Efficient Pattern Matching on Binary Strings[⋆]

Simone Faro[1] and Thierry Lecroq[2]

[1] Dipartimento di Matematica e Informatica, Università di Catania, Italy
[2] University of Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France
faro@dmi.unict.it, thierry.lecroq@univ-rouen.fr

1 Introduction

The *binary string matching problem* is an interesting problem in computer science, since binary data are omnipresent in telecom and computer network applications. The main reason for using binaries is size. A binary is a much more compact format than the symbolic or textual representation of the same information. Consequently, less resources are required to transmit binaries over the network. For this reason the binary string matching problem finds applications also in pattern matching on compressed texts, when using the Huffman compression strategy [KS05,SD06].

Binary vectors are usually structured in blocks of k bits, typically bytes ($k = 8$), halfwords ($k = 16$) or words ($k = 32$), which can be processed at the cost of a single operation. If p is a binary string of length m we use the symbol $P[i]$ to indicate the $(i+1)$-th block of p and use $p[i]$ to indicate the $(i+1)$-th bit of p. If B is a block of k bits we indicate with symbol B_j the $(j+1)$-th bit of B. Thus, for $i = 0, \ldots, m-1$ we have $p[i] = P[\lfloor i/k \rfloor]_{i \mod k}$.

Now we present a high level model to process binary strings which exploits the block structure of text and pattern to speed up the searching phase avoiding to work with bitwise operations.

Let $T[i]$ and $P[i]$ denote, respectively, the $(i+1)$-th byte of the text and of the pattern, starting for $i = 0$ with both text and pattern aligned at the leftmost bit of the first byte. Since the lengths in bits of both text and pattern are not necessarily multiples of k, the last byte may be only partially defined. We suppose that the undefined bits of the last byte are set to 0.

In our high level model we define a sequence of several copies of the pattern memorized in the form of a matrix of bytes, *Patt*, of size $k \times (\lceil m/k \rceil + 1)$. Each row i of the matrix *Patt* contains a copy of the pattern shifted by i position to the right. The i leftmost bits of the first byte remain undefined and are set to 0. Similarly the rightmost $k - ((m+i) \mod k)$ bits of the last byte are set to 0. Formally, for $0 \leq i < k$ and $0 \leq h < \lceil (m+i)/k \rceil$, the j-th bit of byte $Patt[i,h]$ is equal to $p[kh - i + j]$ if $0 \leq kh - i + j < m$, 0 otherwise.

Observe that each factor of length k of the pattern appears once in the table *Patt*. In particular, the factor of length k starting at position j of p is memorized in $Patt[k - (j \mod k), \lceil j/k \rceil]$.

[⋆] An extended version of this paper has been published as technical report in [FL08a]

The high level model uses bytes in the matrix *Patt* to compare the pattern block by block against the text for any possible shift of the pattern. However when comparing the first or last byte of P against its counterpart in the text, the bit positions not belonging to the pattern have to be neutralized. For this purpose we define a matrix of bytes, *Mask*, of size $k \times (\lceil m/k \rceil + 1)$, containing binary masks of length k. In particular a bit in the mask $Mask[i,h]$ is set to 1 if and only if the corresponding bit of $Patt[i,h]$ belongs to P. More formally, for $0 \leq i < k$ and $0 \leq h < \lceil (m+i)/k \rceil$, $Mask[i,h]_j = 1$ if $0 \leq kh - i + j < m$, 0 otherwise. Finally we need to compute an array, *Last*, of size k where $Last[i]$ is defined to be the index of the last byte in the row $Patt[i]$. Formally, for $0 \leq i < k$ we define $Last[i] = \lceil (m+i)/k \rceil$. The procedure used to precompute the tables requires $\mathcal{O}(k \times \lceil m/k \rceil) = \mathcal{O}(m)$ time and $\mathcal{O}(m)$ extra-space.

The model uses the precomputed tables to check whether s is a valid shift without making use of bitwise operations but processing pattern and text byte by byte. In particular, for a given shift position s (the pattern is aligned with the s-th bit of the text), we report a match if $Patt[i,h] = T[j+h]$ & $Mask[i,h]$, for $h = 0, 1, ..., Last[i]$, where $j = \lfloor s/k \rfloor$ is the starting byte position in the text and $i = (s \mod k)$.

2 New Efficient Binary String Matching Algorithms

In this section we present two new efficient algorithms for matching on binary strings based on the high level model presented above. The first algorithm is an adaptation of the q-Hash algorithm [Lec07] which has been presented as an adaptation of the Wu and Manber multiple string matching algorithm [WM94] to single string matching problem.

If the pattern p is a binary string we associate each substring of length q with its numeric value in the range $[0, 2^q - 1]$. In order to exploit the block structure of the text we take into account substrings of length $q = k$. This means that, if $k = 8$, each block B of k bits can be considered as a value $0 \leq B \leq 255$. Thus we define a function $Hs : \{0, 1, \ldots, 2^k - 1\} \rightarrow \{0, 1, \ldots, m\}$, such that $Hs(B) = \min(\{0 \leq u < m \mid p[m-u-k\,..\,m-u-1]$ is a suffix of $B\} \cup \{m\})$, for each byte $0 \leq B < 2^k$. Observe that if $B = p[m-k\,..\,m-1]$ then $Hs[B]$ is defined to be 0.

The preprocessing phase of the algorithm consists in computing the function Hs defined above and requires $\mathcal{O}(m + k2^{k+1})$ time complexity and $\mathcal{O}(m + 2^k)$ extra space. During the search phase the algorithm reads, for each shift position s of the pattern in the text, the block $B = t[s+m-q\,..\,s+m-1]$ of k bits. If $Hs(B) > 0$ then a shift of length $Hs(B)$ is applied. Otherwise, when $Hs(B) = 0$ the pattern p is naively checked in the text block by block.

After the test an advancement of length *shift* is applied where *shift* $= \min(\{0 < u < m \mid p[m-u-k\,..\,m-u-1]$ is a proper suffix of $p[m-k\,..\,m-1]\} \cup \{m\})$. If the block B has its $s\ell$ rightmost bits in in the j-th block of T and the $(k - s\ell)$ leftmost bits in the block $T[j-1]$, then it is computed by performing the following bitwise operation $B = (T[j] \gg (k - s\ell)) \mid (T[j-1] \ll (s\ell + 1))$.

The Binary-Hash-Matching algorithm has a $\mathcal{O}(\lceil m/k \rceil n)$ time complexity and requires $\mathcal{O}(m + 2^k)$ extra space.

The second solution can be seen as an adaptation to binary string matching of the Skip-Search algorithm [CLP98].

In the binary case, for each possible block B of k bits, a bucket collects all pairs (i, h) in the table *Patt* such that $Patt[i, h] = B$. When a block of bits occurs more times in the pattern, there are different corresponding pairs in the bucket of that block. Observe that for a pattern of length m there are $m-k+1$ different blocks of length k corresponding to the blocks $Patt[i, h]$ such that $kh - i \geq 0$ and $k(h + 1) - i - 1 < m$.

However, to take advantage of the block structure of the text, we can compute buckets only for blocks contained in the suffix of the pattern of length $m' = k\lfloor m/k \rfloor$. In such a way m' is a multiple of k and we could reduce to examine a block for each m'/k blocks of the text. Formally, for $0 \leq B < 2^k$, $S_k[B]$ is defined as the set $\{(i, h) : (m \mod k) \leq kh - i \leq m - k \ \wedge \ Patt[i, h] = B\}$.

The preprocessing phase of the Binary-Skip-Search algorithm consists in computing the buckets for all possible blocks of k bits. The space and time complexity of this preprocessing phase is $\mathcal{O}(m + 2^k)$. The main loop of the search phase consists in examining every (m'/k)th text block. For each block $T[j]$ examined in the main loop, the algorithm inspects each pair (i, pos) in the bucket $S_k[T[j]]$ to obtain a possible alignment of the pattern against the text. For each pair (i, pos) the algorithm checks whether p occurs in t by comparing $Patt[i, h]$ and $T[j - pos + h]$, for $h = 0, \ldots, Last[i]$. The Binary-Skip-Search algorithm has a $\mathcal{O}(\lceil m/k \rceil n)$ quadratic worst case time complexity and requires $\mathcal{O}(m + 2^k)$ extra space. In practice, if the block size is k, the Binary-Skip-Search algorithm requires a table of size 2^k to compute the function S_k.

3 Experimental Results

Here we present experimental data which allow to compare, in terms of running time and number of text character inspections, the following algorithms: a Binary-Naive algorithm (BNAIVE), Binary-Boyer-Moore by Klein (BBM), the Binary-Hash-Matching (BHM) and the Binary-Skip-Search (BSKS) algorithms. For the sake of completeness we have also tested the following algorithms for standard pattern matching: the q-Hash algorithm [Lec07] with $q = 8$ (HASH8) and the Extended-BOM algorithm [FL08b] (EBOM). The Extended-BOM and the q-Hash algorithms have been tested on the same texts and patterns but in their standard form. The algorithms have been tested on three $\mathsf{Rand}(1/0)_\gamma$ problems, for $\gamma = 50$ and 70. Searching have been performed for binary patterns, of length m from 20 to 500, which have been taken as substring of the text at random starting positions. In particular each $\mathsf{Rand}(1/0)_\gamma$ problem consists of searching a set of 1000 random patterns of a given length in a random binary text of 4×10^6 bits. The distribution of characters depends on the value of the parameter γ. In particular bit 0 appears with a percentage equal to $\gamma\%$. In the following tables, running times (on the left) are expressed in hundredths of seconds. Tables with the number of text character inspections (on the right) are presented in light-gray background color. Best results are bold faced.

m	BNAIVE	BBM	BSKS	BHM	HASH8	EBOM
20	41.53	13.53	3.66	**3.40**	5.12	8.89
60	41.72	7.77	**1.16**	1.60	1.72	3.85
100	41.68	6.80	**0.70**	1.44	1.64	3.06
140	42.11	6.21	**0.89**	1.24	1.54	2.67
180	41.95	5.76	**0.66**	1.10	1.80	2.25
220	41.93	5.36	**0.74**	1.24	1.79	1.87
260	41.95	5.08	**0.54**	1.05	1.47	2.09
300	41.74	5.07	**0.54**	1.11	1.82	1.48
340	41.93	4.86	**0.39**	1.07	1.56	1.56
380	41.97	4.59	**0.46**	0.97	1.87	1.43
420	42.07	4.52	**0.31**	1.23	1.59	1.23
460	41.99	4.68	**0.23**	1.04	1.52	1.19
500	42.06	4.61	**0.37**	0.81	1.53	1.32

BNAIVE	BBM	BSKS	BHM
9.00	1.82	1.04	**0.90**
9.00	0.85	**0.20**	0.31
9.00	0.63	**0.13**	0.20
9.00	0.54	**0.10**	0.15
9.00	0.47	**0.08**	0.13
9.00	0.44	**0.07**	0.11
9.00	0.41	**0.07**	0.10
9.00	0.39	**0.06**	0.09
9.00	0.38	**0.06**	0.09
9.00	0.37	**0.06**	0.08
9.00	0.36	**0.05**	0.08
9.00	0.35	**0.05**	0.08
9.00	0.35	**0.05**	0.07

Experimental results for a Rand$(0/1)_{50}$ problem

m	BNAIVE	BBM	BSKS	BHM	HASH8	EBOM
20	43.26	17.25	**4.01**	4.21	4.86	10.92
60	43.15	10.26	**1.66**	2.09	2.03	4.27
100	43.80	8.44	**1.60**	2.26	1.95	2.54
140	43.70	8.13	**1.28**	1.61	1.52	2.68
180	43.22	7.37	**1.02**	1.67	2.08	2.33
220	43.29	6.82	**1.08**	1.34	1.94	2.50
260	42.93	6.67	**1.07**	1.53	1.79	1.94
300	43.66	6.46	**0.89**	1.22	1.59	1.94
340	43.53	6.35	**0.97**	1.23	1.28	1.86
380	43.76	6.15	**0.70**	1.42	1.31	1.65
420	43.29	6.03	**0.85**	1.34	1.67	1.48
460	43.45	6.00	**0.92**	1.27	1.37	1.43
500	43.31	6.00	**0.70**	1.28	1.41	1.48

BNAIVE	BBM	BSKS	BHM
9.41	2.27	1.12	**1.01**
9.40	1.14	**0.29**	0.38
9.38	0.89	**0.21**	0.26
9.38	0.77	**0.18**	0.21
9.37	0.71	**0.17**	0.18
9.39	0.65	**0.16**	**0.16**
9.38	0.61	**0.15**	**0.15**
9.39	0.59	0.15	**0.14**
9.39	0.57	0.15	**0.13**
9.38	0.55	0.14	**0.12**
9.38	0.54	0.14	**0.12**
9.38	0.53	0.14	**0.11**
9.37	0.51	0.14	**0.11**

Experimental results for a Rand$(0/1)_{70}$ problem

References

[CLP98] C. Charras, T. Lecroq, and J. D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, volume 1448 of *LNCS*, pages 55–64, Piscataway, NJ, 1998. Springer-Verlag.

[FL08a] S. Faro and T. Lecroq. Efficient pattern matching on binary strings. Report arXiv:0810.2390, Cornell University Library, 2008. http://arxiv.org/abs/0810.2390.

[FL08b] S. Faro and T. Lecroq. Efficient variants of the Backward-Oracle-Matching algorithm. In J. Holub and J. Žďárek, editors, *Proc. of the Prague Stringology Conference*, pages 146–160, 2008.

[KS05] S. T. Klein and D. Shapira. Pattern matching in Huffman encoded texts. *Inf. Process. Manage.*, 41(4):829–841, 2005.

[Lec07] T. Lecroq. Fast exact string matching algorithms. *Inf. Process. Lett.*, 102(6):229–235, 2007.

[SD06] D. Shapira and A. H. Daptardar. Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts. *Inf. Process. Manage.*, 42(2):429–439, 2006.

[WM94] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.