UNIVERSITY OF JOENSUU

DEPARTMENT OF COMPUTER SCIENCE AND STATISTICS

Report Series A

Succinct pattern matching automata

Kimmo Fredriksson

A-2007-1

ACM E.1, E.2, E.4, F.2, H.3.1, H.3.3

ISBN 978-952-458-959-8

ISSN 1796-7317

May 2, 2007

Succinct pattern matching automata

Kimmo Fredriksson*
Department of Computer Science and Statistics
University of Joensuu
kfredrik@cs.joensuu.fi

Abstract

We consider single and multiple string matching in small space and optimal average time. Our algorithm is based on the combination of compressed self-indexes and Backward-DAWG-Matching algorithm. Efficient implementation techniques are discussed.

Keywords: algorithms, string matching, text indexing, compression, succinct data structures

ACM Classification: E.1 [Data structures]; E.2 [Data storage representations]; E.4 [Coding and information theory] — *Data compaction and compression*; F.2 [Analysis of algorithms and problem complexity]; H.3.1 [Information storage and retrieval]: Content Analysis and Indexing — *Dictionaries, Indexing methods* H.3.3 [Information storage and retrieval]: Information Search and Retrieval — *Search process*

1 Introduction

We address the well known exact string matching problem. The problem is to search the occurrences of the pattern $P[1 \dots m]$ from the text $T[1 \dots n]$, where the symbols of P and T are taken from some finite alphabet Σ, of size σ. Numerous efficient algorithms solving the problem have been obtained. The first linear time algorithm (KMP) was given in [19], and the first sublinear average time algorithm (BM) in [3]. Many practical variants of BM family have been suggested, see e.g. [15, 30]. An average optimal $O(n \log_\sigma(m)/m)$ time algorithm (BDM, see also Sec. 4) is obtained e.g. in [8], which also generalizes for searching r patterns simultaneously in average time $O(n \log_\sigma(rm)/m)$, which is again optimal [22]. Aho-Corasick algorithm [1] solves the same problem in $O(n)$ worst case search time. Recently bit-parallelism has been shown to lead to the most efficient algorithms for relatively short patterns, in practice. The first algorithm in this class was Shift-Or [2, 32], which runs in time $O(n\lceil m/w \rceil)$ time, where w is the number of bits in computer word (typically 32 or 64). For $m = O(w)$ this is a linear time algorithm. Among the best algorithms is BNDM [24], which is a bit-parallel version of BDM. This achieves $O(n\lceil m/w \rceil \log_\sigma(m)/m)$ average time, which is optimal for short patterns. These algorithms can be also generalized for multiple patterns, but either the algorithms become filters working well only for small or moderate r, or the $\lceil m/w \rceil$ term usually becomes $\lceil (rm)/w \rceil$, which makes them impractical for large sets of long patterns.

All the above algorithms need to preprocess the pattern and require extra space for data structures. Usually the extra space is $O(rm)$ or $O(rm\sigma)$ words, depending on the implementation details, or $O(\sigma\lceil m/w \rceil)$ for the bit-parallel algorithms. Usually eliminating the σ factor means $O(\log \min(m, \sigma))$ additional factor in the search time. Another line of work is constant additional space algorithms [12, 7, 5, 9]. These obtain $O(n)$ worst case time for single pattern while using only $O(1)$ words of *additional* space for the data structures, but also need the original pattern as well, i.e. at least $O(m \log(\sigma))$ bits.

*Supported by the Academy of Finland, grant 207022.

For more references of exact string matching algorithm see e.g. [25, 8, 10].

Another approach for string matching is *indexing* (also called off-line searching). In this case one is allowed to preprocess the text, so that given a pattern all its occurrences can be counted in close to $O(m)$ time, and located in close to $O(m + occ)$ time, where occ is the number of occurrences. One such (classical) data structure is suffix tree [31], which informally is a path compressed trie of all the text suffixes. This needs $O(n)$ (or $O(n\sigma)$) words of space (i.e. at least $O(n \log(n))$ bits), but the hidden constant can be very large. The text itself needs to be stored as well, taking another $O(n \log(\sigma))$ bits. Suffix array [21] is basically in its simplest form a sorted array of (pointers/indexes to) the text suffixes, so that the pattern query becomes a binary search in that array. The constant factor is much smaller than for suffix trees, but for huge text collections can be still too large. Recent trend in text indexing is succinct or compressed indexes (also called self-indexes if the original text is not needed) [11, 23]. These methods achieve space close to the information theoretical lower bound. For more details see [23] and Sec. 3.

Indexing methods obviously are more attractive as compared to on-line searching. However, an index is not always available, and in some cases it is not even plausible to build one, e.g. the text might be inherently on-line, such as in intrusion detection applications. Still the number of patterns to be searched can be huge, as in e.g. anti-virus scanners. In this paper we propose a method that combines BDM (for single or multiple patterns) with compressed self-indexes, resulting in on-line string matching algorithm that has optimal average case search time and can operate in small space. The small space complexity is important in modern computers that have high cache miss costs. It has been experimentally shown that e.g. AC algorithm, having $O(n)$ worst case time for searching r patterns, has superlinear running time of the form $O(nf(r))$ in practice [29]. This is attributed to the high memory requirements. Our algorithms have space complexity close to the information theoretic lower bound. This is better than the complexity for the "constant space" algorithms, which need also the original pattern. Moreover, our algorithms work for multiple patterns as well.

The paper is organized as follows. Sec. 2 gives the basic defintions. Sec. 3 reviews the indexing technique we are going to use in Sec. 4, which describes BDM and how it can use the compressed indexes. Sec. 5 describes a practical implementation issues. Conclusions are given in Sec. 6.

2 Preliminaries

Let the text $T[1 \dots n]$ and the pattern $P[1 \dots m]$ be strings over a finate ordered alphabet $\Sigma = \{0, \dots, \sigma - 1\}$. The *exact string matching* problem is to find all occurrences of P in T. The pattern P occurrs at position i of T, if $P[j] = T[i + j - 1]$ for $1 \leq j \leq m$.

String $S[1 \dots i]$ is a *prefix* of S, string $S[i \dots n]$ is a *suffix* of S, and $S[i \dots j]$ is a *substring* (*factor*) of S. Any of these can be also an empty string.

The function $\texttt{rank}_c(S, i)$ for a sequence (string) $S[1 \dots n]$ gives the number of occurrences of character $c \in \Sigma$ in the prefix $S[1 \dots i]$. A special case arises when $\sigma = 2$ (binary alphabet). In this case $\texttt{rank}_0(S, i) = n - \texttt{rank}_1(S, i)$.

The zeroth-order empirical entropy of the string S is defined to be

$$H_0(S) = -\sum_{s \in \Sigma} \frac{f(s)}{n} \log_2 \left(\frac{f(s)}{n} \right), \tag{1}$$

where $f(s)$ denotes the number of times s appears in S. The k-th order empirical entropy is

$$H_k(S) = -\sum_{i=0}^{n-1} p_i \log_2(p_i), \tag{2}$$

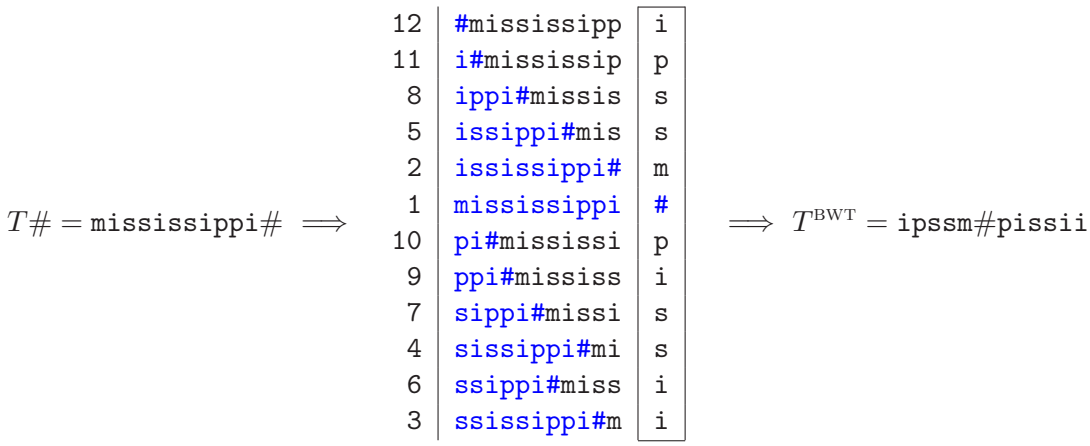

Figure 1: Burrows-Wheeler transforming the string T. Left: the original string; middle: the matrix $\mathcal{M}$, left column shows the corresponding suffix array; right: T^{BWT} (the last column of $\mathcal{M}$).

where $p_i = \text{Probability}(s_i \mid s_{i-k}, \ldots, s_{i-1})$. In other words, the symbol probabilities depend on the context they appear on, i.e. on which are the previous k symbols in S. Obviously, $H_k(S) \leq H_0(S)$.

3 Compressed self-indexing

A full-text index is a data structure that can be used to find all occurrences of a given pattern P from the (indexed) text T efficiently, i.e. without having to scan the text T itself. Classical indexes are e.g. suffix tree [31] and suffix array [21]. Both of these data structures require $O(n \log n)$ bits of space, and especially for suffix tree the constant factor can be very large. The large practical space requirements make these impractical for large text collections. However, in this work we are more interested in good cache performance for relatively very small inputs. One solution for the space problem was proposed by Ferragina and Manzini [11]. Their index (called FM-index by its authors) and its many variations [23] have three main traits: (*i*) the space complexity of the index is proportional to $nH_k(T)$ bits (the k-order empirical entropy of the text), plus some low-order terms depending on the variant; (*ii*) the index can be used to retrieve any substring of the original text, i.e. the index can totally replace the text, hence FM-index is often called *self-index*; (*iii*) the usual search operations can still be performed in close to optimal time.

We now briefly review the FM-index, covering mainly the aspects that are needed in the present work. The index is based on Burrows-Wheeler transformation [4] of the original text, denoted as $T^{\text{BWT}} = \text{BWT}(T)$. Let # denote a special symbol that is lexicographically smaller than any other symbol in the alphabet. Then T^{BWT} is obtained as follows (see also Fig. 1):

1. Generate all cyclic shifts of the string $T\#$.
2. Sort the generated strings into lexicographical order.
3. Assume that the sorted strings form the rows of a matrix $\mathcal{M}$.
4. T^{BWT} is the last column of $\mathcal{M}$.

Note that the cyclic shifts and the matrix $\mathcal{M}$ need not to be explicitly generated, they are used just for the presentation.

Observe that the matrix $\mathcal{M}$ is effectively the suffix array for the text T, i.e. the rows of $\mathcal{M}$ contain all suffixes of T in lexicographical order, and hence any substring S can be searched from

Alg. 1 $\texttt{Count}(P, m)$.

```
1   i ← m; s ← 1; e ← n
2   while s ≤ e AND i > 0 do
3       c ← P[i]
4       s ← C(c) + Occ(c, s − 1) + 1
5       e ← C(c) + Occ(c, e)
6       i ← i − 1
7   return e − s + 1
```

T by searching S with binary search from the rows of $\mathcal{M}$. More precicely, one can use two binary searches to find the interval $[s, e]$ such that the strings in rows $\mathcal{M}_{s...e}$ contain S as a prefix. The most remarkable aspect of BWT is that it is reversible [4], and hence also the matrix $\mathcal{M}$ can be obtained from T^{BWT} only. The novelty of FM-index is to use only T^{BWT} (in compressed form) to simulate the binary search in $\mathcal{M}$ without explicitly constructing it. The key is a so called LF-mapping (Last-to-First), that is, given a position i in the last column $(L, L = T^{\text{BWT}})$ of $\mathcal{M}$, $LF(i)$ gives the position of the corresponding character in the first column (F) of $\mathcal{M}$ (note that each column of $\mathcal{M}$ is a permutation of the string $T\#$). To describe the mapping we need the following definitions:

- $\texttt{C}(c) = \texttt{C}(T, c)$ gives the total number of characters in T that are lexicographically smaller than c.
- $\texttt{Occ}(c, i) = \texttt{rank}_c(T^{\text{BWT}}, i)$ gives the number of occurrences of character c in $T^{\text{BWT}}[1 \dots i]$.

It can then be shown that $LF(i) = \texttt{C}(T^{\text{BWT}}[i]) + \texttt{Occ}(T^{\text{BWT}}[i], i)$. In particular, LF-mapping can be used to scan the original text T backwards, using the transformed text T^{BWT}. If $T[j] = T^{\text{BWT}}[i]$, then $T[j-1] = T^{\text{BWT}}[LF(i)]$. For more details and correctness, see [11, 23].

Using the LF-mapping, Alg. 1 can be used to count the number of occurrences a pattern $P[1 \dots m]$ has in the text T. After each step of the algorithm, the strings in (conceptual) rows $\mathcal{M}_{s...e}$ have $P[i \dots m]$ as a prefix, and hence $e - s + 1$ is the number of occurrences of $P[i \dots m]$ [11]. Note that the pattern must be searched backwards due to the nature of LF-mapping. Other types of queries are of interest in many applications, such as locating the text position for each occurrence, but we only need counting query.

Alg. 1 will be the basic building block in our subsequent algorithms. Its running time depends basically on the efficiency of $\texttt{Occ}()$. If $\texttt{Occ}()$ (implemented as $\texttt{rank}()$) takes constant time, then Alg. 1 takes $O(m)$ worst case time, which is optimal. Note that in principle we do not need T^{BWT} in any explicit form, it is enough that $\texttt{Occ}()$ can be computed *efficiently*, using *as little space as possible*. This has been the central research issue in FM-indexing. See Sec. 5 for discussion on efficient $\texttt{rank}$ implementation.

4 Compressed self-index based BDM automaton

Backward DAWG (Directed Asyclic Word Graph) Matching algorithm (BDM for short) [6] is an average optimal on-line string matching algorithm. The algorithm needs a method to recognize all factors of the pattern. More precicely, suffixes of the *reverse* pattern, i.e. prefixes of the original pattern are enough. To describe the basics of the algorithm we just assume that we have an finite state automaton that recognizes the suffixes (and factors) of the input pattern. We note that such a suffix automaton can be built in $O(m)$ time, for details see [8].

More precicely, we take the pattern in reverse, i.e. $P^r = p_m p_{m-1} \dots p_1$, and build an automaton that recognizes every suffix (including an empty string) of P^r. We now show how this can be used for efficient search. Assume that we are scanning the text window $T[i \dots i + m - 1]$

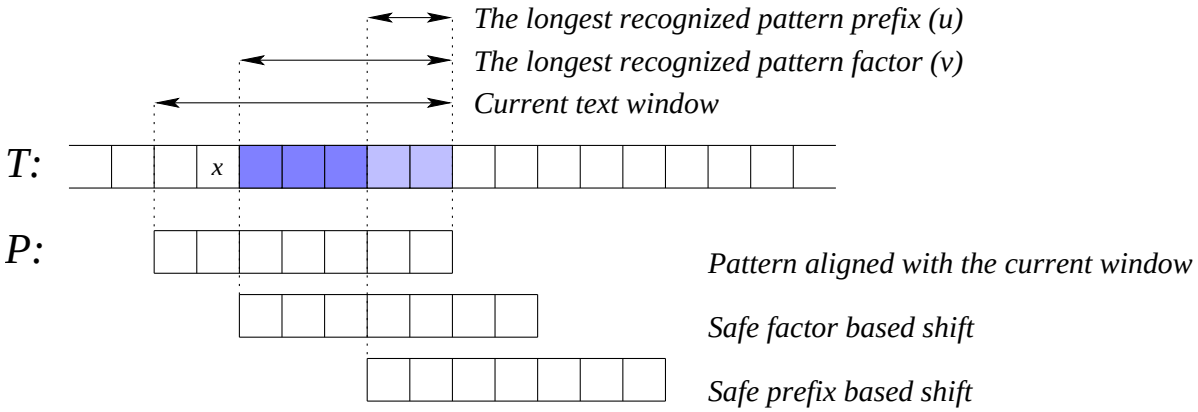

Figure 2: Pattern aligned against text window of length m. The text window is matched against the pattern factors until a mismatch ('x'). The longest recognized factor is v, but xv is not a factor, and hence P can be shifted past x. The longest suffix of v that is also a pattern prefix is u, so P can be shifted to align u with the pattern prefix.

backwards. The invariant is that all occurrences that start before the position i are already reported. The read substring of this window is matched against the automaton. This is continued as long as the substring can be extended without a mismatch, or we reach the beginning of the window. If the whole window can be matched against the automaton, then the pattern occurs in that window. Whether the pattern matches or not, some of the occurrences may still overlap with the current window. However, in this case one of the suffixes stored into the automaton must match, since the reverse suffixes are also the prefixes of the original pattern. The algorithm remembers the longest such suffix, that is not the whole pattern, found from the window. The window is then shifted so that its starting position will become aligned with the last symbol of that suffix. This is the position of the next possible pattern occurrence. If the length of that longest suffix was ℓ, the next window to be searched is $T[i+m-\ell \ldots i+m-1+m-\ell]$. The shifting technique is exactly the same independent of whether or not the pattern occurs in the current window. This process is repeated until the whole text is scanned. Fig. 2 illustrates.

The algorithm runs in $O(n \log_\sigma(m)/m)$ average time, which is optimal [33]. However, the worst case time is $O(nm)$, but it is possible to improve it to $O(n)$.

4.1 Using self-index

We now propose several variants of BDM using self-indexing methods to implement the factor recognition.

The simplest approach is to blindly try to mimick the working of BDM algorithm. Again assume that we are working with the reverse pattern P^r. The idea is to build self-index for P^r and then use Alg. 1 to recognize the factors. The problem with this approach is that the text window should now be read *forwards*, because the pattern was stored backwards. But we cannot afford scanning the text window from the beginning, as this would not allow computing any shift value. Instead, we can take the following approach. Assume that the current window is $T[i \ldots i+m-1]$. On average, the longest matching factor in window is of length $\ell = \Theta(\log(m))$. Therefore the algorithm can read the text substring $t = T[i+m-1-\ell \ldots i+m-1]$ forwards, using algorithm similar to Alg. 1. If no matches are found, then t is not a factor of P, and the window can be safely shifted to $T[i+m-\ell \ldots i+m-\ell+m]$. Otherwise, the window is verified, and the window is shifted only by one character. It is easy to show that the average shift remains $\Theta(m)$, and hence this algorithm is still average optimal, i.e. the average running time is $O(n \log(m)/m)$, but the constant factor is somewhat larger than in BDM, since the shift is based on factors, and not (reverse) suffixes.

Alg. 2 `SBDM`(T, n, P, m).

```
1   P^BWT ← BWT(P)
2   p ← k | P^BWT[k] = #
3   i ← 1
4   while i < n − m + 1 do
5       j ← m; shift ← m; s ← 1; e ← m + 1
6       while s ≤ e AND j > 0 do
7           c ← T[i + j − 1]
8           s ← C(P, c) + rank_c(P^BWT, s − 1) + 1
9           e ← C(P, c) + rank_c(P^BWT, e)
10          j ← j − 1
11          if s ≤ p ≤ e then
12              if j > 0 then shift ← j else report match
13      i ← i + shift
```

Better approach is to use P instead of P^r, since this allows to scan the text backwards, as in the original BDM. First note that building self-index for P means that the index contains the suffixes of P, while BDM is based on the fact the index contains the suffixes of P^r, i.e. the prefixes of P. We now show that the algorithm can still work, even with this change. The main observation is that if we want to recognize only the factors of P, it does not matter whether the index is based on the suffixes of P or P^r. In other words, we can use Alg. 1 to scan the text window backwards, precicely as in plain BDM. To see this, first observe that the whole pattern is obviously recognized, as it is one of the rows of $\mathcal{M}$, and the text window is read backwards. But if the current range $\mathcal{M}_{s\ldots e}$ is not empty, the current substring t of the current window must match a prefix of one of the suffixes of P, and hence it is a factor of P. That is, the scanning is done until (i) we reach the beginning of the window, in which case we have found an occurrence of the pattern; or (ii), the range $s \ldots e$ becomes empty. In the case (i) we simply shift the window by one character position. In the case (ii), we shift the window past the position that caused the range become empty. This algorithm is similar to our first attempt, and is still factor based, since it does not recognize the reverse pattern suffixes. The only difference is that we do not have to use a fixed length text substrings, and hence should be in practice somewhat better.

Finally, the self-index allows an easy method to recognize if any of the recognized factors of P is also a prefix of P, corresponding to reverse suffix of P^r, and hence the original BDM can be simulated exactly. The key observation is that if for the current range $s \ldots e$ any of the characters $P^{\text{BWT}}[s \ldots e]$ include the special symbol #, i.e. the last symbol of P, then the range includes (a prefix of) P. We therefore store the index p of the symbol # in P^{BWT}, and after each step of the backward scanning we check if the range $s \ldots e$ is non-empty, and if p is in that range. If so, the current substring of the text window matches a prefix of the pattern. This text position is recorded, and is used to shift the window precicley as in the original BDM algorithm. Alg. 2 shows the complete pseudo code. As the algorithm exactly simulates each step of BDM in $O(1)$ time (assuming `rank`() takes $O(1)$ time), its average case running time is the optimal $O(n \log(m)/m)$.

4.2 Multiple patterns

Alg. 2 can be easily generalized for matching r patterns simultaneously. Basically the generalization is as in the case of plain BDM algorithm. For simplicity and w.l.o.g., assume that all patterns are of the same length. A simple solution is then to just concatenate all the patterns, appending a special symbol after each pattern, and then Burrows-Wheeler transfom the resulting pattern, and using it as input for Alg. 2. The window length is still obviously just m characters, corresponding to the length of the original patterns. Note that we must also modify

Alg. 3 $\texttt{MSBDM}(T, n, \mathcal{P}, r, m)$.

```
1   P ← P_1 · # · P_2 · # · ... · P_r · #
2   P^BWT ← BWT(P)
3   i ← 1
4   while i < n − m + 1 do
5       j ← m; shift ← m; s ← 1; e ← r(m + 1)
6       while s ≤ e AND j > 0 do
7           c ← T[i + j − 1]
8           s ← C(P, c) + rank_c(P^BWT, s − 1) + r
9           e ← C(P, c) + rank_c(P^BWT, e) + r − 1
10          j ← j − 1
11          if rank_#(P^BWT, e) − rank_#(P^BWT, s − 1) > 0 then
12              if j > 0 then shift ← j else report match
13      i ← i + shift
```

lines 8 and 9 so that we add r to s (instead of 1) and $r-1$ to e (instead of 0), to skip over the additional special symbols. It should be clear that this approach works correctly.

However, there is one non-trivial problem that we must solve. That is, the simple method we used to detect if the range included a pattern prefix is now more complicated. Straightforward generalization of the method would need $O(r)$ time per read text character, which is not acceptable. For each read text character the algorithm needs to detect if the range $s \ldots e$ includes any of the r pattern prefixes. Assume that we have a bitvector B of length $r(m+1)$, so that $B[k] = 1$ iff $P^{\text{BWT}}[k]$ is a special symbol, where P^{BWT} is the Burrows-Wheeler transformed string of the concatenated patterns. Our problem can then be stated as a one-dimensional range query in B; if $B[s \ldots e]$ includes at least one 1-bit, then in the range $s \ldots e$ there is a prefix of at least one of the patterns. This query can be answered in constant time if we have rank data structures built on B: iff

$$\texttt{rank}_1(B, e) - \texttt{rank}_1(B, s-1) > 0$$

then there is at least one pattern prefix in the range. Note that B is in fact not needed, as we can use P^{BWT} directly, i.e. the above query can be stated as

$$\texttt{rank}_\#(P^{\text{BWT}}, e) - \texttt{rank}_\#(P^{\text{BWT}}, s-1) > 0$$

However, an easy "solution" is to just ignore whether or not some pattern prefix is in the range. This turns the method factor based, and the shifts become somewhat shorter (but not asymptotically), but in practice the simpler algorithm may be faster. The average running time becomes $O(n \log(rm)/m)$, as with the generalized BDM [8]. Alg. 3 shows the pseudo code.

5 Rank implementation

The main motivation of using self-indexing methods to implement BDM is to make it memory efficient, in particular cache friendly, which can have great speed impact in practice on modern hardware.

The function $\texttt{C}()$ can be easily implemented with $O(\sigma)$ words of space by using a simple array. The main problem then is to implement $\texttt{rank}_c()$ (a.k.a. $\texttt{Occ}()$) so that it is both fast and uses little space. The simple array based solution would need $O(\sigma m)$ (or $O(\sigma r m)$) words of space, which is the same as for plain BDM if implemented naïvely. Much work – both theoretical and practical – has been done for efficient representation of `rank` structures, see e.g. [17, 28, 14, 18, 27, 13, 20, 26].

We do not go into the details of the various `rank` solutions. We just note some of the basic results. For binary sequence $S[1 \ldots n]$, `rank` can be solved in $O(1)$ time using $n + o(n)$ bits of

space, i.e. the sequence itself, plus $o(n)$ bits for additional directiories and look-up tables. More complicated solutions exists, and one can achieve $nH_0(S) + o(n)$ total space as well. For larger alphabets the simplest solution would be to use σ bit-vectors $B_{0\ldots\sigma-1}$ where the ith bit of B_c is 1 iff $S[i] = c$, and use the binary rank solutions. Wavelet tree [14] is a more elegant solution that use only $nH_0(S) + o(n \log(\sigma))$ bits of space, but $\texttt{rank}_c$ takes $O(\log(\sigma))$ time.

However, in our case we do not necessarily need the most succinct possible solution, since the data structures surely fit into the main memory. However, we would like that the rank structures fit into the CPU cache. Still, the query time is very important in our case. One should also keep in mind that the $O(1)$ time solutions can have large constants in practice, and for large n the cache effects can play important role [27].

In the following, we assume that some reasonably succinct known `rank` solution is used.

5.1 Practical implementation

The practical performance of Alg. 2 and Alg. 3 depends on the implementation of the inner loop. In particular, the performance depends on the `rank` implementation. Since `rank` is hard to implement so that it is both fast and uses little space, we try to avoid it as much as possible.

One possible course is to precompute the steps taken by the algorithm by the first b chars read in a text window, and at the search phase to use a look-up table to perform the steps in $O(1)$ time, and then continue the algorithm normally. This obviously improves the running time complexity, but the main idea is to improve the constant factors for properly selected b.

More precicely, let us have a table $G[0 \ldots \sigma^b - 1]$, indexed by a concatenation of b (consecutive) symbols (of T), i.e. a text substring of length b. Each $G[u]$ stores a triplet (s, e, shift), which is the result of running the `Count` algorithm (the inner loops of Alg. 2 and Alg. 3) for the substring u. Assuming uniform distribution of characters, the probability of two randomly picked characters matching is $1/\sigma$, and hence on average Alg. 3 accesses at most $\lceil \log_\sigma(rm) \rceil$ text characters per window. Thus using $b = \lfloor \log_\sigma(rm) \rfloor$ gives $O(n/m)$ average time, which breaks the lower bound [33] based on comparing single characters. However, our method is not based on comparing single characters and we effectively avoid the $\log(m)$ term by "comparing" b symbols at a time. On the other hand, it is easy to see that increasing b beyond $O(\log_\sigma(rm))$ does not improve the algorithm. Note that in RAM model of computation we can read the substring of length $O(\log_\sigma(rm))$ in $O(1)$ time. The space complexity becomes

$$O\left(\sigma^{\log_\sigma(rm)}\right) = O(rm)$$

words (of $O(\log_2(rm))$ bits). This is also the additonal preprocessing time required. In practice we can choose $b = o(\log_\sigma(rm))$ to use only $o(rm \log_2(rm))$ bits of additional space to be small and more cache friendly, and still expect good practical improvement. For example, we can choose

$$b = \alpha \log_\sigma(rm)$$

for some constant $\alpha < 1$, and obtain

$$O(n \log_\sigma((rm)^{1-\alpha})/m)$$

average time using

$$O((rm)^\alpha) = o(rm)$$

words of additional space. Note that this works for plain BDM just as well. However, for e.g. ASCII alphabets ($\sigma = 256$) we have practically two choices, $b = 1$ or $b = 2$, since otherwise the space becomes too large.

One possiblity to side step the space complexity is to remap the read text characters. In the case of ASCII alphabets $\sigma = 256$, but only a small fraction of the alphabet usually appears in P. Assume that P (or $\mathcal{P}$) contains only σ_{p} distinct characters. The characters that appear

in P are mapped to the interval $0 \ldots \sigma_P - 1$, and the rest of the characters to a value σ_P. The space complexity is then reduced to only $O((\sigma_P + 1)^b)$ words, which in practice should be a large improvement. The drawback of this method is that a substring cannot be accessed in $O(1)$ time anymore, because of the character mapping, but it takes $O(b)$ time instead (but this is still optimal in the comparison model). In other words no asymptotical speed-up is obtained.

The above idea can be still improved. As we already remap the characters, we can as well use entropy bounded prefix codes, such as Huffman codes [16]. I.e. we can compute Huffman codes for the characters of P, add one code for the characters that do not appear in P, and use these codes for remapping and indexing G. Note that we do not have to Huffman code the pattern at all, the concatenated codes are used just for indexing G. The average code length is at most $H = H_0(P) + 1$ bits. As the codes have variable length, we will not read a fixed number of symbols in a text window, but rather we use b as the maximum length for the concatenated codes in bits. We therefore concatenate as many code words as can fit into b bits. Note that by using Huffman coding we are making the implicit assumption that the character distribution is not uniform, as otherwise $H_0(P) = \log_2(\sigma)$. This also means that the average number of characters inspected is more than $\log_\sigma(rm)$, that is, assuming that the probability of two characters matching is p, where $1/p < \sigma$, then the algorithm reads about $\log_{1/p}(rm)$ characters per text window, and the encoded length of this string is about $b = H \log_{1/p}(rm)$ bits. But the space needed for the tables becomes too much in this case, i.e. we have $2^b = \Omega(rm)$ words. Again, we can adjust the space-time trade-off by using smaller b, i.e. $b = \alpha H \log_{1/p}(rm)$, for some $\alpha < H \log_p(2)$ and obtain only $o(rm)$ words of additional space. Observe that using Huffman codes is still quite simple and efficient, certainly faster than any space efficient `rank` implementation. In particular, obtaining the Huffman code for a single character costs only $O(1)$ time using a look-up table of size $O(\sigma)$. Another approach is to use k-th order entropy for the codes. This achieves smaller space, i.e. the extra space becomes $o(rm) + O(\sigma^{k+1})$ words for $\alpha < H_k \log_p(2)$, if $H_k(P) < H_0(P)$, but this is interesting only for $k = o(\log_\sigma(rm))$, and in practice the performace is probably not very good.

6 Conclusions

We have obtained average optimal on-line string matching algorithm for single and multiple patterns that uses space close to the information theoretical lower bound. The algorithm is aimed especially for large pattern sets. In such cases the small memory requirements should provide good performance in practice, as the data structures fit better into the CPU cache. We plan to implement the proposed method and experimentally test the space-time trade-offs of various `rank` implementations.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[2] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[3] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[4] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[5] M. Crochemore. String-matching on ordered alphabets. *Theor. Comput. Sci.*, 92(1):33–47, 1992.

[6] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[7] M. Crochemore and D. Perrin. Two-way string-matching. *J. Assoc. Comput. Mach.*, 38(3):651–675, 1991.

[8] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[9] M. Crochemore and W. Rytter. Squares, cubes and time-space efficient string-searching. *Algorithmica*, 13(5):405–425, 1995.

[10] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002. ISBN 981-02-4782-6. 310 pages.

[11] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Compute r Science (FOCS 2000)*, pages 390–398, Washington, DC, USA, 2000. IEEE Computer Society.

[12] Z. Galil and J. Seiferas. Linear-time string matching using only a fixed number of local storage locations. *Theor. Comput. Sci.*, 13(3):331–336, 1981.

[13] A. Golynski, I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of SODA'06*, pages 368–373. ACM Press, 2006.

[14] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 841–850. ACM, 2003.

[15] R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.

[16] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. I.R.E.*, 40:1098–1101, 1951.

[17] G. Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1989.

[18] D. K. Kim, J. C. Na, J. E. Kim, and K. Park. Efficient implementation of rank and select functions for succinct representation. In *Proc. of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 315–327. Springer, 2005.

[19] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[20] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 2006. Special issue on "The Burrows-Wheeler Transform and its Applications". To appear.

[21] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[22] G. Navarro and K. Fredriksson. Average complexity of exact and approximate multiple string matching. *Theoretical Computer Science A*, 321(2–3):283–290, 2004.

[23] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 2006. To appear.

[24] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000.

[25] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

[26] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of ALENEX'07*. ACM Press, 2007.

[27] R. onzález, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

[28] R. Pagh. Low redundancy in static dictionaries with o(1) worst case lookup time. In *Proceedings of ICALP'99*, pages 595–604. Springer-Verlag, 1999.

[29] Leena Salmela, Jorma Tarhio, and Jari Kytöjoki. Multipattern string matching with q-grams. *J. Exp. Algorithmics*, 11:1.1, 2006.

[30] D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.

[31] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

[32] S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

[33] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.