Speeding Up Pattern Matching by Text Sampling

Francisco Claude[1,⋆], Gonzalo Navarro[1,⋆], Hannu Peltola[2], Leena Salmela[2], and Jorma Tarhio[2]

[1] Department of Computer Science, University of Chile
{fclaude, gnavarro}@dcc.uchile.cl
[2] Department of Computer Science and Engineering
Helsinki University of Technology
{hpeltola, lsalmela, tarhio}@cs.hut.fi

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 select the corresponding subsequence of the text. Online or indexed searching is then carried out on that 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. For indexed searching we achieve indexes that are as fast as the classical suffix array, yet occupy space less than 0.5 times the text size (instead of 4) plus text. Our experiments show no competitive alternatives in a wide space/time range.

1 Introduction

The string matching problem is to find all the occurrences of a given pattern $P = p_0 p_1 \ldots p_{m-1}$ in a large text $T = t_0 t_1 \ldots t_{n-1}$, both being sequences of characters drawn from an alphabet Σ of size σ.

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 $\mathcal{O}(n \cdot f(m))$. The worst-case complexity of the problem is $\Theta(n)$, first achieved by the Knuth-Morris-Pratt algorithm [9]. The average complexity of the problem is $\Theta(n \log_\sigma m/m)$, achieved for example by the BDM algorithm [3]. Other non-optimal algorithms such as the Boyer-Moore-Horspool (BMH) algorithm [7] 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 $\mathcal{O}(m \cdot g(n) + occ \cdot h(n))$ time, where occ is the number of occurrences of the pattern in the text. Popular solutions to this approach are suffix trees and suffix arrays [10]. 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 $\mathcal{O}(m + occ)$ using extra space [1]. The problem of these approaches is that the space needed is too large for many practical situations (4–20 times the text size). Recently, a

⋆ Partially funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile

lot of effort has been spent to compress these indexes [13] obtaining a significant reduction in space, but requiring considerable implementation effort [5].

In this work we explore sampling the text by removing a set of characters from the alphabet. We first apply an online algorithm to this sampled text, obtaining an approach in between online searching 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 its search complexity is still of the form $\mathcal{O}(n \cdot f(m))$. To be interesting, a semi-index should be easy to implement and require little extra space. Several other semi-indexes exist in the literature, even without using that name. For example, q-gram indexes [12], directly searchable compression formats [11], 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 [8] as both index a subset of the suffixes, but the different sampling properties induce rather different search algorithms.

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. For indexed searching we achieve indexes that are as fast as the classical suffix array, yet occupy less than 0.5 times the text size (instead of 4) plus text.

2 Text Sampling

The main idea of our online 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. At regular intervals we map the positions of the sampled text to their corresponding positions in the original text. When searching, we build the sampled pattern from the pattern by omitting all characters not in the sampled alphabet and then search for this sampled pattern in the sampled text. For each candidate returned by this search we verify a short range of the original text with the help of the position mapping.

Let $T = t_0 t_1 \ldots t_{n-1}$ be the text over the alphabet Σ and $\tilde{\Sigma} \subset \Sigma$ the sampled alphabet. The proposed semi-index is composed of the following items:

– Sampled text $\tilde{T}$: Let $\tilde{T} = t_{i_0} t_{i_1} \ldots t_{i_{\tilde{n}-1}}$ be the sequence of the t_i's that belong to the sampled alphabet $\tilde{\Sigma}$. The length of the sampled text is $\tilde{n}$.
– The position mapping M: A table of size $\lfloor \tilde{n}/q \rfloor$ where $M[i]$ maps the $q \cdot i$'th character of $\tilde{T}$ to its corresponding position in T so $\tilde{T}[q \cdot i] = T[M[i]]$.

Given a pattern $P = p_0 p_1 \ldots p_{m-1}$, search on this semi-index is carried out as follows. Let $\tilde{P} = p_{j_0} p_{j_1} \ldots p_{j_{\tilde{m}-1}}$ be the subsequence of p_i's that belong to the sampled alphabet $\tilde{\Sigma}$. The length of the sampled pattern is thus $\tilde{m}$. The sampled

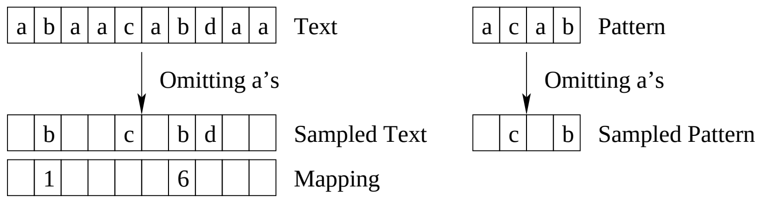


Fig. 1. Example of preprocessing

```
search ($\tilde{T} = \tilde{t}_0\tilde{t}_1 \dots \tilde{t}_{\tilde{n}-1}, \tilde{P} = \tilde{p}_0\tilde{p}_1 \dots \tilde{p}_{\tilde{m}-1}, T = t_0t_1 \dots t_{n-1}$,
        $P = p_0p_1 \dots p_{m-1}, j_0, q, M[0 \dots \tilde{n}/q]$)
1.  for ($i \leftarrow 0$ to $\sigma - 1$) $d[i] \leftarrow \tilde{m}$
2.  for ($i \leftarrow 0$ to $\tilde{m} - 2$) $d[\tilde{p}_i] \leftarrow \tilde{m} - 1 - i$
3.  $pos \leftarrow 0$
4.  while ($pos < \tilde{n} - \tilde{m}$)
5.      $j \leftarrow \tilde{m} - 1$
6.      while ($j \geq 0$ and $\tilde{t}_{pos+j} = \tilde{p}_j$) $j \leftarrow j - 1$
7.      if ($j = -1$)
8.          Check for occurrence from $M[pos/q] + (pos \bmod q) - j_0$
9.            to $M[pos/q + 1] - (q - pos \bmod q) - j_0$
10.     $pos \leftarrow pos + d[\tilde{t}_{pos+\tilde{m}-1}]$
```

Fig. 2. Searching the sampled text for a sampled pattern with the BMH algorithm

text $\tilde{T}$ is then searched for $\tilde{P}$, and for every occurrence, the positions to check in the original text are delimited by the position mapping M. If the sampled pattern is found in position i_r in $\tilde{T}$, the area $T[M[i_r/q] + (i_r \bmod q) - j_0 \dots M[i_r/q + 1] - (q - i_r \bmod q) - j_0]$ is checked for possible startings of real occurrences.

For example, if the text is $T = abaacabdaa$, the sampled text built omitting the a's ($\tilde{\Sigma} = \{b, c, d\}$) is $\tilde{T} = t_1t_4t_6t_7 = bcbd$. If we map every other position in the sampled text, the position mapping M is $\{1, 6\}$. For searching the pattern $acab$ we omit the a's and get $\tilde{P} = p_1p_3 = cb$. We search for $\tilde{P} = cb$ in $\tilde{T} = bcbd$, finding an occurrence at position 1. The previous mapped position is $M[0] = 1$, so $\tilde{t}_0$ corresponds to t_1, and the next mapped position is $M[1] = 6$, so $\tilde{t}_2$ corresponds to t_6. Because the first sampled character in P is in position 1, we verify the area $1 \dots 4$ in the original text finding the match at position 3. Preprocessing for the text and pattern of the previous example is shown in Fig. 1.

Because the sampled patterns tend to be quite short, we implemented the search phase with the BMH algorithm [7], which has been found to be fast in such settings [14]. Figure 2 shows the algorithm for this basic method.

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. The average complexity of the BMH algorithm is $\mathcal{O}(n(1/m + 1/\sigma)) = \mathcal{O}(n/\min(m, \sigma))$ assuming a uniform and independent distribution of the characters of the alphabet [2]. If the distribution is not uniform, a better approximation is to replace σ by the the effective alphabet size $\bar{\sigma}$, which is defined as the inverse of the probability of two random characters matching, i.e. $1/\bar{\sigma} = \sum_{c \in \Sigma} p_c^2$, where p_c is the empirical probability of occurrence of the character c.

To determine if it would be faster to just search the pattern in the original text we tried calculating the ratios $n/\min(m, \bar{\sigma})$ and $n \cdot (1/m + 1/\bar{\sigma})$ both for the sampled text and pattern and for the original text and pattern. If the ratio is lower for the original text and pattern, we search only in the original text. The results were better using the ratio $n/\min(m, \bar{\sigma})$.

3 Optimal Sampling for Online Search

A question arises from the previous description of our sampling method: How to form the sampled alphabet $\tilde{\Sigma}$? 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 independent and we analyze the approach for a general pattern not known when preprocessing the text.

Let us define $b_A = \sum_{c \in A} p_c$ and $a_A = \sum_{c \in A} p_c^2$ where $A \subset \Sigma$. Now the length of the sampled text will be $b_{\tilde{\Sigma}} n$, the average length of the sampled pattern $b_{\tilde{\Sigma}} m$ (assuming it distributes similarly to the text) and the probability of two random characters matching in the sampled text $a_{\tilde{\Sigma}}/b_{\tilde{\Sigma}}^2$. Given the average complexity of the BMH algorithm, $\mathcal{O}(n(1/m + 1/\bar{\sigma}))$, the average search cost in the sampled text is

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

When considering the verification cost we assume for simplicity that the mapping M contains the position of each sampled character in the original text, i.e. $q = 1$. The probability that a position has to be verified is then

$$p_{\text{ver}} = \sum_{i=0}^{m} \binom{m}{i} b_{\tilde{\Sigma}}^i (1 - b_{\tilde{\Sigma}})^{m-i} \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}^2}\right)^i = \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + 1 - b_{\tilde{\Sigma}}\right)^m \quad .$$

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

$$n \cdot p_{\text{ver}} \cdot \mathcal{O}(m) = n \cdot \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + 1 - b_{\tilde{\Sigma}}\right)^m \cdot \mathcal{O}(m) \quad .$$

The total cost of searching in our scheme is thus

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

and hence the optimal sampled alphabet $\tilde{\Sigma}$ minimizes the cost per text character

$$E(\tilde{\Sigma}) = \frac{1}{m} + \frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + 1 - b_{\tilde{\Sigma}}\right)^m \cdot m$$

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

The verification cost always increases when a character is removed from the alphabet so the search cost in the sampled text must decrease for the combined cost to decrease. If $R = \Sigma \backslash \tilde{\Sigma}$ is the set of removed characters, the function

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

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 $p_z = (1-b_R) - \sqrt{(1 - b_R)^2 - (a_\Sigma - a_R)}$ 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. 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 decreases 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 p_c and now we need to decide if we should remove the character c. If $p_c > p_R$, 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 $p_c < p_R$ we know that if we added only c to R the searching 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 $p_c < p_R$. Now the new removed set will be $R' = R \cup \{c\}$ so we get $a_{R'} = a_R + p_c^2$ and $b_{R'} = b_R + p_c$. Now the new critical probability will be

$$p_{R'} = \frac{a_\Sigma - a_{R'}}{1 - b_{R'}} = \frac{a_\Sigma - a_R - p_c^2}{1 - b_R - p_c} \ .$$

We know that $h_R(p_c) > h_R(p_R) = h_R(0)$ because $p_c < p_R$. Therefore

$$\frac{1}{m} + \frac{a_\Sigma - a_R - p_c^2}{1 - b_R - p_c} > \frac{1}{m} + \frac{a_\Sigma - a_R}{1 - b_R}$$

and so

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

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

```
$R_{opt} = \{\}$
sort characters of $\Sigma$ in descending order
find_opt(0, {})
return $R_{opt}$

**find_opt**($i$, $R$)
1.  if ($i = \sigma$)
2.      if ($E(\Sigma\backslash R) < E(\Sigma\backslash R_{opt})$)
3.          $R_{opt} = R$
4.  else
5.      $p_R = \frac{a_\Sigma - a_R}{1-b_R}$
6.      if ($p_i > p_R$)
7.          find_opt($i+1$, $R \cup \{i\}$)
8.          find_opt($i+1$, $R$)
9.      else
10.         find_opt($\sigma$, $R$)
```

Fig. 3. Pseudo code for searching for the optimal set of removed characters

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 in 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 the decreasing order of frequency. We then figure out if it is beneficial for searching in the sampled text to remove the most frequent character not considered yet. If it is, we try both removing and not removing that character and proceed recursively for both cases. If it is not, we prune the search here because none of the remaining characters should be removed. Figure 3 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 2,810 when considering the King James Bible 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 set 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) it is possible that the optimal set of removed characters is not a set of most frequent characters.

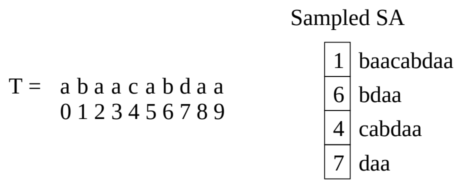


Fig. 4. The sampled suffix array for the text $T = abaacabdaa$ with the sampled alphabet $\tilde{\Sigma} = \{b, c, d\}$. The sorted suffixes are only shown for convenience. They are not part of the structure.

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 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 $\mathcal{O}(\tilde{n})$ words of space if we apply the construction technique of the word suffix array [4]. The sampled suffix array for the text $T = abaacabdaa$ is shown in Fig. 4, where the sampled alphabet is $\tilde{\Sigma} = \{b, c, d\}$.

Search on the sampled suffix array is carried out as follows. Given a pattern $P = p_0 p_1 \dots p_{m-1}$ 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 $p_0 \dots p_{j-1}$ and the suffix starting with the first sampled character $p_j \dots p_{m-1}$. 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 the sampled text, 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.

The sampled suffix array resembles a sparse suffix array [8], 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 would need 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.

5 Optimal Sampling for Suffix Array

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 $\tilde{\Sigma}$ so that the search time would be optimal? Obviously $b_{\tilde{\Sigma}} = b$ 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 probability of matching for the first sampled character is minimized. Thus the sampled alphabet $\tilde{\Sigma}$ 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 $a_R = a_\Sigma - a_{\tilde{\Sigma}}$ where R is the set of removed characters. Thus the average cost of a single verification is $1/(1 - a_\Sigma + a_{\tilde{\Sigma}})$.

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

$$b_{\tilde{\Sigma}} \frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}^2} (a_\Sigma)^{m_s - 1} = \frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} (a_\Sigma)^{m_s - 1}$$

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

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

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

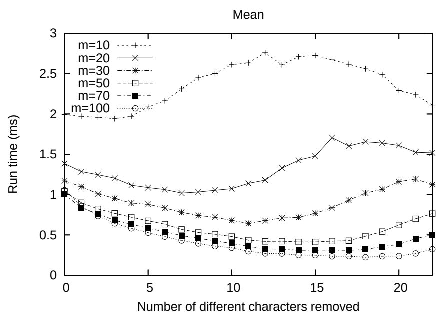
$$f'(a_{\tilde{\Sigma}}) = \frac{1 - a_\Sigma}{(1 - a_\Sigma + a_{\tilde{\Sigma}})^2} > 0$$

so the verification cost increases when $a_{\tilde{\Sigma}}$ increases. To minimize $a_{\tilde{\Sigma}}$ the sampled alphabet $\tilde{\Sigma}$ 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 Sect. 3.

6 Experiments

6.1 Semi-Index

To determine the sampled alphabet, we ran the exact algorithm of Sect. 3 for different pattern lengths to choose the sampled alphabet that produces the smallest estimated cost $E(\tilde{\Sigma})$. For all pattern lengths the algorithm recommended removing a set of most frequent characters. To see how well these results correspond to practice, we tested the semi-index approach by removing the k most frequent



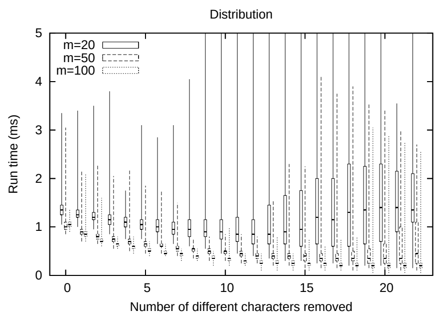


Fig. 5. The running time for various pattern lengths for the basic method. The left figure shows the mean running time; the right shows the median, minimum, maximum, and 25% and 75% quartiles.

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	7	11	13	14	15	17	17	16	18

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. The most frequent characters in the decreasing order of frequency were "␣ethaonsirdlfum,wycgbp" where ␣ is the space character. The tests were run on a 1.0 GHz AMD Athlon dual core processor with 2 GB of memory, 64 kB L1 cache and 512 kB L2 cache, running Linux 2.6.23. The code is in C and compiled with `gcc` using `-O3` optimization.

Figure 5 shows the results of these experiments with the basic method mapping every 64'th sampled character to its position in the original text. If we make the mapping sparser the running time will start to increase a little earlier, but the effect is quite mild. The results for zero removed characters correspond to the original BMH algorithm. As we can see, the semi-index is up to 5 times faster, especially when the patterns are long. Figure 5 also shows that, for each pattern length, there is an optimal number of characters to remove. A comparison of these optima and those given by the analysis is shown in Table 1. As we can see, the analysis gives reasonably good results although it recommends removing too few characters with long patterns, 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.

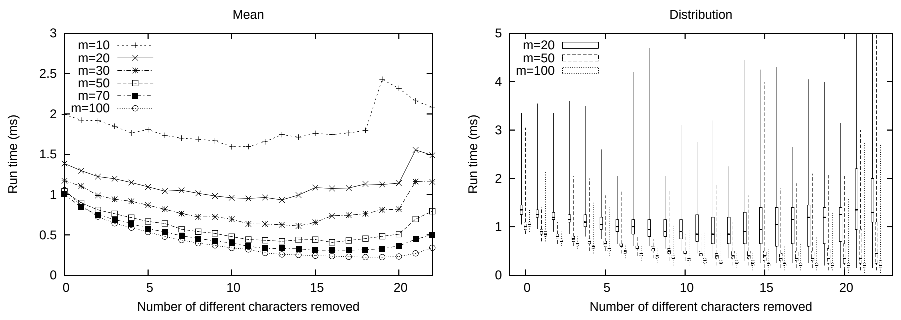


Fig. 6. The running time for various pattern lengths for the tuned version where searching in the sampled text is skipped if it looks like searching in the original text is faster. The left figure shows the mean running time; the right figure shows the median, minimum, maximum, and 25% and 75% quartiles.

The results for the tuned method, where we search the original text if the ratio $n/\min(m, \bar{\sigma})$ looks unfavorable for searching the sampled text, is shown in Fig. 6. Again we are mapping every 64'th sampled character to its position in the original text. As we can see, the optimal number of removed characters is closer to being the same for all pattern lengths than in the basic approach. For example by choosing to remove the 13 most frequent characters, we would do reasonably well for all pattern lengths using just 0.18 times the original text size to store the sampled text. Comparing Figs. 5 and 6 we see that the median running times are almost the same, but the maximum and the 75% quartile are lower for the tuned method. This is also reflected in the average values.

6.2 Sampled Suffix Array

Figure 7 shows the results obtained by comparing our sampled suffix array against our implementation of the sparse suffix array [8] and the locally compressed suffix array (LCSA) [6], an index that compresses the differential suffix array using Re-Pair. 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 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 texts from the *PizzaChili* site, `http://pizzachili.dcc.uchile.cl`.

Our approach performs very well for moderate to long patterns. Already for $m = 50$ 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 [5,13] are designed to use much less space (e.g. 0.8 times the text size including the text) but take much more time, and thus

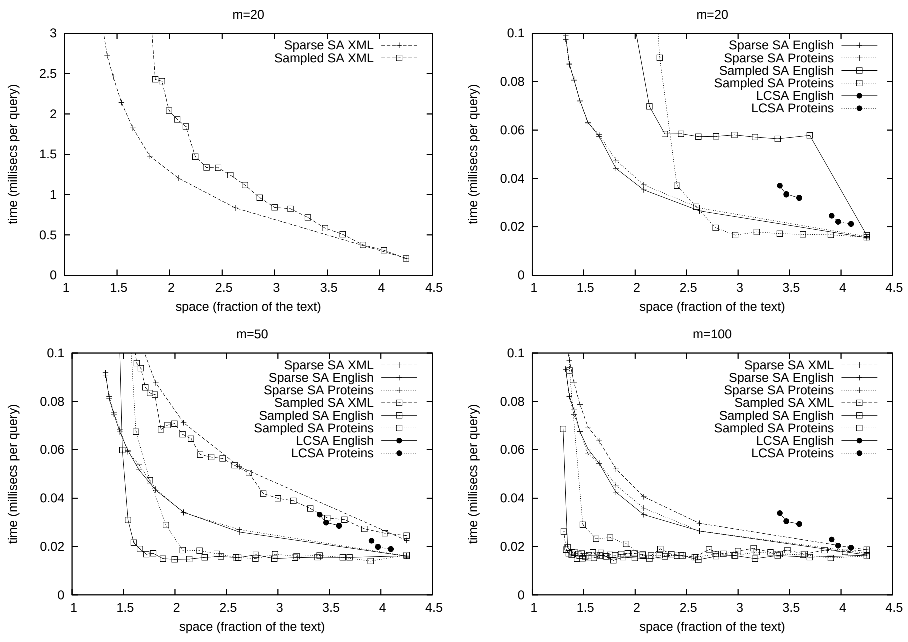


Fig. 7. Search times for the sampled and sparse suffix arrays and LCSA for XML, English and protein data. LCSA uses little space for XML data but it is much slower than the other approaches, so these results are not shown. The top figures show results for pattern length 20 and the bottom figures show the results for pattern lengths 50 and 100. The space fraction includes that of the text, so it is of the form $1 + \frac{\text{index size}}{\text{text size}}$.

are inappropriate for this comparison. We chose the LCSA as an alternative that compresses less but is much faster than the other self-indexes [6]. 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.

7 Conclusions and Further Work

We have presented two sampling approaches to speed up string matching with long patterns. The sampled semi-index profits from nonuniform character distribution to gain a speedup over online searching, while the sampled suffix array works also with a uniform distribution. It is also worth noting that in the semi-index approach the sampled text is an internal structure of the semi-index so any transform, like compression or code splitting [15], could be applied to it.

The current approach is not applicable to small alphabets. To extend the approach to smaller alphabets 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 interesting direction to minimize the extra space of the semi-index is to replace the original text by the subsequence of the non-sampled characters, and use a bitmap to indicate the subset each symbol of T belongs to. With *rank*/*select* capabilities [13] this bitmap replaces the current position mapping for verification and permits searching on the sampled or the unsampled characters.

References

1. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enchanced suffix arrays. Journal of Discrete Algorithms **2**(1) (2004) 53–86
2. Baeza-Yates, R.: String searching algorithms revisited. In Dehne, F., Sack, J.R., Santoro, N., eds.: WADS 1989. LNCS, vol. 382, Springer, Heidelberg (1989) 75–96
3. Crochemore, M., Czumaj, A., Gąsieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., Rytter, W.: Speeding up two string-matching algorithms. Algorithmica **12** (1994) 247–267
4. Ferragina, P., Fischer, J.: Suffix arrays on words. In Ma, B., Zhang, K., eds.: CPM 2007. LNCS, vol. 4580, Springer, Heidelberg (2007) 328–339
5. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice! Manuscript. `http://pizzachili.dcc.uchile.cl` (2007)
6. González, R., Navarro, G.: Compressed text indexes with fast locate. In Ma, B., Zhang, K., eds.: CPM 2007. LNCS, vol. 4580, Springer, Heidelberg (2007) 216–227
7. Horspool, R.N.: Practical fast searching in strings. Software – Practise & Experience **10** (1980) 501–506
8. Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In Cai, J., Wong, C.K., eds.: COCOON 1996. LNCS, vol. 1090, Springer, Heidelberg (1996) 219–230
9. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing **6** (1977) 323–350
10. Manber, U., Myers, G.: Suffix arrays: A new method for online string searches. SIAM Journal on Computing **22**(5) (1993) 935–948
11. Moura, E., Navarro, G., Ziviani, N., Baeza-Yates, R.: Fast and flexible word searching on compressed text. ACM Trans. on Information Systems **18**(2) (2000) 113–139
12. Navarro, G., Baeza-Yates, R., Sutinen, E., Tarhio, J.: Indexing methods for approximate string matching. IEEE Data Engineering Bulletin **24**(4) (2001) 19–27
13. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys **39**(1) (2007) 1–61
14. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences. Cambridge University Press (2002)
15. Rautio, J., Tanninen, J., Tarhio, J.: String matching with stopper encoding and code splitting. In Apostolico, A., Takeda, M., eds.: CPM 2002. LNCS, vol. 2373, Springer, Heidelberg (2002) 45–52