Journal of Discrete Algorithms 11 (2012) 37–50

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

String matching with alphabet sampling ☆

Francisco Claude [a,1], Gonzalo Navarro [b,2], Hannu Peltola [c,3], Leena Salmela [d,*,4], Jorma Tarhio [c,3]

[a] David R. Cheriton School of Computer Science, University of Waterloo, Canada
[b] Department of Computer Science, University of Chile, Chile
[c] Department of Computer Science and Engineering, Aalto University, Finland
[d] Department of Computer Science, University of Helsinki, Finland

ARTICLE INFO

Article history:
Available online 16 September 2010

Keywords:
Pattern matching
Semi-indexes
Alphabet sampling

ABSTRACT

We introduce a novel alphabet sampling technique for speeding up both online and indexed string matching. We choose a subset of the alphabet and extract the corresponding subsequence of the text. Online or indexed searching is then carried out on the extracted subsequence, and candidate matches are verified in the full text. We show that this speeds up online searching, especially for moderate to long patterns, by a factor of up to 5, while using 14% extra space in our experiments. For indexed searching we achieve indexes that are as fast as the classical suffix array, yet occupy less than 50% extra space (instead of the usual 400%). Our experiments show no competitive alternatives exist in a wide space/time range.

1. Introduction

The string matching problem is to find all the occurrences of a given pattern $P = p_1 p_2 \dots p_m$ in a large text $T = t_1 t_2 \dots t_n$, both being sequences of characters drawn from an alphabet Σ of size σ. For simplicity, we assume $\Sigma = [1, \sigma]$.

One approach to string matching is *online* searching, which means the text is not preprocessed. Thus these algorithms need to scan the text when searching and their time cost is of the form $\mathcal{O}(n \cdot f(m))$, for some $f(m) \geqslant 1/(2m-1)$ even in the best case. The worst-case complexity of the problem is $\Theta(n)$, first achieved by the Knuth–Morris–Pratt algorithm [14]. The average complexity of the problem is $\Theta(n \log_\sigma m/m)$ [25], achieved for example by the BDM algorithm [5]. Some non-optimal algorithms such as the Boyer–Moore–Horspool (BMH) algorithm [11] are very competitive in practice.

The second approach, *indexed searching*, tries to speed up searching by preprocessing the text and building a data structure that allows searching in $\mathcal{O}(g(m,n) + occ \cdot h(n))$ time, where occ is the number of occurrences of the pattern in the text, $g(m,n) \geqslant m$ is the cost per pattern and $h(n) \geqslant 1$ is the cost per occurrence. Popular solutions to this approach are suffix trees [3] and suffix arrays [15]. The first gives an $\mathcal{O}(m + occ)$ time solution, while the suffix array gives an $\mathcal{O}(m \log n + occ)$ time complexity, which can be improved to $\mathcal{O}(m + \log n + occ)$ [15] or even $\mathcal{O}(m + occ)$ [1], using extra space. The problem of these approaches is that the space needed is too large for many practical situations (4–20 times the text size). Recently,

☆ An earlier version of this paper appeared in the *Proceedings of SPIRE'08*, pp. 87–98.

* Corresponding author at: Department of Computer Science, P.O. Box 68, FI-00014 University of Helsinki, Finland. Tel.: +358 9 1915 1275.
E-mail addresses: fclaude@cs.uwaterloo.ca (F. Claude), gnavarro@dcc.uchile.cl (G. Navarro), hpeltola@cs.hut.fi (H. Peltola), leena.salmela@cs.helsinki.fi (L. Salmela), jorma.tarhio@aalto.fi (J. Tarhio).

[1] Partially funded by Go-Bell Scholarships and David R. Cheriton Scholarships program.
[2] Partially funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.
[3] Supported by Academy of Finland grant 134287 (IASMB).
[4] Supported by Academy of Finland grant 118653 (ALGODAN).

1570-8667/$ – see front matter
doi:10.1016/j.jda.2010.09.004

a lot of effort has been spent to compress these indexes [21] obtaining a significant reduction in space, but requiring considerable implementation effort and a high time per occurrence reported [7]. An intermediate approach, in space and time, is the *sparse suffix array* [12], which indexes every h-th position of the text.

In this work we explore techniques based on sampling the alphabet by selecting a set of characters from the alphabet, and then forming the sampled text with only the positions that belong to the selected subset. We first apply a sequential scanning algorithm to this sampled text, obtaining an approach between online and indexed searching. We call this kind of structure a *semi-index*. This is a data structure built on top of a text, which permits searching faster than any online algorithm, yet not as fast as indexed algorithms. Its search complexity is still of the form $\mathcal{O}(n \cdot f(m))$, but the $f(m) \geqslant 1/(2m-1)$ barrier can be broken. To be interesting, a semi-index should in addition be easy to implement and require little extra space. Several other semi-indexes exist in the literature, even without using that name. Some examples are block addressing inverted indexes [16], q-gram indexes [2,20,23], directly searchable compression formats [18,24], and other sampling approaches.

We also consider indexing the sampled text. We build a suffix array indexing the sampled positions of the text, and get a sampled suffix array. This approach is similar to the sparse suffix array [12] as both index a subset of the suffixes, but the different sampling properties induce rather different search algorithms and performance characteristics.

A challenge in our method is how to choose the best alphabet subset to sample. We present analytical results, supported by experiments, that simplify this process by drastically reducing the number of combinations to try. We show that it is sufficient in practice to sample the least frequent characters up to some limit.

In both cases, online and indexed, our sampling technique significantly improves upon the state of the art, especially for relatively long search patterns. For example, online searching is speeded up by a factor of up to 5 on English text, while using 1.05 extra bits per symbol (which translates into 14% extra space if symbols are stored in bytes). For indexed searching we achieve indexes that are as fast as the classical suffix array, yet in practice their extra space is less than 50% of the text size (instead of the 400% required by the classical suffix array).

2. Alphabet sampling

The main idea of our semi-indexed approach is to choose a subset of the alphabet to be the sampled alphabet and then to build a subsequence of the text by omitting all characters not in the sampled alphabet. When searching, we build the sampled pattern by omitting all pattern characters not in the sampled alphabet and then search for this sampled pattern in the sampled text. At regular intervals of the sampled text, we map its positions to their corresponding positions in the original text. For each candidate returned by the search on the sampled text, we verify a short range of the original text with the help of the position mapping.

Let $T = t_1 \dots t_n$ be the text over the alphabet Σ and $\Sigma_X \subset \Sigma$ the sampled alphabet. The proposed semi-index is composed of the following items:

- The sampled text T_X: Let $T_X = t_{i_1} \dots t_{i_{n_X}}$ be the sequence of the t_i's that belong to the sampled alphabet Σ_X. The length of the sampled text is n_X.
- The position mapping M: A table of size $\lfloor n_X/q \rfloor$ such that $M[i] = j$ such that $T[j]$ corresponds to $T_X[q \cdot i]$; we also set $M[0] = 0$.

Given a pattern $P = p_1 \dots p_m$, search on this semi-index is carried out as follows. Let $P_X = p_{j_1} \dots p_{j_{m_X}}$ be the subsequence of p_i's that belong to the sampled alphabet Σ_X. The length of the sampled pattern is m_X. The sampled text T_X is then searched for P_X, and for every occurrence, the positions to check in the original text are delimited by the position mapping M. We note that if the occurrence in T_X includes a mapped position, it suffices to check only one position of T as we know the exact position of one of the characters of P in the possible occurrence. Otherwise, if the sampled pattern is found at position i_r in T_X, the area

$$T\big[M[i_r/q] + (i_r \bmod q) - j_1 + 1 \dots M[i_r/q+1] - \big(q - (i_r \bmod q)\big) - j_1 + 1\big]$$

is checked for possible start positions of real occurrences. If P_X becomes empty, i.e. P contains no sampled characters, we search for P in the original text T.

For example, if the text is $T = abaacabdaa$, the sampled text built omitting the a's ($\Sigma_X = \{b, c, d\}$) is $T_X = t_2 t_5 t_7 t_8 = bcbd$. If we map every other position in the sampled text, the position mapping M is $\{5, 8\}$. To search for the pattern $P = acab$ we omit the a's and get $P_X = p_2 p_4 = cb$. We search for $P_X = cb$ in $T_X = bcbd$, finding an occurrence at position 2. We note that $T_X[2]$ is mapped and thus it suffices to verify for an occurrence starting at position 4 and we find a match. Preprocessing for the text and pattern of the previous example is shown in Fig. 1.

Because the sampled patterns tend to be short, we implemented the search phase with the BMH algorithm [11], which has been found to be fast in such settings [22]. Algorithm 1 shows the pseudo code for the search.

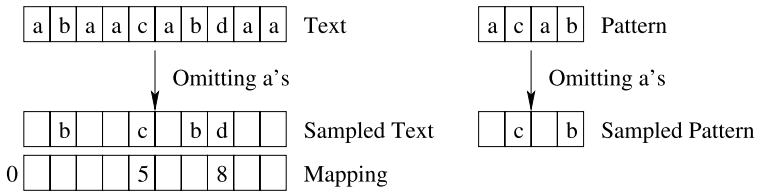

Fig. 1. Example of preprocessing.

Algorithm 1 – Search (P_X, m_X, j_1)

```
for i ← 1 to σ do
    d[i] ← m_X
end for
for i ← 1 to m_X − 1 do
    d[P_X[i]] ← m_X − i
end for
pos ← 1
while pos ⩽ n_X − m_X + 1 do
    j ← m_X
    while j ⩾ 1 and T_X[pos + j − 1] = P_X[j] do
        j ← j − 1
    end while
    if j = 0 then
        if Occurrence in T_X contains a mapped position then
            Check the corresponding position in T for an occurrence
        else
            Check for occurrences starting from M[pos/q] + (pos mod q) − j_1 + 1 to M[pos/q + 1] − (q − (pos mod q)) − j_1 + 1 in T
        end if
    end if
    pos ← pos + d[T_X[pos + m_X − 1]]
end while
```

Although the above scheme works well for most of the patterns, it is obvious that there are some bad patterns which would be searched faster in the original text. We attempt to recognize such patterns as follows. For a given pattern P and text T we estimate the average shift length $S(P,T)$ in the BMH algorithm with the help of the d array[5]:

$$S(P,T) = \sum_{c\in\Sigma} \Pr(c,T)\cdot d[c],$$

where $\Pr(c,T)$ is the empirical probability of occurrence of the character c in text T. We further estimate the average number of characters read $L(P,T)$ by the BMH algorithm in each alignment as

$$L(P,T) = 1 + \sum_{i=2}^{m}\prod_{j=i}^{m}\Pr(p_j,T).$$

The cost of searching in the BMH algorithm can then be estimated as $n\cdot L(P,T)/S(P,T)$. We further estimate the verification cost to be

$$V(P_X,T_X) = C\cdot n_X\cdot\prod_{i=1}^{m_X}\Pr\bigl(P_X[i],T_X\bigr),$$

where C is a constant. The value $C=20$ gave a reasonably good estimate in practice. We can then estimate the total cost of searching in the sampling scheme as

$$W_X = \frac{n_X\cdot L(P_X,T_X)}{S(P_X,T_X)} + V(P_X,T_X). \tag{1}$$

If we just search the original pattern in the original text, we can estimate the cost to be

$$W = \frac{n\cdot L(P,T)}{S(P,T)}. \tag{2}$$

[5] In BMH, $d[c]$ is the distance from the last occurrence of c in P towards the end, or m if c does not appear in P. If c occurs at the end of P we take its next-to-last occurrence. When it is found that P does not occur in a text window, the window is shifted by $d[c]$ where c is the last text character of the window.

If $W_X < W$, we search the sampled text for the sampled pattern and check the occurrences. Otherwise we search the original text for the original pattern.

2.1. Optimal sampling

A question arises from the previous description of our sampling method: How to form the sampled alphabet Σ_X? We will first analyze how the average running time of the BMH algorithm changes when we sample the text and then, based on this, we will develop a method to find the optimal sampled alphabet. Throughout this section we assume that the characters are independently distributed and analyze the approach for a general pattern not known when preprocessing the text. If the pattern were known in advance, we could further optimize the sampling based on the pattern, as done in Eqs. (1) and (2) to choose the text to scan.

Let $\bar{\sigma}$ be defined as the inverse of the probability of two random characters matching, that is, $1/\bar{\sigma} = \sum_{c\in\Sigma} \Pr(c)^2$, where $\Pr(c)$ is the empirical probability of occurrence of character c in T (so $\bar{\sigma} = \sigma$ if all characters are equiprobable, $\Pr(c) = 1/\sigma$ for all c). Let us also define

$$b_A = \sum_{c\in A} \Pr(c), \quad \text{and}$$
$$a_A = \sum_{c\in A} \Pr(c)^2,$$

where $A \subseteq \Sigma$. Now the length of the sampled text will be $b_{\Sigma_X} \cdot n$, the average length of the sampled pattern $b_{\Sigma_X} \cdot m$ (assuming it distributes similarly to the text), and the probability of two random characters matching in the sampled text $a_{\Sigma_X}/b_{\Sigma_X}^2$ (this is $\sum_{c\in\Sigma_X} \Pr'(c)^2$, where $\Pr'(c) = \Pr(c)/b_{\Sigma_X}$ is the probability of character c in T_X). The average complexity of the BMH algorithm is $\mathcal{O}(n(1/m + 1/\bar{\sigma}))$. Thus the average search cost in the sampled text is

$$\mathcal{O}\left(b_{\Sigma_X} \cdot n\left(\frac{1}{b_{\Sigma_X} \cdot m} + \frac{a_{\Sigma_X}}{b_{\Sigma_X}^2}\right)\right) = \mathcal{O}\left(n\left(\frac{1}{m} + \frac{a_{\Sigma_X}}{b_{\Sigma_X}}\right)\right).$$

When considering the verification cost we assume for simplicity that the mapping M contains the position of each sampled character in the original text, that is, $q = 1$. The probability that a position has to be verified is the sum of products of the probabilities of having i symbols samples and matching that i-length string. This corresponds to

$$\begin{aligned}\Pr(\text{verif}) &= \sum_{i=0}^{m} P(|P_X| = i) \cdot P\big(\text{match given that } |P_X| = i\big) \\ &= \sum_{i=0}^{m} \binom{m}{i} b_{\Sigma_X}^i (1 - b_{\Sigma_X})^{m-i} \cdot \left(\frac{a_{\Sigma_X}}{b_{\Sigma_X}^2}\right)^i \\ &= \left(\frac{a_{\Sigma_X}}{b_{\Sigma_X}} + 1 - b_{\Sigma_X}\right)^m.\end{aligned}$$

If we assume that each verification costs $\mathcal{O}(m)$ then the cost of verification is

$$n \cdot \Pr(\text{verif}) \cdot \mathcal{O}(m) = n \cdot \left(\frac{a_{\Sigma_X}}{b_{\Sigma_X}} + 1 - b_{\Sigma_X}\right)^m \cdot \mathcal{O}(m).$$

The total cost of searching in our scheme is thus

$$\mathcal{O}\left(n \cdot \left(\frac{1}{m} + \frac{a_{\Sigma_X}}{b_{\Sigma_X}} + \left(\frac{a_{\Sigma_X}}{b_{\Sigma_X}} + 1 - b_{\Sigma_X}\right)^m \cdot m\right)\right)$$

and hence the optimal sampled alphabet Σ_X minimizes the cost per text character

$$E(\Sigma_X) = \frac{1}{m} + \frac{a_{\Sigma_X}}{b_{\Sigma_X}} + \left(\frac{a_{\Sigma_X}}{b_{\Sigma_X}} + 1 - b_{\Sigma_X}\right)^m \cdot m, \tag{3}$$

which can be divided into the search cost in the sampled text $1/m + a_{\Sigma_X}/b_{\Sigma_X}$ and the verification cost $(a_{\Sigma_X}/b_{\Sigma_X} + 1 - b_{\Sigma_X})^m \cdot m$.

The verification cost always increases when a character is removed from Σ_X, so the search cost in the sampled text must decrease for the combined cost to have a chance to decrease. If $R = \Sigma \setminus \Sigma_X$ is the set of removed characters, then $b_{\Sigma_X} + b_R = 1$ and $a_{\Sigma_X} + a_R = a_\Sigma$, and the function

$$h_R(p) = \frac{1}{m} + \frac{a_{\Sigma_X} - p^2}{b_{\Sigma_X} - p} = \frac{1}{m} + \frac{a_\Sigma - a_R - p^2}{1 - b_R - p}$$

gives the search cost in the sampled text, per text character, if an additional character with probability p is removed. The derivative of $h_R(p)$ is

$$h'_R(p) = 1 - \frac{(1-b_R)^2 - (a_\Sigma - a_R)}{(1-b_R-p)^2}$$

which has exactly one zero at $p_z = (1-b_R) - \sqrt{(1-b_R)^2 - (a_\Sigma - a_R)}$ in the interval $[0, 1-b_R]$. We can see that the function $h_R(p)$ is increasing until p_z and decreasing after that, so removing a character can only be beneficial after h_R goes below $h_R(0)$ again. Solving the equation $h_R(p_R) = h_R(0)$ we get $p_R = (a_\Sigma - a_R)/(1-b_R)$. So removing a single additional character can decrease the search cost in the sampled text only if the probability of occurrence for that character is larger than p_R. Otherwise both the search cost in the sampled text and the verification cost will increase and thus removing the character is not beneficial.

Suppose now that we have already fixed whether we are going to keep or remove each character with probability of occurrence higher than $\Pr(c)$ and now we need to decide if we should remove the character c. If $\Pr(c) > p_R$, we will need to explore both options as removing the character will decrease search cost in the sampled text and increase verification cost. However, if $\Pr(c) < p_R$ we know that if we added only c to R the search time in the sampled text would also increase and therefore we should not remove c. But could it be beneficial to remove c together with a set of other characters with probabilities of occurrence less than p_R? In fact it cannot be. Suppose that we remove a character c with probability $\Pr(c) < p_R$. Now the new removed set will be $R' = R \cup \{c\}$ so we get $a_{R'} = a_R + \Pr(c)^2$ and $b_{R'} = b_R + \Pr(c)$. Now the new critical probability will be

$$p_{R'} = \frac{a_\Sigma - a_{R'}}{1-b_{R'}} = \frac{a_\Sigma - a_R - \Pr(c)^2}{1-b_R-\Pr(c)}.$$

We know that $h_R(\Pr(c)) > h_R(p_R) = h_R(0)$ because $\Pr(c) < p_R$. Therefore

$$\frac{1}{m} + \frac{a_\Sigma - a_R - \Pr(c)^2}{1-b_R-\Pr(c)} > \frac{1}{m} + \frac{a_\Sigma - a_R}{1-b_R}$$

and so

$$p_{R'} = \frac{a_\Sigma - a_R - \Pr(c)^2}{1-b_R-\Pr(c)} > \frac{a_\Sigma - a_R}{1-b_R} = p_R.$$

Thus even now it is not good to remove a character with probability less than the critical value p_R for the previous set and this will again hold if another character with a small probability is removed. Therefore we do not need to consider removing characters with probabilities less than p_R. Note however that removing a character with a higher probability will decrease the critical probability p_R and after this it can be beneficial to remove a previously unbeneficial character. In fact, if the sampled alphabet contains two characters with different probabilities of occurrence, the probability of occurrence for the most frequent character in the sampled alphabet is always larger than p_R. Thus it is always beneficial for searching the sampled text to remove the most frequent character.

The above can be applied to prune the exhaustive search for the optimal set of removed characters. First we sort the characters of the alphabet in decreasing order of frequency. We then figure out if it can be beneficial for searching the sampled text to remove the most frequent character not considered yet. If it can be, we try both removing and not removing that character and proceed recursively for both cases. If it cannot, we prune the search here because none of the remaining characters should be removed. Algorithm 2 gives the pseudo code.

In practice when using this pruning technique the number of examined sets drops drastically as compared to the exhaustive search, although the worst case is still exponential. For example, the number of examined sets drops from 2^{61} to 2826 when considering the King James Bible from the Canterbury Corpus (http://corpus.canterbury.ac.nz/) as the text.

In our experiments, the optimal set of removed characters always contained the most frequent characters up to some limit depending on the length of the pattern, as shown in Table 1. Therefore a simpler heuristic is to remove the k most frequent characters for varying k and choose the k that predicts the best overall time. However, if the verification cost is very high for some reason (e.g., going to disk to retrieve the text, or uncompressing part of it, or using a very sparse sampling) it is possible that the optimal set of removed characters is not a set of most frequent characters.

2.2. Experimental results

To determine the sampled alphabet, we ran the exact algorithm of Section 2.1 for different pattern lengths to choose the sampled alphabet that produces the smallest estimated cost $E(\Sigma_X)$ (Eq. (3)). For all pattern lengths the algorithm recommended removing a set of most frequent characters (that is, to use a set of least frequent characters for the sampling), so we use this simplified method henceforth.

We tested the semi-index approach by removing the k most frequent characters from the text for varying k. We used a 2 MB prefix of the King James Bible as the text, and the patterns are random substrings of the text. For each pattern length 500 patterns were generated, and the reported running times are averages over 200 runs with each of the patterns.

Algorithm 2 – Finding the Optimal Sampling

```
R_opt ← {}
sort Σ in descending order based on the probabilities of occurrence
findOpt(1,{})
return R_opt

findOpt(c, R)
if c = σ + 1 then
    if E(Σ \ R) < E(Σ \ R_opt) then
        R_opt = R
    end if
else
    p_R = (a_Σ − a_R) / (1 − b_R)
    if Pr(c) > p_R then
        findOpt(c + 1, R ∪ {c})
        findOpt(c + 1, R)
    else
        findOpt(σ + 1, R)
    end if
end if
```

Table 1
Predicted and observed optimal number of removed characters for the King James Bible. The predicted optima are computed with the algorithm suggested by the analysis, which in our experiments always returned a set of most frequent characters.

m	10	20	30	40	50	60	70	80	90	100
Predicted optimal number of removed characters	3	7	9	11	12	13	14	15	16	16
Observed optimal number of removed characters	3	8	12	12	14	15	16	17	18	18

The most frequent characters in the decreasing order of frequency were "␣ethaonsirdlfum,wycgbp" where ␣ is the space character. We note that the distribution of characters is not independent in real texts. However, we did not notice any significant changes in the performance that could be due to the non-independence of nearby characters.

The tests in this section were run on a 2.6 GHz AMD Athlon dual core processor with 2 GB of memory, 64 kB L1 cache and 512 kB L2 cache, running Linux 2.6.31. The code is in C and compiled with `gcc` using `-O3` optimization.

We implemented the following versions of our approach:

Basic We always search the sampled text unless the length of the sampled pattern is zero.

Estimated Best Text We estimate the cost of searching using the sampled text (W_X, Eq. (1)) or the original text (W, Eq. (2)). We choose the text with smaller estimated cost.

Optimal Text For each pattern we search both the sampled and the original text, and pick the smaller runtime. This version only serves to give a lower bound to the performance of the previous estimation method, given the alphabet partition.

Fig. 2 shows the runtime of the three above versions using the mapping array M where every 8th or 32nd sampled character is mapped to its position in the original text. The results for zero removed characters correspond to the original BMH algorithm. The figure shows that making the mapping sparser increases the runtime as more characters are removed because verification is more costly and a larger amount of verification is needed. The effect is less noticeable for longer patterns, where the sampling method is up to 5 times faster than sequentially scanning the original text.

Fig. 2 also shows that choosing the text with lower estimated cost drastically improves upon the performance of the basic method, especially as more characters are removed. We further see that the difference between our improved method and optimally choosing the text to search is extremely small (basically indistinguishable in the plots).

Furthermore, we notice that the optimal number of removed characters (in both the optimal or our improved method) grows slowly with m. Table 1 compares these optimal values with those given by the analysis. The observed optimal values are given for the basic version with mapping density $q = 8$. As we can see, the analysis gives reasonably good results although it recommends removing slightly fewer characters, because we estimated the verification time quite pessimistically: When more characters are removed it is unlikely that we would need to scan m characters for each verified position (as verifications can stop as soon as the first mismatch is encountered). We tried more sophisticated ways of estimating the cost of a single verification but these were not much better.

Nevertheless, the effect of this error is negligible: The curves are sufficiently smooth so that using a value close to the optimal one makes little difference. This is good also because we have to make this decision at text preprocessing time, for all future pattern lengths. For example, by choosing to remove the 13 most frequent characters, the estimated best text version would do reasonably well for all pattern lengths using just 0.18 times the original text size to store the sampled text. To this we have to add one integer each q symbols for the mapping, and σ bits to describe Σ_X.

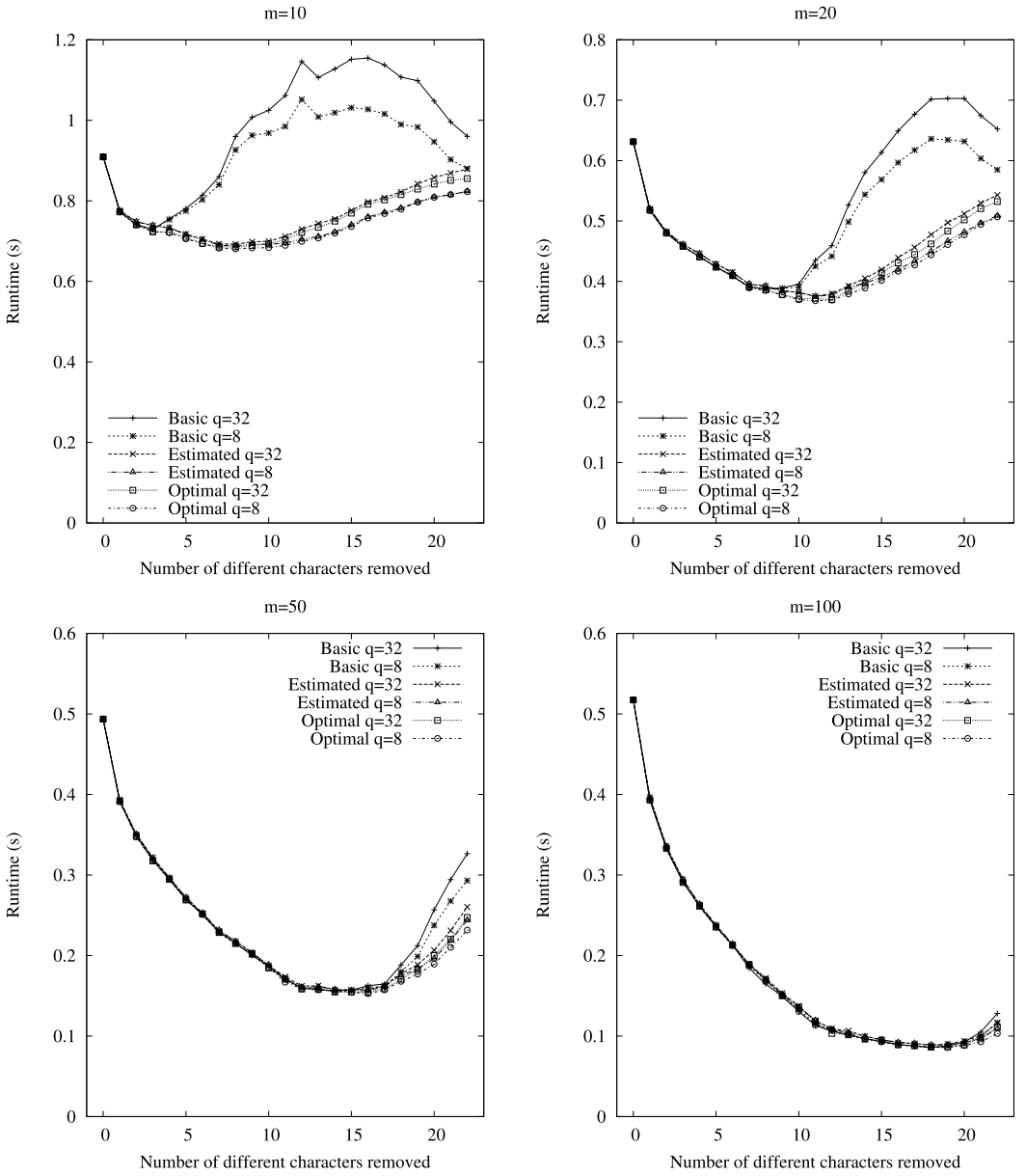

Fig. 2. The running time of the sampling approach for pattern lengths 10, 20, 50, and 100.

3. Succinct alphabet sampling

An alternative to storing a mapping M is to use a bitmap B of length n, where we mark with a one every position in T where the symbol belongs to Σ_X. We index this bitmap in order to support select queries in constant time and $o(n)$ extra bits [4,19]: $\text{select}_B(b, j)$ finds the position where the j-th bit b occurs in B. In practice we use an implementation [9] requiring just 5% of extra space over the bare bitmap to compute select_B.

This bitmap is used for verification as follows. Suppose that we have found the sampled pattern P_X in the sampled text T_X at position k. The position of the character $T_X[k]$ in the original text is now $i = \text{select}_B(1, k)$. If j_1 is the position of $P_X[1]$ in the original pattern P, we verify the position $i - j_1 + 1$ in the original text for an occurrence of the whole pattern.

Bitmap B requires 1.05 bits per text symbol, which is 14% extra space when symbols are stored in bytes. This is close to the space required by mapping M for $q = 4$. However, it turns out that this bitmap gives us sufficient information to reduce space and become a much succincter alternative than using M for any q: Up to now we store T and T_X is redundant. With bitmap B we can instead store two texts T_X and T_Y, containing the characters in Σ_X and $\Sigma_Y = \Sigma - \Sigma_X$, so that bitmap B is the only redundant information (which suffices to reconstruct T from T_X and T_Y; this would not be possible with

table M unless $q = 1$). So this 14% extra space is already less than the 18% extra that was used for storing T_X under the well-chosen alphabet size $|\Sigma_X| = 13$.

In a more formal statement, consider two sets Σ_X, Σ_Y, such that $\Sigma_X \cap \Sigma_Y = \emptyset$ and $\Sigma_X \cup \Sigma_Y = \Sigma$. We represent both Σ_X and Σ_Y with a single bitmap of length σ. Now given text T of length n, drawn from alphabet $\Sigma = [1, \sigma]$, we define bitmap B, of length n, such that $B[i] = 1$ iff $T[i] \in \Sigma_X$. We index B for select_B queries. We also create $T_X = t_{i_1} t_{i_2} \dots t_{i_{n_X}}$ and $T_Y = t_{j_1} t_{j_2} \dots t_{j_{n_Y}}$, where the i_ks correspond to the increasing positions where $B[i_k] = 1$ and similarly the j_ks for positions where $B[j_k] = 0$.

It is clear that the sequences B, T_X, T_Y suffice for representing T. Let $\text{rank}_B(b, i)$ be the number of occurrences of bit b in $B[1, i]$, that is, a kind of inverse of select_B that can also be computed in constant time within the same space and time of select_B. Then it holds

$$T[i] = \begin{cases} T_X[\text{rank}_B(1, i)] & \text{if } B[i] = 1, \\ T_Y[\text{rank}_B(0, i)] & \text{otherwise} \end{cases}$$

which allows reconstructing any desired substring of T in optimal time.

Since $n_X + n_Y = n$, the extra space for our representation is only $\sigma + n + o(n)$ bits. Note that, unlike the scheme based on M, the extra space is independent of how we partition the alphabet and of any sampling density.

We now search for a pattern P of length m as follows. First, we partition P into P_X and P_Y, and create a bitmap B_P of length m such that $B_P[i] = 1$ iff $P[i] \in \Sigma_X$. Then we choose whether to search for P_X in T_X or for P_Y in T_Y, using the estimation W_X of Eq. (1) and its analogous version W_Y. Finally, we verify every match in T_X or T_Y as follows. Suppose that we have a candidate match in T_X at position k. The position of $T_X[k]$ in the original text is $i = \text{select}_B(1, k)$ and thus the potential match starts at position $i - j_1 + 1$ where j_1 is the position of the first character of P_X in P. We then verify that $B[i - j_1 + 1 \dots i - j_1 + m]$ matches bitmap $B_P[1, m]$. If the bits match, we further verify that P_Y occurs in T_Y at position $i - j_1 + 2 - k$. The process is analogous if we search T_Y, now using $\text{select}_B(0, k)$ to map to T.

3.1. Experimental results

The same experimental setting as for alphabet sampling was used to test succinct alphabet sampling. We implemented two versions of succinct alphabet sampling:

Estimated Best Text We estimate the cost of searching in T_X and T_Y. We choose the text with smaller estimated cost.
Optimal Text We search for each pattern in text T_X and in text T_Y. We pick the smaller runtime for each pattern. This version only serves to give a lower bound to the performance of the previous estimation method.

Fig. 3 shows a comparison of alphabet sampling and succinct alphabet sampling. The figure also contains a version of alphabet sampling where bit vector B is used instead of M, but still the scheme of storing and searching T_X and/or T is maintained. We see that the alphabet sampling approach with mapping M is somewhat slower than the other approaches for short patterns, which require more verifications to be performed. Alphabet sampling using bit vector B is as fast as succinct alphabet sampling.

Fig. 4 (left) shows the memory usage for the various approaches. We can see that succinct alphabet sampling uses less memory than alphabet sampling in most cases. Only when 18 or more characters are removed, alphabet sampling using mapping M is slightly more space-efficient, but removing that many characters is time-efficient only for long patterns ($m = 100$). On the right, the figure shows the runtime of the versions that search the best estimated text as a function of the pattern length when 13 most frequent characters are removed from the alphabet to form the sampled alphabet.

To test our method on data where the distribution of characters is more uniform than in natural language texts, we ran experiments on a 2 MB prefix of the protein data from *PizzaChili* site, http://pizzachili.dcc.uchile.cl. Again we generated 500 patterns and report the averages of 200 runs with each of the patterns. This protein file has an alphabet of size 23 including 20 characters for the various amino acids, 'X' denoting any amino acid, 'Z' denoting either glutamine or glutamic acid, and newline for separating the different proteins. The most frequent characters were "LSAGEPVTRKQDIFNYHMCW" in decreasing order of frequency. Fig. 5 shows the results of these experiments. We see that in this case the speedup of our method is smaller than for English data: For patterns of length 100 we are up to two times faster than when sequentially scanning the text, while for English data we were five times faster. We notice that choosing which text to search based on estimating the cost does not work as well as with English data as for pattern lengths 50 and 100 the basic version performs better than alphabet sampling choosing the text based on estimated cost.

It is interesting that succinct alphabet sampling actually reduces the search times by up to 40% even on truly random data, as shown in Fig. 6. We also note that choosing the best text to search for based on the estimated cost works even worse than for protein data, as the basic version outperforms the estimated version for alphabet sampling on long patterns. Still succinct alphabet sampling performs better.

4. Sampled suffix array

To turn the sampling approach into an index, we use a suffix array to index the sampled positions of the text. When constructing the suffix array, only suffixes starting with a sampled character will be considered, but the sorting will still

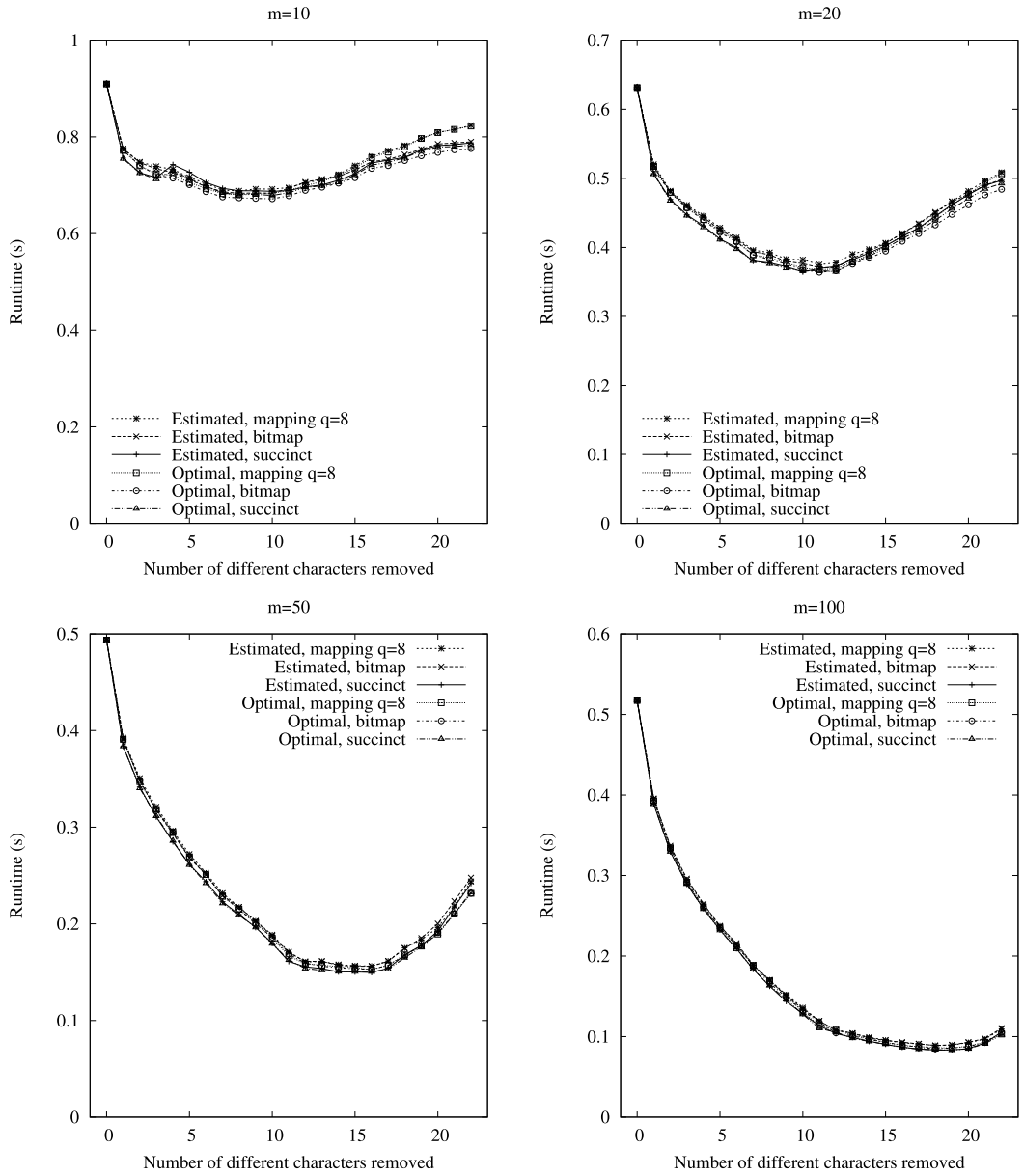

Fig. 3. The running time of alphabet sampling and succinct alphabet sampling for pattern lengths 10, 20, 50, and 100.

be done considering the full suffixes. The resulting sampled suffix array is like the suffix array of the original text where suffixes starting with unsampled characters have been omitted.

The construction of the sampled suffix array can be done in $\mathcal{O}(n)$ time using $\mathcal{O}(n_X)$ words of space if we apply the construction technique of the word suffix array [6]. The sampled suffix array for the text $T = abaacabdaa$ is shown in Fig. 7, where the sampled alphabet is $\Sigma_X = \{b, c, d\}$.

Search on the sampled suffix array is carried out as follows. Given a pattern $P = p_1 \dots p_m$ we first find the first sampled character of the pattern. Let this be at index j. The pattern is now divided into the unsampled prefix $p_1 \dots p_{j-1}$ and the suffix starting with the first sampled character $p_j \dots p_m$. We search the sampled suffix array for this suffix of the pattern like in an ordinary suffix array. Each candidate match returned by this search will then be verified by comparing the unsampled prefix against the text.

We could also construct the suffix array directly for T_X, and search for P_X, but this would entail more verifications as the unsampled characters of the pattern suffix would not be required to match. We would also need to store the sampled text, or to skip the unsampled characters in the original text each time we read a suffix. For the same reasons using a scheme splitting into T_X and T_Y does not make sense in this scenario.

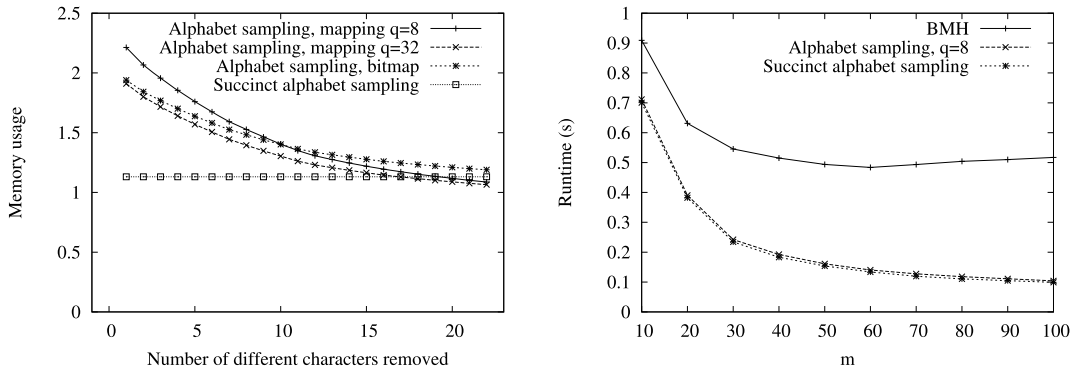

Fig. 4. On the left, memory usage for the various semi-indexes, reported as size of the index (including the text) divided by the size of the text. On the right, the running time of the original BMH algorithm and alphabet sampling as a function of the pattern length when the 13 most frequent characters are removed to form the sampled alphabet.

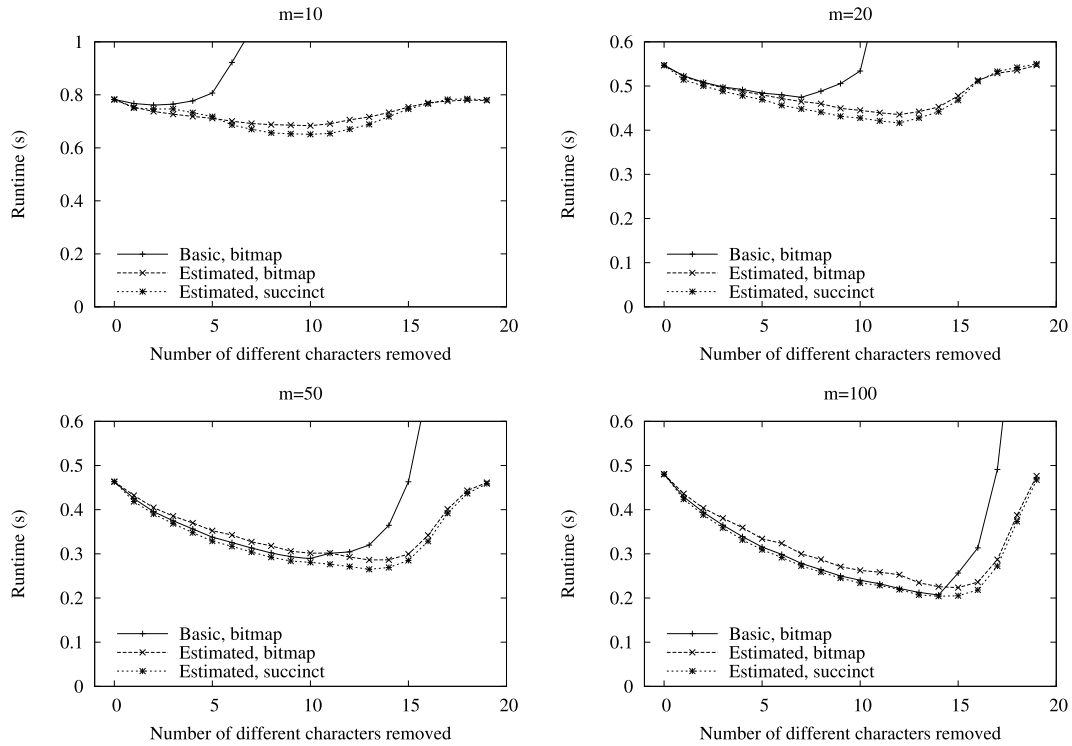

Fig. 5. The running time of alphabet sampling and succinct alphabet sampling for protein data with pattern lengths 10, 20, 50, and 100.

The sampled suffix array resembles a sparse suffix array [12], which indexes regularly sampled text positions. However, we only need to make one search on the sampled suffix array, while using a sparse suffix array one needs to make q searches if the sparse suffix array indexes every q-th position. On the other hand, the sampled suffix array can only be used for patterns that contain at least one sampled character, whereas the sparse suffix array can be used if the pattern length is at least q. The variance of the search time when using the sampled suffix array is also larger than when using a sparse suffix array because in the sampled suffix array we have much less control over the length of the string that is used in the suffix array search.

4.1. Optimal sampling

Suppose that we have enough space to create the sampled suffix array for $b \cdot n$ suffixes where $0 < b < 1$. How should we now choose the sampled alphabet Σ_X so that the search time would be optimal? Obviously $b_{\Sigma_X} \approx b$, but we still have a number of possible sampled alphabets to choose from. The search on the suffix array will compare the suffix of the pattern starting with the first sampled character against a text string $\mathcal{O}(\log n)$ times. The comparison time is minimized when the

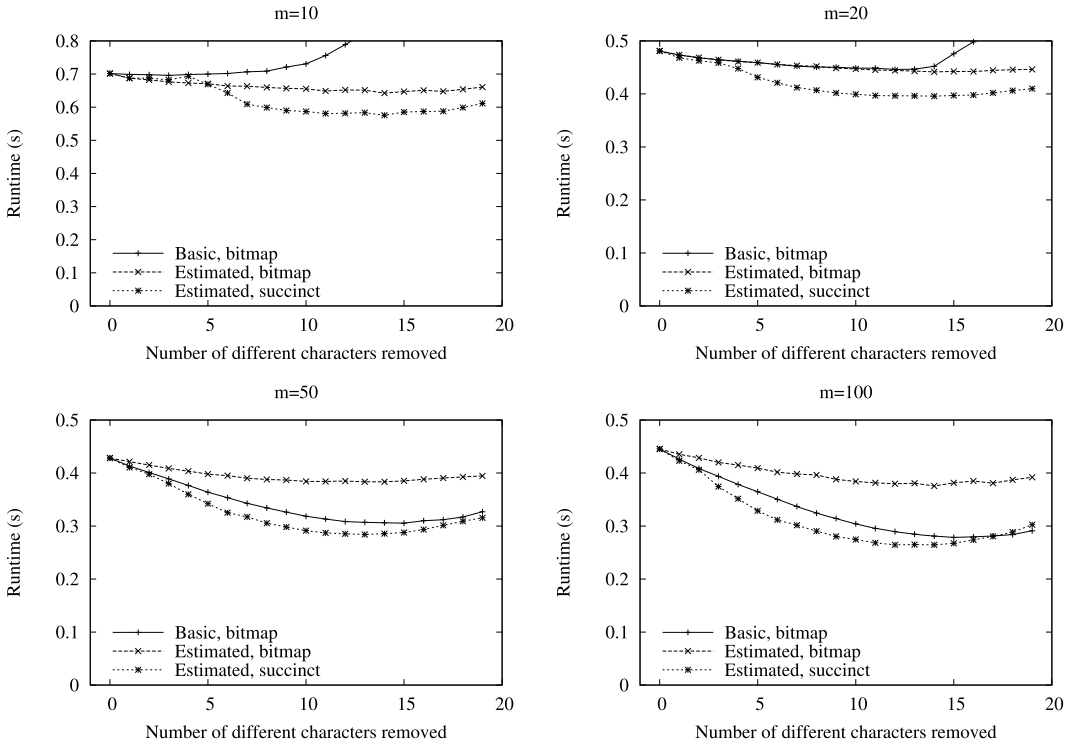

Fig. 6. The running time of alphabet sampling and succinct alphabet sampling for random data of alphabet size 26 with pattern lengths 10, 20, 50, and 100.

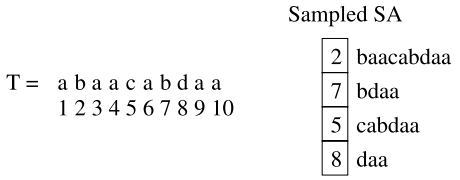

Fig. 7. The sampled suffix array for the text $T = abaacabdaa$ with the sampled alphabet $\Sigma_X = \{b, c, d\}$. The sorted suffixes are only shown for convenience. They are not part of the structure.

probability of matching for the first sampled character is minimized (the fate of the rest of the comparison is independent of the choice of Σ_X). Thus the sampled alphabet Σ_X should be a set of least frequent characters.

Let us then consider the verification. The probability that two random characters are unsampled and match is $a_R = a_\Sigma - a_{\Sigma_X}$ where R is the set of removed characters. Thus the average cost of a single verification is $1/(1 - a_\Sigma + a_{\Sigma_X})$.

The probability that the suffix of the pattern starting with the first sampled character matches a random string of equal length is

$$\begin{aligned}&\Pr(\text{1st char of string is sampled})\\ &\quad\cdot \Pr(\text{1st char matches given that it is sampled})\\ &\quad\cdot \Pr(\text{rest matches})\\ &= b_{\Sigma_X} \cdot \frac{a_{\Sigma_X}}{b_{\Sigma_X}^2} \cdot (a_\Sigma)^{m_s-1} = \frac{a_{\Sigma_X}}{b_{\Sigma_X}} (a_\Sigma)^{m_s-1}\end{aligned}$$

where m_s is the length of the suffix starting with the first sampled character. This is also the probability of verification per character in the original text. The average cost of verification per text character is then

$$\frac{a_{\Sigma_X}}{b_{\Sigma_X}} (a_\Sigma)^{m_s-1} \cdot \frac{1}{1 - a_\Sigma + a_{\Sigma_X}} = \frac{a_{\Sigma_X}}{1 - a_\Sigma + a_{\Sigma_X}} \cdot \frac{(a_\Sigma)^{m_s-1}}{b_{\Sigma_X}}.$$

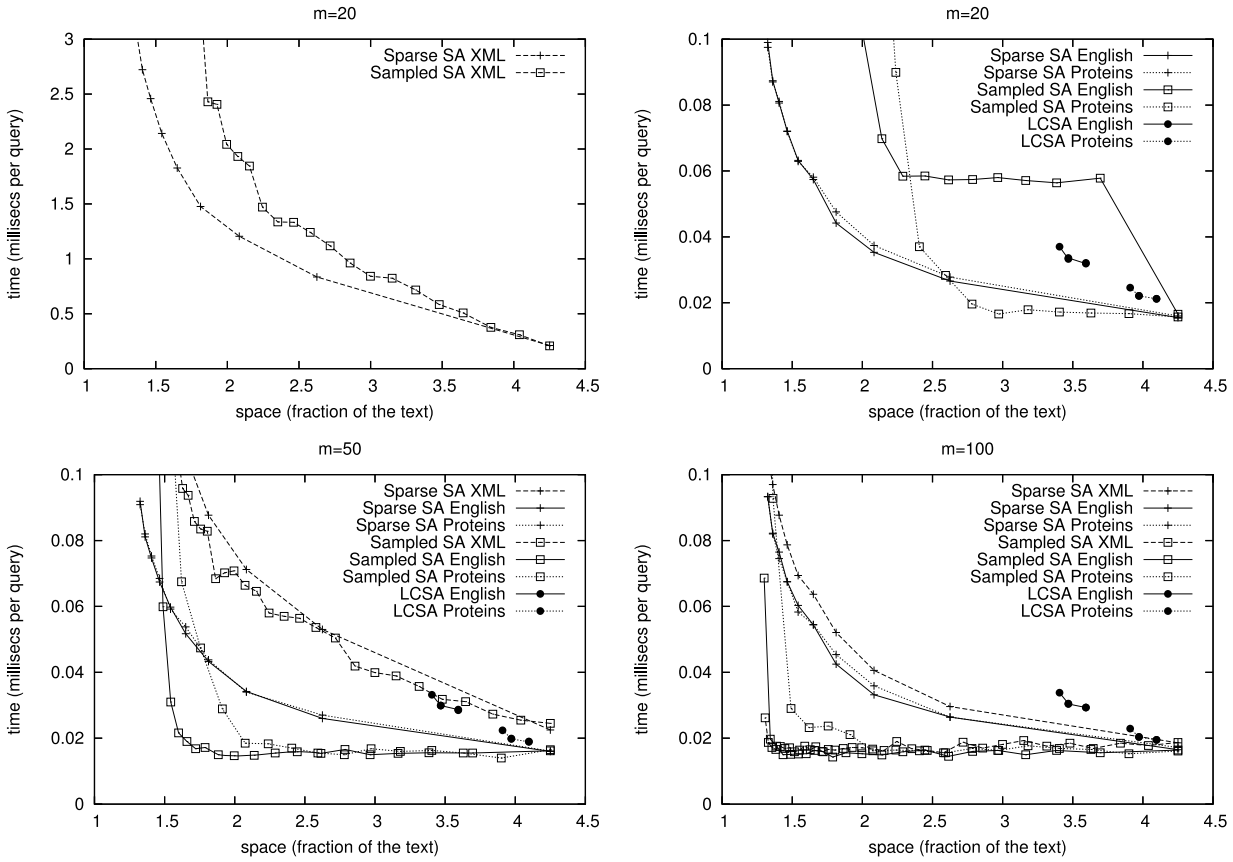

Fig. 8. Search times for the sampled and sparse suffix arrays and LCSA for XML, English and protein data. LCSA uses little space for XML data but it is much slower than the other approaches, so these results are not shown. The top figures show results for pattern length 20 and the bottom figures show the results for pattern lengths 50 and 100. The space fraction includes that of the text, so it is of the form $1 + \frac{\text{index size}}{\text{text size}}$.

Because we attempt to determine the optimal sampled alphabet such that $b_{\Sigma_X} = b$, b_{Σ_X} and the distribution of m_s do not depend on which characters we remove. Thus we should minimize $f(a_{\Sigma_X}) = a_{\Sigma_X}/(1 - a_\Sigma + a_{\Sigma_X})$. The derivative of $f(a_{\Sigma_X})$ is

$$f'(a_{\Sigma_X}) = \frac{1 - a_\Sigma}{(1 - a_\Sigma + a_{\Sigma_X})^2} > 0$$

so the verification cost increases when a_{Σ_X} increases. (Thus the best in terms of time is to use the original suffix array, with $b_{\Sigma_X} = a_{\Sigma_X} = 0$; we are fixing b_{Σ_X} at a threshold b with the aim of reducing space, not time.) To minimize a_{Σ_X} with fixed b_{Σ_X} the sampled alphabet Σ_X should be a set of least frequent characters. This also minimizes the total cost because also the suffix array search cost is minimized by this choice. Interestingly, this corresponds to the simplified heuristic we proposed in Section 2.1.

4.2. Experimental results

Fig. 8 shows the results obtained by comparing our sampled suffix array against our implementation of the sparse suffix array [12] and the locally compressed suffix array (LCSA) [10], an index that compresses the differential suffix array using grammar-based compression. Note that when the space usage of the sampled or sparse suffix array is maximal (3.25 times the text) both of them index all suffixes and behave exactly like a normal suffix array.

The experiments in this section were run on a Pentium IV 2.0 GHz processor with 2 GB of RAM running SuSE Linux with kernel 2.4.31. The code was compiled using `gcc` version 3.3.6 with `-O9` optimization. We used 50 MB English, protein, and XML texts from the *PizzaChili* site.

Our approach performs very well for moderate to long patterns. It is superseded by the sparse suffix array up to $m = 50$, from where it starts to dominate the other alternatives. For $m = 100$ the sampled suffix array behaves almost like a suffix array (and much faster than the other methods), even when using less than 0.5 times the text size (plus text).

The novel compressed self-indexes [7,21] are designed to use much less space (e.g., 0.8 times the text size including the text) but take much more time at reporting the occurrences, and thus are inappropriate for this comparison. We chose the

LCSA as an alternative that compresses less but is much faster than the other self-indexes [10]. Its compression performance varies widely with the text type, and is not particularly good on English and proteins. On XML it requires extra space equal to the size of the text, yet its times are much higher and fall well outside the plot (and this is still much faster than the other self-indexes!). The LCSA, on the other hand, would perform better on shorter patterns, where our index is not competitive.

5. Conclusions

We have presented two approaches to speed up string matching with moderate to long patterns. Both are based on creating a sample of the text by choosing some characters of the alphabet. A first approach is a *semi-index* where the search in the sampled text is sequential. The second approach builds a suffix array on the sampled text.

The semi-index profits from nonuniform character distribution to gain a speedup of up to 5 times over online searching, at the price of 1.05 extra bits per symbol, that is, around 14% extra space. The sampled suffix array works also with a uniform distribution, and matches the performance of the full suffix array while using just 1/8 of its space, that is, 50% of space overhead over the text size.

It is worth noting that in the semi-index approach the text to search is an internal structure of the semi-index so any transform, like compression or code splitting [24], could be applied to it.

In particular, we could easily compress the text of the sampled suffix array in order to reduce the total space and make it more competitive with compressed self-indexes. Consider applying Hu–Tucker (or Garsia–Wachs) compression to the text [13]. This is similar to Huffman coding but it preserves the lexicographic order of the code, and therefore the comparisons carried for searching on the suffix array can be made directly between the compressed pattern and the compressed text [17]. This compression achieves at worst one bit over the file size achieved by Huffman compression. Using Hu–Tucker, we would even improve our search times (as there are fewer bytes to compare on average to decide each comparison,[6]) and still compress the text to its zero-order entropy plus at most 2 bits. On our 50 MB English file, we would compress the text to 4.8 bits per character. As now the suffix array points to bit offsets, pointers use two more bits (28 instead of 26) in our example. Considering a sampled suffix array using 1/8 of the pointers, which was rather efficient in our experiments, the overall space is 8.3 bits per character, that is, almost the same size of the original file. Note this size includes the compressed text and its sampled suffix array.

The current approach is not applicable to small alphabets. To extend the approach to that case we could use q-grams. In the semi-index approach we would then define a sampled alphabet for each $(q-1)$-long context and the sampled text would contain those characters that are sampled in the context where they occur. When searching for a pattern, we must always discard the first $q-1$ characters of the pattern as their context is not known. Using q-grams with the sampled suffix array is simpler. The sampled suffix array would just index all suffixes starting with a sampled q-gram.

Another case where our method does not apply is that of short patterns. For the sampled semi-index one can use any pattern matching algorithm well suited for this task [8,22], directly on T. For the succinct alphabet sampling, where the text is partitioned into T_X and T_Y, we could sequentially search the bitmap B that describes the partitioning of the text, for the bitmap B_P that describes the partitioning of the pattern, and both T_X and T_Y would be used for verification (via rank queries on B to find the proper locations). The use of a binary alphabet would make it possible to search for short and moderate patterns quite efficiently. For example, one can preprocess every $w = \frac{\log_2 n}{2}$-long bitmap so that one can run a KMP-like [14] algorithm on B, yet advancing by chunks of w bits: We build a table of $\sqrt{n} \cdot m$ entries, within $O(m\sqrt{n})$ time. Each entry corresponds to a w-bit chunk and a KMP state, and tells the new KMP state at the end of the chunk, listing also the occurrences found within the chunk.

Another interesting direction of future work for the succinct semi-index is to partition the alphabet into more than two partitions. We would then keep a text for each alphabet partition and replace the bitmap by a sequence indicating the partition of each character in the text. When searching we could scan any of the texts and use the sequence and other texts for verification.

References

[1] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, J. Discrete Algorithms 2 (1) (2004) 53–86.
[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman, Basic local alignment search tool, J. Mol. Biol. 215 (3) (1990) 403–410.
[3] A. Apostolico, The myriad virtues of suffix trees, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, in: NATO Advanced Science Institutes, Series F, vol. 12, Springer-Verlag, 1985, pp. 85–96.
[4] D. Clark, Compact pat trees, Ph.D. thesis, University of Waterloo, 1996.
[5] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, Speeding up two string-matching algorithms, Algorithmica 12 (4) (1994) 247–267.
[6] P. Ferragina, J. Fischer, Suffix arrays on words, in: Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM), in: LNCS, vol. 4580, 2007, pp. 328–339.

[6] The compressed pattern must be padded to the next byte boundary by adding zero bits when searching for the left suffix array interval, and one bits when searching for the right interval.

[7] P. Ferragina, R. González, G. Navarro, R. Venturini, Compressed text indexes: From theory to practice, ACM J. Exp. Algorithmics 13 (2009), article 12, 30 pages.
[8] K. Fredriksson, S. Grabowski, Practical and optimal string matching, in: 12th International Conference on String Processing and Information Retrieval (SPIRE), in: LNCS, vol. 3772, 2005, pp. 376–387.
[9] R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical implementation of rank and select queries, in: Poster Proc. 4th Workshop on Efficient and Experimental Algorithms (WEA), 2005, pp. 27–38.
[10] R. González, G. Navarro, Compressed text indexes with fast locate, in: Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM), in: LNCS, vol. 4580, 2007, pp. 216–227.
[11] R.N. Horspool, Practical fast searching in strings, Software – Practice & Experience 10 (6) (1980) 501–506.
[12] J. Kärkkäinen, E. Ukkonen, Sparse suffix trees, in: Proc. 2nd Annual International Conference on Computing and Combinatorics (COCOON), in: LNCS, vol. 1090, 1996, pp. 219–230.
[13] D.E. Knuth, The Art of Computer Programming: Sorting and Searching, vol. 3, 2nd edition, Addison-Wesley, 1998.
[14] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.
[15] U. Manber, G. Myers, Suffix arrays: A new method for online string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[16] U. Manber, S. Wu, GLIMPSE: A tool to search through entire file systems, in: Proc. USENIX Technical Conference, USENIX Association, Berkeley, CA, USA, Winter 1994, pp. 23–32.
[17] E. Moura, G. Navarro, N. Ziviani, Indexing compressed text, in: Proc. 4th South American Workshop on String Processing (WSP), Carleton University Press, 1997, pp. 95–111.
[18] E. Moura, G. Navarro, N. Ziviani, R. Baeza-Yates, Fast and flexible word searching on compressed text, ACM T. Inform. Syst. 18 (2) (2000) 113–139.
[19] I. Munro, Tables, in: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), in: LNCS, vol. 1180, 1996, pp. 37–42.
[20] G. Navarro, R. Baeza-Yates, E. Sutinen, J. Tarhio, Indexing methods for approximate string matching, IEEE Data Engrg. Bull. 24 (4) (2001) 19–27.
[21] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007) 1–61.
[22] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings – Practical On-line Search Algorithms for Texts and Biological Sequences, Cambridge University Press, 2002.
[23] S. Puglisi, W. Smyth, A. Turpin, Inverted files versus suffix arrays for locating patterns in primary memory, in: Proc. 13th International Conference on String Processing and Information Retrieval (SPIRE), in: LNCS, vol. 4029, 2006, pp. 122–133.
[24] J. Rautio, J. Tanninen, J. Tarhio, String matching with stopper encoding and code splitting, in: Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM), in: LNCS, vol. 2373, 2002, pp. 45–52.
[25] A.C. Yao, The complexity of pattern matching for a random string, SIAM J. Comput. 8 (3) (1979) 368–387.