' 51,50,49,48,43,19,44,15,26,24,22,20,30,2,32,4,37,9,45,16,
27,34,6,39,11,47,42,18,14,25,23,21,29,1,31,3,36,8,33,5,38,
10,46,41,17,13,28,35,7,40,12



TS 34,42,6,9,22,5

ST 19,9,12,16,1,5

S 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1

Fig. 7. Finding the text position of occurrence $B[11]$ in our running example.