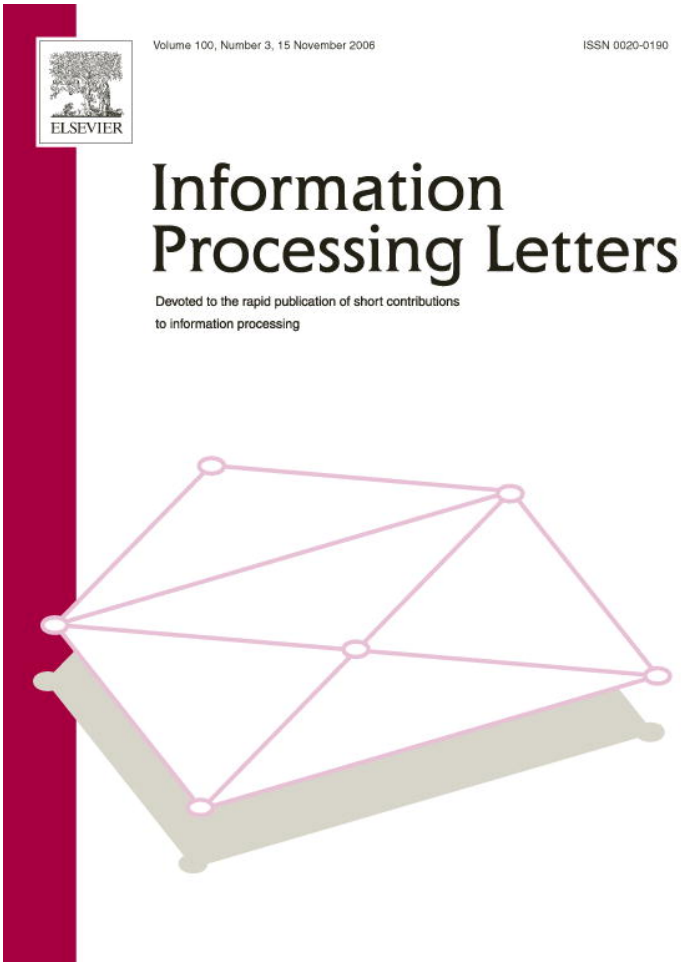

Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 100 (2006) 91–96
Information Processing Letters
www.elsevier.com/locate/ipl

Efficient parameterized string matching

Kimmo Fredriksson *,[1], Maxim Mozgovoy

Department of Computer Science, University of Joensuu, P.O. Box 111, 80101 Joensuu, Finland

Received 13 February 2006; received in revised form 1 June 2006; accepted 22 June 2006

Available online 2 August 2006

Communicated by S.E. Hambrusch

Abstract

In parameterized string matching the pattern P matches a substring t of the text T if there exist a bijective mapping from the symbols of P to the symbols of t. We give simple and practical algorithms for finding all such pattern occurrences in sublinear time on average. The algorithms work for a single and multiple patterns.

Keywords: Algorithms; Parameterized string matching; Bit-parallelism; Suffix automaton

1. Introduction

In traditional string matching problem one is interested in finding the occurrences of a pattern P from a text T, where P and T are strings over some alphabet Σ. Many variations of this basic problem setting exist, such as searching multiple patterns simultaneously, and/or allowing some limited number of errors in the matches, and indexed searching, where T can be preprocessed to allow efficient queries of P. See, e.g., [13,16,11] for an overview and references. Yet another variation is *parameterized matching* [6]. In this variant we have two disjoint alphabets, Σ for *fixed* symbols, and Λ for *parameter* symbols. In this setting we search *parameterized* occurrences of P, where the symbols from Σ must match exactly, while the symbols in Λ can be also renamed. This problem has important applications, e.g., in software maintenance and plagiarism detection [6], where the symbols of the strings can be, e.g., reserved words and identifier or parameter names of some (possibly tokenized) programming language source code. Hence one might be interested in finding code snippets that are the same up to some systematical variable renaming.

A myriad of algorithms have been developed for the classical problem, but only a few exist for parameterized matching. In [5] exact on-line matching algorithm for a single pattern was developed. This algorithm runs in $O(n \log \min(m, |\Lambda|))$ worst case time. However, the average case time was not analyzed. Another algorithm was given in [2], that achieves the same time bound both in average and worst cases. In the same paper it was shown that this is optimal, and that in particular the log factor cannot be avoided for general alphabets. However, for fixed alphabets we can avoid it, as shown in the present paper. In [14] it was shown that multiple patterns can be searched in $O(n \log(|\Sigma| + |\Lambda|) + occ)$ time, where occ is the number of occurrences of all the patterns. Other algorithms exist for the off-line problem

* Corresponding author.
E-mail address: kfredrik@cs.joensuu.fi (K. Fredriksson).
[1] Supported by the Academy of Finland, grant 202281.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2006.06.009

[6,9]. In this paper we develop algorithms that under mild assumptions run in optimal time on average, are simple to implement and perform well in practice. Our algorithms are based on generalizing the well-known Shift-Or [4] and Backward DAWG (Directed Acyclic Word Graph) Matching algorithms [7,10]. Our algorithms generalize for the multipattern matching as well.

2. Preliminaries

We use the following notation. The *pattern* is $P[0 \dots m-1]$ and the *text* is $T[0 \dots n-1]$. The symbols of P and T are taken from two disjoint finite alphabets Σ of size σ and Λ of size λ. The pattern P matches the text substring $T[j \dots j+m-1]$, iff for all $i \in \{0 \dots m-1\}$ it holds that $M_j(P[i]) = T[j+i]$, where $M_j(\cdot)$ is one-to-one mapping on $\Sigma \cup \Lambda$. Moreover, the mapping must be identity on Σ, but on Λ can be different for each text position j. For example, assume that $\Sigma = \{A, B\}$, $\Lambda = \{X, Y, Z\}$ and $P =$ AAZYZABXYZAX. Then P matches the text substring AAZYZABXYZAX with identity mapping, and AAXYXABZYXAZ with parameter mapping X $\mapsto$ Z, Y $\mapsto$ Y, and Z $\mapsto$ X. This mapping is simple with *prev* encoding [6]. For a string S, $prev(S)$ maps all parameter symbols s in S to a non-negative integer p, where p is the number of symbols since the last occurrence of symbol s in S. The first occurrence of the parameter is encoded as 0. If s belongs to Σ, it is mapped to itself (s). For our example pattern, $prev(P) =$ AA002AB055A4. This is the same as the encoding for the two example substrings, i.e., $prev$(AAZYZABXYZAX) = $prev$(AAXYXABZYXAZ). Hence the problem is reduced to exact string matching, where we match $prev(P)$ against $prev(T[j \dots j+m-1])$ for all $j = 0 \dots n-m$. The string $prev(S)$ can be easily computed in linear time for constant size alphabets. The only remaining problem then is how to maintain $prev(T[j \dots j+m-1])$ (and any algorithmic parameters that depend on it) efficiently as j increases. The key is the following lemma [6].

Lemma 1. *Let $S' = prev(S)$. Then for $S'' = prev(S[j \dots j+m-1])$ for all i such that $S[i] \in \Lambda$ it holds that $S''[i] = S'[i]$ iff $S'[i] < m$. Otherwise $S''[i] = 0$.*

We are now ready to present our algorithms. For simplicity we assume that Σ and Λ are finite constant size alphabets. For large alphabets all our time bounds hold if we multiply them by $O(\log(m))$.

3. Parameterized bit-parallel matching

In this section we present bit-parallel approach for parameterized matching, based in Shift-Or algorithm [4]. For the bit-parallel operations we adopt the following notation. A machine word has w bits, numbered from the least significant bit to the most significant bit. We use C-like notation for the bit-wise operations of words; & is bit-wise `and`, | is `or`, ^ is `xor`, $\sim$ negates all bits, $\ll$ is shift to left, and $\gg$ shift to right, both with zero padding. For brevity, we make the assumption that $m \leqslant w$, unless explicitly stated otherwise.

The standard Shift-Or automaton is constructed as follows. The automaton has states $0, 1, \dots, m$. The state 0 is the initial state, state m is the final (accepting) state, and for $i = 0, \dots, m-1$ there is a transition from the state i to the state $i+1$ for character $P[i]$. In addition, there is a transition for every $c \in \Sigma$ from the initial state to the initial state, which makes the automaton nondeterministic. The preprocessing algorithm builds a table B, having one bit-mask entry for each $c \in \Sigma$. For $0 \leqslant i \leqslant m-1$, the mask $B[c]$ has ith bit set to 0, iff $P[i] = c$. These correspond to the transitions of the implicit automaton. That is, if the bit i in $B[c]$ is 0, then there is a transition from the state i to the state $i+1$ with character c. The bit-vector D encodes the states of the automaton. The ith bit of the state vector is set to 0, iff the state i is active, i.e., the pattern prefix $P[0 \dots i]$ matches the current text position. Initially each bit is set to 1. For each text symbol c the vector is updated by $D \leftarrow (D \ll 1) \mid B[c]$. This simulates all the possible transitions of the nondeterministic automaton in a single step. If after the update the mth bit of d is zero, then there is an occurrence of P. If $m \leqslant w$, then the algorithm runs in time $O(n)$.

In order to generalize Shift-Or for parameterized matching, we must take care of three things:

(i) P must be encoded with *prev*;
(ii) $prev(T[j \dots j+m-1])$ must be maintained in $O(1)$ time per text position;
(iii) the table B must be built so that all parameterized pattern prefixes can be searched in parallel.

The items (i) and (ii) are trivial, while (iii) is a bit more tricky. To compute $prev(P)$ we just maintain an array $prv[c]$ that for each symbol $c \in \Lambda$ stores the position of its last occurrence. Then $prev(P)$ can be computed in $O(m)$ time by a linear scan over P. To simplify indexing in the array B, we assume that $\Sigma = \{0 \dots \sigma-1\}$, and map the *prev* encoded parameter offsets into the

```
1   P' ← Encode(P, m)
2   for i ← 0 to σ + m − 1 do B[i] ← ∼0 ≫ (w − m)
3   for i ← 0 to λ − 1 do prv[σ + i] ← −∞
4   for i ← 0 to m − 1 do B[P'[i]] ← B[P'[i]] & ∼(1 ≪ i)
5   for i ← 1 to m − 1 do B[σ + i] ← B[σ + i] & (B[σ] | (∼0 ≪ i))
6   D ← ∼0; mm ← 1 ≪ (m − 1)
7   for i ← 0 to n − 1 do
8       c ← T[i]
9       if c ∈ Λ then
10          c ← i − prv[T[i]] + σ
11          if c > σ + m − 1 then c ← σ
12          prv[T[i]] ← i
13      D ← (D ≪ 1) | B[c]
14      if (D & mm) ≠ mm then report match
```

Algorithm 1. P-Shift-Or(T, n, P, m).

range $\{\sigma \ldots \sigma + m - 1\}$. The text is encoded in the same way, but the encoding is embedded into the search code. The only difference is that we apply Lemma 1 to reset offsets that are greater than $m - 1$ (i.e., offsets that are for parameters that are outside of the current text window) to zero. Otherwise the search algorithm is exactly the same as for normal Shift-Or.

The tricky part is the preprocessing phase. We denote the *prev* encoded pattern as P'. At first P' is preprocessed just as P in the normal Shift-Or algorithm. This includes the parameter offsets, which are handled as any other symbol. However, this is not enough. We illustrate the problem by an example. Let $P = \text{XAXAX}$ and $T = \text{ZZAZAZAZ}$. In encoded forms these are $P' = 0\text{A}2\text{A}2$ and $T' = 01\text{A}2\text{A}2\text{A}2$. Clearly P has two (overlapping) parameterized matches in T. However, P' does not match in T' at all. The problem is that as the algorithm searches all the m prefixes of the pattern in parallel, then some non-zero encoded offset p (of some text symbol) should be interpreted as zero in some cases. These prefixes have lengths from 1 to m. To successfully apply Lemma 1 we should be able to apply it in parallel to all m substrings. In other words, any non-zero parameter offset p must be treated as zero for all pattern prefixes whose length h is less than p, since by Lemma 1 the parameter with offset p is dropped out of the window of length h. This problem can be solved as follows. The bit-vector $B[\sigma + i]$ is the match vector for offset i. If the j bit of this vector is zero, it means by definition that $P'[j] = i$. If any of the i least significant bits of $B[\sigma]$ are zero, we clear the corresponding bits of $B[\sigma + i]$ as well. More precisely, we set

$$B[\sigma + i] \leftarrow B[\sigma + i] \,\&\, \big(B[\sigma] \mid (\sim 0 \ll i)\big).$$

This means that the offset i is treated as offset i for prefixes whose length is greater than i, and as zero for the shorter prefixes, satisfying the condition of Lemma 1.

Algorithm 1 gives the complete code. The algorithm clearly runs in $O(n\lceil m/w \rceil)$ worst case time. For long patterns one can search just a length w prefix of the pattern, and verify with the whole pattern whenever the prefix matches, giving $O(n)$ average time. However, note that a long variable name (string) is just one symbol (token) in typical applications, hence w bits is usually plenty. Finally, note that for unbounded alphabets we cannot use arrays for *prv* and B. We can use balanced trees instead, but then the time bounds must be multiplied by $O(\log(m))$.

Standard Shift-Or can be improved to run in optimal $O(n \log_\sigma(m)/m)$ average time [12]. The algorithm takes a parameter q, and from the original pattern generates a set $\mathcal{P}$ of q new patterns $\mathcal{P} = \{P^0, \ldots, P^{q-1}\}$, each of length $m' = \lfloor m/q \rfloor$, where $P^j[i] = P[j + iq]$ for $i = 0 \ldots \lfloor m/q \rceil - 1$. In other words, the algorithm generates q different alignments of the original pattern P, each alignment containing only every qth character. The total length of the patterns in $\mathcal{P}$ is $q\lfloor m/q \rfloor \leqslant m$. For example, if $P = \text{ABCDEF}$ and $q = 3$, then $P^0 = \text{AD}$, $P^1 = \text{BE}$ and $P^2 = \text{CF}$. Assume now that P occurs at $T[i..i + m - 1]$. From the definition of P^j it directly follows that $P^j[h] = T[i + j + hq]$, where $j = i \bmod q$ and $h = 0 \ldots m' - 1$. This means that we can use the set $\mathcal{P}$ as a filter for the pattern P, and that the filter needs only to scan every qth character of T. All the patterns must be searched simultaneously. Whenever an occurrence of P^j is found in the text, we must verify if P also occurs, with the corresponding alignment.

This method clearly works for parameterized matching as well. We generate the set of patterns $\mathcal{P}$, and also *prev*-encode them. In the search phase the text is also encoded on-line, encoding only every qth symbol, but assuming that they are consecutive. In other words, every parameter offset is effectively divided by q to agree with the encoding of the patterns. Finally, the verifica-

tion phase checks if $prev(P) = prev(T[v \ldots v+m-1])$, where v is the starting position of a potential match.

The search of the pattern set can be done using the parameterized Shift-Or algorithm. This is possible by concatenating and packing the set of patterns into a single machine word [12,4]. Another alternative is to use the parameterized version [14] of Aho–Corasick algorithm [1]. Both lead to the same average case running time, but the latter does not require that $m \leqslant w$, as it is not based on bit-parallelism. We denote the Shift-Or based algorithm as PFSO. The filtering time is $O(n/q)$. The filter searches the exact matches of q patterns, each of length $\lfloor m/q \rfloor$. We are not able to analyze the exact effect of the parameter alphabet to the probability that two randomly picked symbols match. However, if we assume that a constant fraction ε of the pattern positions are randomly selected to have a randomly selected symbol from Σ, then the probability that P^j occurs in a given text position is $O((1/\sigma)^{\lfloor \varepsilon m/q \rfloor})$. A brute force verification cost is in the worst case $O(m)$ (but only $O(1)$ on average). To keep the total time at most $O(n/q)$ on average, we select q so that $n/q = mn/\sigma^{\varepsilon m/q}$, i.e., $q = O(m/\log_\sigma(m))$. The total average time is therefore $O(n \log_\sigma(m)/m)$. This is optimal [17] within a constant factor.

Finally, note that this method works for searching r patterns simultaneously. The only difference is that we search q pieces of all the r patterns simultaneously, and verify the corresponding pattern whenever any of the rq pieces match. Redoing the analysis we obtain that the $O(\log(m))$ factor is replaced with $O(\log(rm))$. In this case we prefer using the Aho–Corasick based algorithm [14], since the number of patterns it can handle does not depend on w.

4. Parameterized backward trie matching

We now present an algorithm based on Backward DAWG Matching (BDM) [7,10]. BDM is optimal on average, i.e., it runs in $O(n \log_\sigma(m)/m)$ average time. We call our parameterized version of BDM as Parameterized Backward Trie Matching, PBTM, for short. In the preprocessing phase PBTM builds a trie for the encoded suffixes of the reversed pattern. A trie is a rooted tree, where each edge is labeled by a symbol. The edges of the path from the root node to some leaf node then spell out the string of symbols stored into that leaf. The pattern in reverse is denoted by P^r. The set of its suffixes is $\{P^r[i \ldots m-1] \mid 0 \leqslant i < m\}$ (note that this corresponds to the prefixes of the original pattern). Each suffix is then encoded with *prev*, and the encoded strings are inserted into a trie. For example, if $P = \text{AZBZXBXY}$, then the set of stored strings is $\{00b20b2a, 0b20b2a, b00b2a, 00b2a, 0b2a, b0a, 0a, a\}$, The trie allows efficient searching of any pattern substring that occurs in P^r. A brute force algorithm for this takes $O(m^2)$ time, but can be improved to $O(m)$ by using efficient suffix tree construction algorithms for parameterized strings [9]. An alternative to the trie is suffix array [15], i.e., the trie can be replaced with sorted array of *prev* encoded suffixes of the reverse pattern. For the above example string, $P = \text{AZBZXBXY}$, we create an array $A = \{00b20b2a, 00b2a, 0a, 0b20b2a, 0b2a, a, b00b2a, b0a\}$. Following an edge in the trie can then be simulated by a binary search in the array. We call the resulting algorithm PBAM. The benefit is that the array based method is easy to implement space efficiently since only one pointer is needed for each suffix.

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

Some care must be taken to be able to do the encoding of the text window in $O(1)$ time per read symbol. To achieve constant time per symbol we must use an auxiliary array *prv* (as before) to store the position of the last occurrence for each symbol. We cannot afford to initialize the whole array for each window, so before shifting the window we rescan the symbols just read in the current window, and reinitialize the array only for

```
1   root ← EncSTrie(P^r)
2   for i ← 0 to λ − 1 do prv[σ + i] ← −∞
3   i ← 0
4   while i < n − m do
5       j ← m; shift ← m; u ← root
6       while u ≠ null do
7           c ← T[i + j − 1]
8           if c ∈ Λ then
9               c ← m − j − prv[T[i + j − 1]] + σ
10              if c > σ + m − 1 then c ← σ
11              prv[T[i + j − 1]] ← m − j
12          j ← j − 1
13          u ← child(u, c)
14          if u ≠ null AND issuffix(u) then
15              if j > 0 then shift ← j else report match
16      for k ← i + j to i + m − 1 do if T[k] ∈ Λ then prv[T[k]] ← −∞
17      i ← i + shift
```

Algorithm 2. PBTM(T, n, P, m).

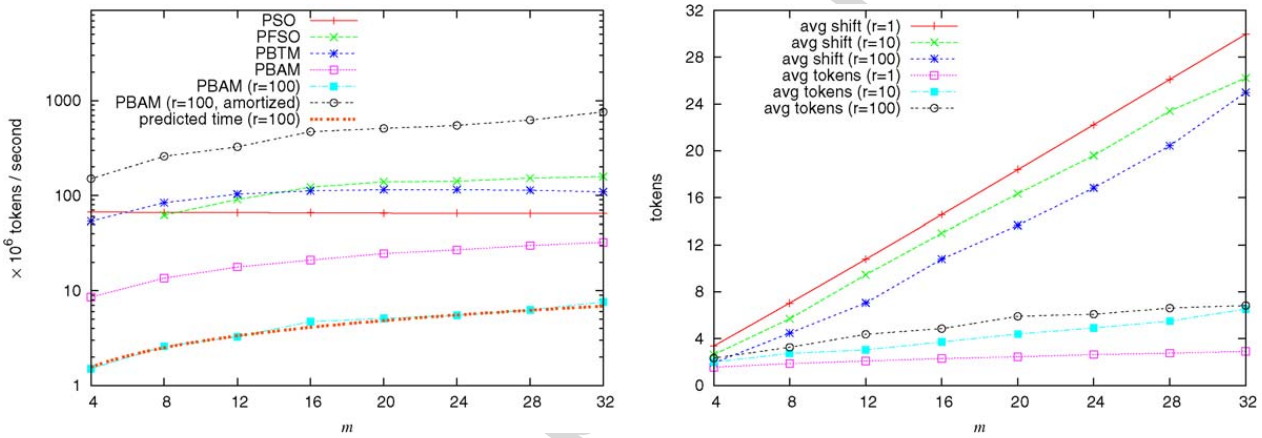

Fig. 1. Left: the search speed in 10^6 tokens/second. Right: the average shift and average number of tokens inspected in each window of length m.

those symbols. This ensures O(1) total time for each symbol read. Algorithm 2 gives the code.

The average case running time of this algorithm depends on how many symbols x are examined in each window. Again, if we make the simplifying assumption that a constant fraction of the pattern positions are randomly selected to have a randomly selected symbol from Σ, then the original analysis of BDM holds for PBTM as well, and the average case running time is $O(n \log_\sigma(m)/m)$. For general alphabets and for the PBAM version the time must be multiplied by $O(\log(m))$. Finally, this algorithm can be easily modified to search r patterns simultaneously. Basically, if all the patterns are of the same length, this generalization requires just storing all the suffixes of all the patterns into the same trie. This results in $O(n \log_\sigma(rm)/m)$ average time. With modest additional complexity patterns of different lengths can be handled as well in the same way as with regular BDM [11].

5. Comparison

For a single pattern our only competitor [5] is based on (Turbo) Boyer–Moore [8,10] algorithm. However, BM-type algorithms are known to be clearly worse than the more simple bit-parallel and suffix-automaton based approaches [16], an this becomes more and more clear as the pattern length increases. Moreover, BM-type algorithms have poor performance when generalized for multiple string matching [16]. As for the multiple matching, our only competitor [14] is the algorithm based on Aho–Corasick automaton, but as detailed in Section 3, we can use exactly their algorithm (even the same implementation) as a fast filter to obtain (near) optimal average case time. Their worst case time can be also preserved. Hence, their algorithm cannot beat ours. We note that all our algorithms can be improved to take only $O(n)$ (or $O(n \log(rm))$ for unbounded alphabets) worst case time. PFSO can be combined with PSO (as

in [12]) and PBTM with the algorithm in [14]. See also [3,10] for similar techniques.

Our goals in this paper are two-folded. First, to develop algorithms that have optimal average case running time for both single and multiple patterns. All the previous results only prove optimal worst case time. Second, to be practical, i.e., to develop algorithms that are simple to implement and have good average case time in practice. We now show that our algorithms behave like predicated, with realistic real world data.

5.1. Experimental results

We have implemented the algorithms in C++, and compiled them with Borland C++ Builder 6. We performed the experiments on the AMD Sempron 2600+ (1.88 GHz) machine with 768 MB RAM, running Windows XP. A tokenized string of concatenated Java source files (taken from various open source projects, such as jPOS, smppapi, and TM4J) was used as a text to be searched. The tokenization procedure (based on JavaCC[2] parser) converted an input file into a sequence of two-byte codes, representing single characters, reserved Java words and distinct identifiers. The initial string had a size of 5.48 MB, and after encoding it consisted of 1259799 tokens, including 51 reserved Java words and 10213 unique identifiers. A set of 100 patterns for each length reported was randomly extracted from the input text. We report the average number of tokens searched per second for each algorithm.

Fig. 1 summarizes the results. PSO denotes the basic parameterized shift-or algorithm, PFSO the fast parameterized shift-or, PBTM the parameterized backward trie matching algorithm, and PBAM the suffix array version of PBTM. For short patterns plain PSO and PBTM give the best results. PSO is the fastest for $m < 8$, and PBTM takes over until $m = 16$, and PFSO dominates for longer patterns in case of optimal q selection. For $m \in \{8, 12, 16, 20, 24, 28, 32\}$ we used $q = \{2, 3, 4, 4, 4, 5, 6\}$, respectively. For long patterns PBTM suffers from the large alphabet size. In our implementation we used arrays to implement the trie nodes and for long patterns the trie requires a lot of initialization time and memory, not fitting into the CPU cache. PBAM does not have this flaw, but the binary search step needed for each accessed text symbol makes it comparatively slow. We also experimented with the multipattern version of PBAM, searching $r = 100$ patterns simultaneously. The plot shows that while the raw speed is reduced, the amortized speed per pattern is clearly better than for any of the single pattern matching algorithms. The time also coincides nicely with the theoretical curve $O(n \log_\sigma(rm) \log_2(rm)/m)$, supporting our analysis. This is also clear given the right plot, showing the average number of tokens inspected in each text window, and the average shift for $r = 1, 10, 100$. These behave like in random texts supporting our assumptions in the analysis.

We have shown how two well-known algorithms, namely Shift-Or and BDM, can be generalized for parameterized matching. The algorithms are easy to implement, and work well in practice.

[2] http://javacc.dev.java.net/.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Comm. ACM 18 (6) (1975) 333–340.
[2] A. Amir, M. Farach, S. Muthukrishnan, Alphabet dependence in parameterized matching, Inform. Process. Lett. 49 (3) (1994) 111–115.
[3] R.A. Baeza-Yates, String searching algorithms revisited, in: Proceedings of WADS'89, in: Lecture Notes in Computer Science, vol. 382, Springer, Berlin, 1989, pp. 75–96.
[4] R.A. Baeza-Yates, G.H. Gonnet, A new approach to text searching, Comm. ACM 35 (10) (1992) 74–82.
[5] B.S. Baker, Parameterized pattern matching by Boyer–Moore-type algorithms, in: Proceedings of the 6th ACM–SIAM Annual Symposium on Discrete Algorithms, San Francisco, CA, 1995, pp. 541–550.
[6] B.S. Baker, Parameterized duplication in strings: algorithms and an application to software maintenance, SIAM J. Comput. 26 (5) (1997) 1343–1362.
[7] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen, J. Seiferas, The smallest automaton recognizing the subwords of a text, Theoret. Comput. Sci. 40 (1) (1985) 31–55.
[8] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Comm. ACM 20 (10) (1977) 762–772.
[9] R. Cole, R. Hariharan, Faster suffix tree construction with missing suffix links, in: Proceedings of ACM–STOC'00, Portland, Oregon, 2000, pp. 407–415.
[10] 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.
[11] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, Oxford, 1994.
[12] K. Fredriksson, Sz. Grabowski, Practical and optimal string matching, in: Proceedings of SPIRE'2005, in: Lecture Notes in Computer Science, vol. 3772, Springer-Verlag, Berlin, 2005, pp. 374–385.
[13] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997.
[14] R.M. Idury, A.A. Schäffer, Multiple matching of parameterized patterns, Theoret. Comput. Sci. 154 (2) (1996) 203–224.
[15] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[16] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings, Cambridge University Press, Cambridge, 2002.
[17] A.C. Yao, The complexity of pattern matching for a random string, SIAM J. Comput. 8 (3) (1979) 368–387.