Strategic Pattern Search in Factor-Compressed Text

Simon Gog[1,2], Alistair Moffat[1], and Matthias Petri[1]

[1] Department of Computing and Information Systems,
The University of Melbourne, Australia 3010
[2] Institute of Theoretical Informatics,
Karlsruhe Institute of Technology, Germany

Abstract. We consider the problem of pattern-search in compressed text in a context in which: (a) the text is stored as a sequence of factors against a static phrase-book; (b) decoding of factors is from right-to-left; and (c) extraction of each symbol in each factor requires $\Theta(\log \sigma)$ time, where σ is the size of the original alphabet. To determine possible alignments given information about decoded characters we introduce two Boyer-Moore-like searching mechanisms, including one that makes use of a suffix array constructed over the pattern. The new mechanisms decode fewer than half the symbols that are required by a sequential left-to-right search such as the Knuth-Morris-Pratt approach, a saving that translates directly into improved execution time. Experiments with a two-level suffix array index structure for 4 GB of English text demonstrate the usefulness of the new techniques.

Keywords: string search, pattern matching, suffix array, Burrows-Wheeler transform, succinct data structure, disk-based algorithm, experimental evaluation.

1 Introduction and Background

String search, or pattern search, is a classic problem in computing. Given a sequence T of n symbols and a pattern P of m symbols, both over an alphabet Σ of size σ, the requirement is to identify all locations in T at which P appears as a substring. Two paradigms for tackling this problem have emerged – if both T and P vary with every problem instance, the best that can be hoped for is linear $\Theta(n+m)$ time processing. But if T is regarded as being fixed, and only P varies with each instance, then the cost of pre-processing T to build an index can be regarded as being amortized down to zero. Large numbers of algorithms and data structures have been developed for both types of pattern search, as well as for variants of the basic problem; see, for example, Navarro and Raffinot [11]. Our work in this paper fits in the second "static T" category, but also requires application of techniques suited to the first "dynamic T" paradigm.

Sequential Pattern Search. The Knuth-Morris-Pratt (KMP) [10] and Boyer-Moore (BM) [1] methods remain significant more than 35 years after they were first developed. The KMP approach scans T from left to right, extending a prefix of P known to be in alignment with T; if a non-matching symbol is encountered, the pattern is shifted right by the amount indicated in a pre-computed table that is based on P (and not on T). In the BM method, the checking is from right-to-left in P. Two shift tables are used, the "good

E. Moura and M. Crochemore (Eds.): SPIRE 2014, LNCS 8799, pp. 1–12, 2014.
© Springer International Publishing Switzerland 2014

suffix" table that has a similar function to the KMP table; and a "bad symbol" table, which, for each symbol in Σ, records the rightmost occurrence of it in P.

Horspool [9] noted that use of the bad symbol shift associated with the symbol in T tentatively matched against $\mathsf{P}[m-1]$ was sufficient for fast execution on average, since it never results in negative shifts, and is likely to create long shifts on average. The combination of right-to-left processing and a bad-symbol shift array is referred to as the BMH mechanism. Sunday [16] noted that the symbol in T *after* the last one of the current alignment could also be used for the same purpose, since it too must be part of the next alignment after the shift of P has taken place. Together, these two approaches then lead to Smith's [15] proposal to make use of the larger of the Horspool bad-shift and the Sunday bad-shift. We make use of these ideas in the development below. A 1997 web site[1] developed by Christian Charras and Thierry Lecroq gives details and examples of all of these methods, plus many more, as does the book of Navarro and Raffinot [11] and a survey of recent results by Faro and Lecroq [3].

Index-Based Pattern Search. There is again a myriad of methods in this category. Best-known are the suffix array, the suffix tree, and compressed/succinct variants thereof. The FM-INDEX of Ferragina and Manzini [5] represents T in compressed form (that is, in fewer than $n \log \sigma$ bits) yet still provides pattern search in $O(m \log \sigma)$ time. The combination of compact space and fast access make the FM-INDEX highly applicable for in-memory searching applications. On the other hand, the access pattern within the FM-INDEX is highly non-sequential, and it is not suitable for use on secondary storage devices such as mechanical disk and SSD memory.

The RoSA. In recent work, Gog et al. [8] introduce an indexed data structure for pattern search called the ROSA, or *reduced space on-disk suffix array*, as a mechanism to support exact pattern search. As with the previous LOF-SA structure of Sinha et al. [14], the ROSA supports efficient pattern search over very large static sequences by constructing a suffix array, and partitioning it into on-disk blocks. Each suffix block contains at most b pointers, and is formed so that every string addressed by the block has a unique common prefix, known as the *block prefix string*. The value of b is fixed at the time the index is constructed; Gog et al. [8] make use of $b = 4{,}096$ in their experiments. The first innovative feature of the ROSA is that the in-memory index is structured as a *condensed BWT*, that contains all of the block prefix strings, so that the suffix block a given pattern falls in to (if it exists at all) can be efficiently determined. The use of the condensed BWT, and careful engineering in regard to storage of bitvectors and pointers, mean that the in-memory index can be as small as just a few percent of the original string. In experiments using multi-gigabyte files of English text, Gog et al. [8] show that the ROSA's in-memory index requires as little as 2% of the original text, with *count* queries – in which the objective is to determine how many occurrences there are of the pattern P, but not their exact locations – requiring at most two accesses to secondary storage: one to fetch a suffix block of at most b pointers, and a second to fetch a section of the original text T. The second access is required to verify that the pattern does indeed exist, and is not a false-positive arising from the use of a *bit-blind tree* [4] during the within-block search [8].

[1] `http://www-igm.univ-mlv.fr/~lecroq/string/`

Suffix blocks have common prefix strings allowing for *block reductions* to be identified which map blocks of the suffix array to subsections of other blocks, allowing disk space to also be reduced. For the same test file, the space required by the suffix array is less than $2n$ bytes. Including the text as well, the total storage cost of the RoSA structure is less than $3n$ bytes when representing English text [8], substantially better than the $5n$ or $9n$ required by uncompressed suffix array structures.

In a followup paper, Gog and Moffat [7] further reduce the RoSA's space cost in two key ways. First, they re-use the block prefix strings as a static phrase-book for greedy-dictionary compression, and show that the condensed BWT index can be used to decode factors with only a small amount of overhead space. Second, Gog and Moffat approximate each of the stored suffix pointers (which, because of the compression of T, is a factor address rather than a byte address), truncating it to a multiple of R, a second parameter selected at the time the index is constructed. The first change reduces both the space required for T and the space required by each suffix pointer, since there are fewer factors in T than there are bytes; the second change saves a further approximately $\log_2 R$ bits from each suffix pointer. Using both techniques, a complete two-level RoSA structure for the same 62.5 GB file of English text requires less than $2n$ bytes when $R = 64$, with searching time approximately doubled compared to the $R = 1$ situation [7].

Our Contribution. We further enhance the RoSA by exploring alternative sequential pattern matching options, improving querying costs dramatically for large values of R, exactly the ones that give rise to the most compact index. In particular, we introduce two Boyer-Moore-like searching mechanisms: one that makes use of a suffix array constructed over the pattern; and one that makes use of a shift matrix covering a total of $\sigma \times (m+1)$ different position/symbol combinations. Both mechanisms decode fewer than half the symbols that are required by the previous KMP approach, a saving that translates directly into improved execution time. We give detailed experiments with a two-level suffix array index structure for 4 GB of English text demonstrate the usefulness of the new techniques.

2 Search in Factorized Text

Each suffix block in the RoSA is structured as a bit-blind tree (see Gog et al. [8] for a description and an example) so that it can be quickly queried after it has been read from disk. The drawback of the bit-blind tree is that once the potential location in T for the pattern P has been identified via the index, it must be checked to ensure that it is not a false match. In the original RoSA the suffix pointers indicate byte offsets in T, and checking is easy. For a pattern of m symbols, a second disk access fetches a block of T, and then m character comparisons are required.

But when the suffix pointers address "div R" approximated factor numbers, the checking process is more costly. Now a sequence of $R + m - 1$ factor identifiers is supplied, and P must be searched for in the variable-length string that those factors represent. Table 1 lists the low-level operations that apply to compressed factors, and the cost of each such operation when the factors are represented via the condensed BWT structure. To decode a single factor identified by the reference f, function *length_of_factor*(f) is

Table 1. Operations used during factor decoding, where f is a reference to a factor and is assumed to include the necessary state variables. See Figure 4 of Gog and Moffat [7] for details.

Operation	Returns...	Time
length_of_factor(f)	the length in symbols of the factor	$O(1)$
final_symbol(f)	the rightmost symbol of the factor	$O(1)$
next_symbol(f)	the next (from the right) symbol of the factor	$O(\log\sigma)$

called to initialize the state variables and determine the length $len(f)$ of that factor (in terms of decoded symbols); then the rightmost symbol is accessed using *final_symbol*(f); and finally a loop iterates $len(f)-1$ times, calling *next_symbol*(f) to fill in the remaining symbols of that factor, from right to left. Each call to the latter function requires $O(1)$ rank and $O(\log\sigma)$ select operations, where σ is the cardinality of the alphabet – for example, $\sigma = 256$ for byte streams, and perhaps $\sigma \approx 10^6$ or more for streams of word tokens. The *final_symbol*(f) is a much faster operation than *next_symbol*(f); indeed, it is the final component of all calls to *next_symbol*(f). In our implementation *final_symbol*(f) (and the equivalent in each call to *next_symbol*(f)) is implemented as a local $O(\log\sigma)$ binary search, to determine the current symbol. We note that the binary search could be replaced with an $O(1)$-time bitvector operation [13] for a slight further speed advantage, and this is what is presumed in Table 1.

That is, we face the set of constraints outlined in the abstract: (a) the text is stored as a sequence of factors against a static phrase-book; (b) decoding of factors is from right-to-left; and (c) extraction of each symbol in each factor requires $H_0(\mathsf{T})$ time on average, where $H_0(\mathsf{T})$ is the zeroth order entropy of the original text.

In their presentation, Gog and Moffat [7] describe the use of the KMP pattern search algorithm, with factors expanded on-demand as required in a left-to-right manner, starting with the first one. If $f[i]$ is the ith factor in the fragment of T that is being searched, $0 \le i \le R+m-1$, then $cum[i] = \sum_{j=0}^{i-1} len(f[j])$ is the relative starting point in T of the ith factor. With this arrangement, the last valid offset at which pattern alignment is possible is given by $F = cum[R+1]-1$; that is, at the final symbol of the first R factors. The other $m-1$ factors that are passed to the search function are a worst-case allowance to ensure that the entirety of the pattern is covered.

If it is assumed that P appears in the fragment of T at an alignment that is equally likely to be any value between 0 and F, then the expected number of symbols decoded by the KMP-based approach is given by

$$\frac{F}{2} + (m-1) + \frac{F/R}{2},$$

where $F/2$ is the cost of reaching the starting point of the matching alignment; $m-1$ is the number of further symbols that must be checked to confirm the alignment; and F/R is the average number of symbols per factor.

Gog and Moffat [7] explored a range of values of R in their experiments, working with English text and a ROSA stored on SSD secondary memory. They demonstrated that when $R = 16$, around half a millisecond is required by the KMP sequential search phase, with overall search times (for *count* queries) under two milliseconds; but that

when R is increased to 256, more than 80% of query computation was spent on the pattern-search phase, more than 4 milliseconds out of a total query time a little over 5 milliseconds. Our goal is to reduce that ratio by replacing the KMP search module by methods specifically targeted for the constraints listed above.

3 Strategic Search

Searching in the factorized text representation follows a restricted access model, and it is no longer appropriate to assume a random-access machine model of computation. For example, if i corresponds to the first symbol of a new factor of length p, the cost of accessing $\mathsf{T}[i]$ is $O(p \log \sigma)$. However, after accessing i, positions $\mathsf{T}[i+1 \ldots i+p-1]$ can be referenced without incurring additional decoding costs.

On the other hand, the search pattern is still represented in plain text. Thus, compared to accessing T, operations on P are relatively inexpensive. This imbalance in costs opens up two aspects of the pattern matching process as being potential targets for investigation: (1) text access strategies during pattern alignment which are aware of the underlying factor representation and focus towards the end of factors whenever possible; and (2) more complex pattern pre-processing steps – including the construction of larger shift tables – which enable longer shifts during the matching process.

Maintaining a Rightward Focus. Both KMP and BMH align P at a certain position i in the text T, and compare symbols in $\mathsf{T}[i..i+m-1]$ against their tentative equivalents in $\mathsf{P}[0..m-1]$. The KMP approach starts with $\mathsf{T}[i]$ and seeks to build matching prefixes, whereas BMH starts with $\mathsf{T}[i+m-1]$ and seeks to builds matching suffixes. But starting at the ends of factorized pattern can be expensive. Instead, we suggest that the rightmost factor fully contained within the current alignment of P be identified, and then decoded from the right. As each symbol is extracted from the factor, the pattern is checked at the equivalent position; if a mismatch is identified, the pattern is shifted as far to the right as is consistent with the characters that have been decoded. Figure 1 shows an example of the rightward focused alignment process, searching for the string `ACTTTGCCGTATAAGACG` for which $m = 18$. Presuming that the shifts can be calculated as shown, only three different alignments are explored before the match is found, and just 23 symbols are decoded while that is taking place.

Determining Shifts Using A Suffix Array. Each time a mismatch occurs, the alignment position is shifted. Algorithms such as KMP or BMH analyze P at the beginning of the search process to pre-compute the possible shifts. The rightward focus requires that for a given pattern P and a short fragment F drawn from the text T, that the fragment be shifted as far leftward over P as is consistent with the currently-decoded symbols. That is, the problem is now flipped – the objective is to determine locations in P at which F occurs, in order to determine possible pattern alignments.

For example, consider Figure 1, in which the substring `TG` is decoded during the first alignment of P. Determining the rightmost occurrence in P to the left of the current alignment of the longest suffix of the substring `TG` gives rise to the second alignment. Because two further factors are now within the span of P in its proposed alignment, the

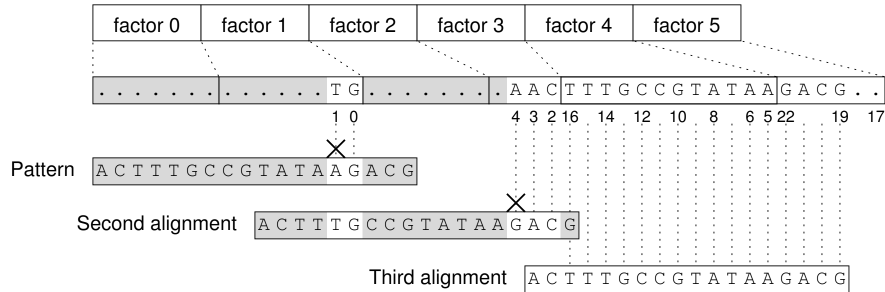

Fig. 1. Search process with partial factor decoding and rightward focus. Numbers show the order in which symbols are decoded. Grayed-out symbols in T are neither decoded nor accessed.

focus shifts to the right. Two symbols (labeled 2 and 3) are decoded from the fourth factor, and match P. But when a third symbol is decoded, the resulting fragment `AAC` is in conflict with the corresponding positions in P. Indeed, substring F = `AAC` does not occur in P at all. However, the suffix `AC` of F is a prefix of P, and so a complete shift is still not possible. In general, if F is not found in P, the pattern is re-aligned to match the longest suffix of F that matches a prefix of P, including the empty suffix if there are none longer. If the empty suffix is detected, the result is a complete shift of P past all currently-decoded characters.

To carry out the required search, we pre-process P to compute a suffix array SA for it – in effect, constructing a one-off index for the pattern that can be used to locate occurrences of text fragments F. Index construction might be prohibitively expensive for regular pattern search, but the high costs associated with accessing T mean that it can be considered in this context. Algorithm 1 describes how each shift is computed, given the inputs F, P, the current alignment *fpos* of F in P, and a suffix array SA over P.

There are two stages. In the first stage (steps 3–9), *backwards search* is carried out using SA. That is, F is processed from right to left (as symbols are decoded from the factor) to determine ranges (sp, ep) within SA that always match the currently-decoded fragment F. For each range (sp, ep) it is determined if the current suffix of F is a prefix of P (*head_overlap*) by determining if $0 \in \mathsf{SA}[sp \ldots ep]$. The first stage of the algorithm terminates once F is completely processed, or if the range (sp, ep) becomes empty. The second stage (steps 10–18) determines the correct shift amount. If F occurs in P, the rightmost occurrence to the left of *fpos* corresponds to the next alignment. If no such occurrence exists, a check is made as to whether a suffix of F has matched a prefix of P (recorded by variable *head_overlap*) at any stage. If not, P is moved completely past the occurrence of F. Figure 1 does not illustrate an instance of this step.

Algorithm 1 is a high-level description, and several details have been omitted. The actual implementation uses the usual technique of building SA over the reverse P^{r} of P, similar to the suffix automaton of the reverse pattern of the BDM algorithm, allowing iterative determination of ranges for suffixes of F; and also makes use of an inverse suffix array for P^{r}, in order to expedite step 5.

Algorithm 1. Searching for a fragment F in a restricted section of string P.

```
0: Decide whether factor F[0...p−1] has an alignment with P[0...m−1] to the left of offset
   fpos. Array SA is a suffix array for P. Returns the shift such that P can be aligned to F, the
   rightmost matching prefix of F in P or past F if no matching prefix is found. Symbols in F
   are decoded on demand using the functions shown in Table 1.
1: function fragment_shift(P[0...m−1], F[0...p−1], fpos)
2:     (sp,ep) ← (0,m−1), i ← p, head_overlap ← 0, shift ← 0
3:     while |(sp,ep)| > 0 and i ≠ 0 do                        ▷ Search for F in P
4:         (sp,ep) ← refine_interval(SA, P, (sp,ep), F[i−1])
5:         if 0 ∈ SA[sp...ep] then                    ▷ Suffix of F matches prefix of P
6:             head_overlap ← head_overlap + 1
7:         end if
8:         i ← i − 1
9:     end while
10:    if |(sp,ep)| > 0 then                                   ▷ Found F in P
11:        candidate ← max(SA[i] ∈ SA[sp...ep] | SA[i] < fpos)
12:        if candidate ≠ ∅ then                     ▷ Found to the right of fpos?
13:            shift ← fpos − candidate
14:        end if
15:    end if
16:    if shift = 0 then                           ▷ No full match to the left of fpos
17:        shift ← fpos + p − head_overlap            ▷ Compute prefix match, if any
18:    end if
19:    return shift
20: end function
```

Determining Shifts Using A BMH Matrix. The suffix-array based approach has two potential disadvantages: the time taken to build the suffix array for P^{r} may dominate the matching time; and it is unable to fully exploit non-contiguous fragments. To see the issue posed by the latter concern, consider Figure 1 again, and suppose that the "C" decoded at label 2 had in fact been an "A". Working solely with the right-focused factor, in this case the suffix array would generate a shift of one, ignoring the additional (and conflicting) information provided by the "TG" fragment that is also available in factor 1.

To address these concerns, we have explored a second mechanism, based even more closely on the Boyer-Moore-Horspool pattern search algorithm. The BMH mechanism uses the last position within the alignment to determine the shift, regardless of where in the pattern a mismatch occurs, based on a "bad symbol" table S. For each symbol $s \in \Sigma$, $S[s]$ stores the value $m-1-\ell_s$, where ℓ_s is the index of the last occurrence of s in $\mathsf{P}[0\ldots m-2]$, $\ell_s = \max\{0 \le k < m-1 \mid \mathsf{P}[k] = s\}$, or $S[s] = m$ if s does not appear in $\mathsf{P}[0\ldots m-2]$. When shifting on from an explored alignment i, BMH sets $i \leftarrow i + S[\mathsf{T}[i+m-1]]$, using the symbol in T currently placed against $\mathsf{P}[m-1]$ as a single reference point against which the proposed next alignment is located.

In the RoSA context $\mathsf{T}[i+m-1]$ may not be known. On the other hand, the preprocessing on P is not required to be $O(m)$. The solution is to extend the shift table S, and make it two dimensional, setting

Table 2. Example of BMH shift matrix S for P = ACTTTGCCGTATAAGACG

	A	C	T	T	T	G	C	C	G	T	A	T	A	A	G	A	C	G	–
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	1	1	2	3	4	5	6	7	8	9	10	1	2	1	1	2	1	2	3
C	1	2	1	2	3	4	5	1	1	2	3	4	5	6	7	8	9	1	2
G	1	2	3	4	5	6	1	2	3	1	2	3	4	5	6	1	2	3	1
T	1	2	3	1	1	1	2	3	4	5	1	2	1	2	3	4	5	6	7

$$S[s,p] \leftarrow p - \begin{cases} -1 & \text{if } s \notin \mathsf{P}[0 \ldots p-1] \\ \max\{0 \le k < p \mid \mathsf{P}[k] = s\} & \text{otherwise}, \end{cases}$$

where $s \in \Sigma$ is a symbol in the alphabet; p is the length of a prefix of the pattern, $0 \le p \le m$; and $S[s,p]$ records the interval in P between offset p and the rightmost previous occurrence of symbol s. This shift table is similar to the δ_1 table of Colussi [2] which uses it as part of a more complex string matching algorithm. An example is shown in Table 2 for the same pattern as is used in Figure 1; clearly, the table requires $O(m\sigma)$ time to construct. Note also that the shift for the character aligned one past the end of the pattern can also be computed, as was first proposed by Sunday [16].

To apply the table S, every decoded character in T that overlaps the current alignment i (including $\mathsf{T}[i+m]$, if it has been decoded) is used as an index into S, together with the corresponding offset in P. Each of those indicated values in S represents a minimum shift amount; hence, the largest of them also represents a minimum shift. Resuming the previous example, if the symbol labeled 2 in Figure 1 was an "A" rather than a "C", three elements of S would be considered: $S[\text{"A"},16]$, $S[\text{"G"},5]$, and $S[\text{"T"},4]$. The corresponding shifts (Table 2) are 1, 6, and 1; the maximum of these, 6, is used as the overall shift amount. More generally, if the current proposed alignment of P commencing at $\mathsf{T}[i]$ is to be shifted, then the update performed is given by:

$$i \leftarrow i + \max\{S[\mathsf{T}[i+j], j] \mid 0 \le j \le m \text{ and } \mathsf{T}[i+j] \text{ has been decoded}\} .$$

Smith [15] also proposed the use of a "max" shift amount, based on the two shift vectors $S[*][m-1]$ and $S[*][m]$, shown as the last two columns in Table 2; and that Raita [12] explored the notion of checking non-sequential symbols from P, with his proposal to compare T against $\mathsf{P}[m-1]$, $\mathsf{P}[0]$, and $\mathsf{P}[m/2]$ before looking at any other symbols.

4 Experiments

Methodology and Implementation. Our experimental study extends the original RoSA implementation, adding four further factor matching algorithms. Including the original KMP implementation of Gog and Moffat [7], we are able to explore: Exhaustive left-to-right matching (denoted EXH); KMP; Boyer-Moore-Horspool (BMH); the new suffix array based approach (SA); and the two-dimensional multiple-BMH technique

(MBMH). We measure the execution time cost of the factor matching process, together with the relative percentage of decoded symbols for different sample rates R, and different pattern lengths m, in all cases using the same block parameter $b = 4{,}096$ as was employed by Gog and Moffat [7]. We do not measure the other steps in the ROSA query process; they were explored in detail in the two previous studies [7,8], and those components of the implementation are reused here without alteration. All algorithms are implemented in C++11 and compiled using GCC 4.8.1 with optimizations. The suffix array for P^{r} was created using Yuta Mori's LIBDIVSUFSORT library[2] version 2.0.1.

Data Sets, Queries and Test Environment. We use the WEB-4G prefix of the data set used in the experimental evaluation of Gog and Moffat [7], and generate 1000 patterns for each length $m \in \{4, 10, 20, 40, 100\}$, with each pattern occurring 10–100 times in the collection. We built two ROSA indexes with $b = 4{,}096$, and factor approximation rates $R = 16$ and $R = 256$. Our machine was equipped with 148 GB RAM and we used one Intel Xeon core (E5640) running at 2.67 Ghz, approximately 1.6 times the clock speed of the 1.7 GHz Macbook Air used by Gog and Moffat. We report only the times required to perform the factor matching step, which does not include the use of the condensed BWT, does not include the access of a suffix block, does not include the use of the bit-blind tree within the block, and does not include fetching the set of factor identifiers. For the relative runtimes of these phases see Figure 5 of Gog and Moffat [7].

Symbols Decoded. In uncompressed text, the number of comparisons performed by an algorithm is generally a good indicator of run time performance. However, decoding symbols is the dominant cost when matching in factor compressed text. Figure 2 (top) shows the number of decoded symbol per query, for different sample rates and pattern lengths. When $R = 16$, the number of decoded symbols is roughly one order of magnitude smaller than for $R = 256$, as there are 16 times more factors which can contain the pattern. The relative performance of each algorithm remains similar. The two algorithms which employ left-to-right processing – EXH and KMP – decode the most symbols. The classic BMH approach is more efficient, as alignment is performed right-to-left, and so some factors are only partially decoded. As the pattern length increases, this effect is more visible, since the percentage of symbols that can be skipped per alignment increases. The two advanced methods – SA and MBMH – perform much better than the classical pattern matching algorithms. For $R = 16$ and $m = 4$, the new methods decode roughly half as many symbols as BMH, and decode a third of the symbols of EXH and KMP. For larger patterns and larger match regions, the difference is even more marked. For $R = 256$ and $m = 100$, both SA and MBMH on average decode 20% of the symbols of BMH, and one eighth of the symbols required by KMP and EXH.

The fraction of "possible" symbols decoded is shown in the bottom half of Figure 2, calculated as the percentage of symbols decoded compared to a "blind search" algorithm which aligns P to all positions amongst the first R factors. That is, the denominator is the count of symbols in all of the first R factors, plus the whole of the further factors required to span a further $m - 1$ symbols. If the occurrence of P is uniformly distributed over the positions within the first R factors, as was assumed earlier,

[2] Available at `https://code.google.com/p/libdivsufsort`

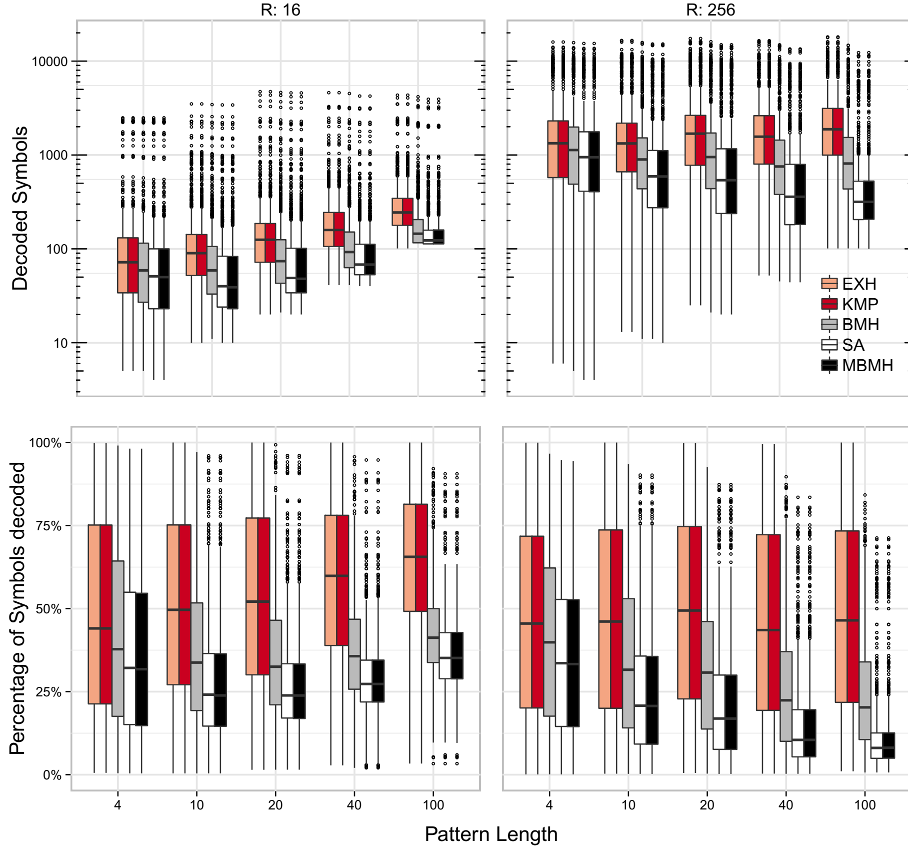

Fig. 2. Total symbols decoded (top) and percentage of decoded symbols (bottom) per query for queries of length $m \in \{4, 10, 20, 40, 100\}$, using $b = 4096$ and $R \in \{16, 256\}$ over WEB-4G. The boxes represent the median and quartiles of the measured distributions, and the whiskers depict elements within 1.5 times of the corresponding inter-quartile ranges.

then algorithms that access all symbols in each alignment should on average decode a little over 50% of that total number of symbols. For $R = 256$, both exhaustive decoding algorithms (EXH and KMP) do indeed decode close to 50% of the available symbols. For $R = 16$ the total number of positions covered by the first R factors is much smaller relative to the pattern; hence, for EXH and KMP, the percentage of decoded symbols is over 60% when $R = 16$.

Regular BMH decodes a smaller fraction of the symbols than the two exhaustive schemes. For $R = 256$ and large patterns, around 25% of all symbols are decoded. The SA and MBMH approaches significantly outperform the other three. The relative difference between the different methods increases as the pattern size increases, because longer patterns allow larger shifts to be performed. For $R = 256$ the percentages of decoded symbols for BMH, SA and MBMH decreases as the pattern length increases. For the small sample rate ($R = 16$), this is not the case as the number of decoded symbols

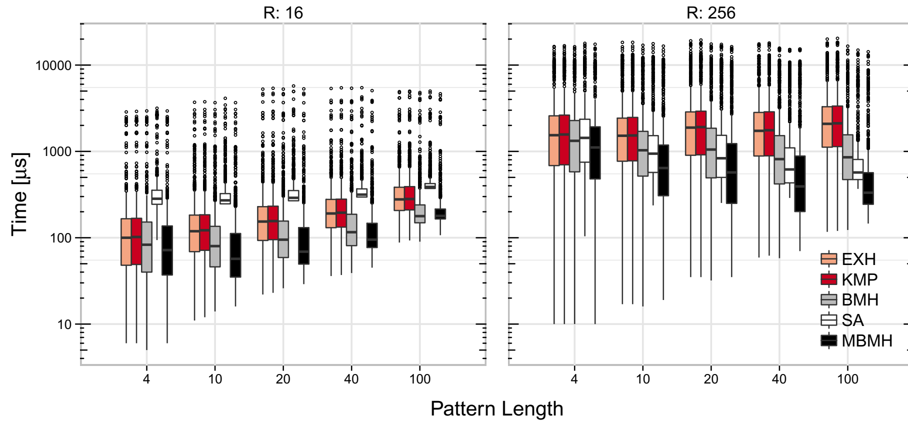

Fig. 3. Time in microseconds per query for queries of length $m \in \{4, 10, 20, 40, 100\}$, using $b = 4096$ and $R \in \{16, 256\}$ over WEB-4G

for large patterns (roughly 100, as shown in the top graph) is close to the length of the pattern. As a match always occurs in our experiments, at least m symbols have to be decoded by all methods. The two new approaches – SA and MBMH – require similar numbers of characters to be decoded, across all of the configurations tested.

Factor Matching Runtime Performance. As already noted, factor matching time can be a dominant component of overall ROSA search time, especially for large values of R. Replacing left-to-right search methods by right-to-left ones (BMH, SA, MBMH) substantially decreases the number of symbols decoded. Figure 3 shows how these savings translate into reduced execution times, showing the cost of the matching phase in microseconds for each of the different matching algorithms. When $R = 256$, the speed differentials match the arrangement shown in Figure 2. For shorter patterns the relativities are similar, but BMH outperforms both EXH and KMP for larger patterns. The original BMH approach is three times slower than SA, and five times slower than MBMH, validating the decision to search for more complex shift mechanisms. For $R = 16$, the SA method is slower than all other methods, whereas MBMH remains fast. This is caused by the additional time spent to construct the suffix array – small values of R don't allow the pre-processing investment to be sufficiently recouped. Suffix array construction takes between 150 to 350 microseconds in these experiments, and the SA method is only viable for large R and $m \geq 10$, whereas MBMH remains fast in all instances.

5 Conclusion

We have described enhanced string searching mechanisms that provide accelerated pattern matching when a particular combination of constraints applies, most notably, when access to elements of the text T is not $O(1)$ per operation. The particular application for the new methods is in the ROSA large-scale suffix array data structure, which provides indexed pattern search over large texts for which it is not possible to hold a compressed

index, such as an FM-INDEX, in memory. The two mechanisms we describe expend additional pre-processing time on building multi-faceted shift structures, to reduce the number of alignments that must be checked, and hence reduce the number of characters of T that are accessed. Our experimental results show that the new methods provide a two-fold speed improvement for patterns of length $m = 20$, and a six-fold improvement for patterns of length $m = 100$. Reducing the cost of the pattern-matching phase in ROSA searching gives rise to an equivalent saving in overall querying costs. Other pattern search mechanisms beyond the two canvassed here may also be applicable. In particular, because the pattern pre-processing cost can be allowed to be super-linear in m, many further options are available [3].

Acknowledgment. This work was supported under Australian Research Council's Discovery Projects funding scheme (project number DP110101743).

Software. The ROSA software is available at `https://github.com/mpetri/RoSA`; it is based on the Succinct Data Structure Library (SDSL) [6].

References

1. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. C. ACM 20, 1075–1091 (1977)
2. Colussi, L.: Fastest pattern matching in strings. J. Alg. 16, 163–189 (1994)
3. Faro, S., Lecroq, T.: The exact online string matching problem: A review of the most recent results. ACM Comput. Surv. 45(2), 13:1–13:42 (2013)
4. Ferragina, P., Grossi, R.: The string B-tree: A new data structure for search in external memory and its applications. J. ACM 46(2), 236–280 (1999)
5. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52(4), 552–581 (2005)
6. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: Proc. Symp. Experimental Algorithms, pp. 326–337 (2014)
7. Gog, S., Moffat, A.: Adding compression and blended search to a compact two-level suffix array. In: Proc. Symp. String Processing and Inf. Retrieval, pp. 141–152 (2013)
8. Gog, S., Moffat, A., Culpepper, J.S., Turpin, A., Wirth, A.: Large-scale pattern search using reduced-space on-disk suffix arrays. IEEE Trans. Knowledge and Data Engineering 26(8), 1 (2014)
9. Horspool, R.N.: Practical fast searching in strings. Soft. Prac. & Exp. 10(6), 501–506 (1980)
10. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comp. 6(1), 323–350 (1977)
11. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge University Press (2002)
12. Raita, T.: Tuning the Boyer-Moore-Horspool string searching algorithms. Soft. Prac. & Exp. 22(10), 879–884 (1992)
13. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. ACM-SIAM Symp. Discrete Algorithms, pp. 233–242 (2002)
14. Sinha, R., Puglisi, S.J., Moffat, A., Turpin, A.: Improving suffix array locality for fast pattern matching on disk. In: Proc. ACM SIGMOD Int. Conf. Management of Data, pp. 661–672 (2008)
15. Smith, P.D.: Experiments with a very fast substring search algorithm. Soft. Prac. & Exp. 21(10), 1065–1074 (1991)
16. Sunday, D.M.: A very fast substring search algorithm. C. ACM 33(8), 132–142 (1990)