Shift-or string matching with super-alphabets

Kimmo Fredriksson [1]
kfredrik@cs.joensuu.fi

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

Abstract

Given a text $T[1 \ldots n]$ and a pattern $P[1 \ldots m]$ over some alphabet Σ of size σ, we want to find all the (exact) occurrences of P in T. The well-known shift-or algorithm solves this problem in time $\mathcal{O}(n\lceil m/w \rceil)$, where w is the number of bits in machine word, using bit-parallelism. We show how to extend the bit-parallelism in another direction, using super-alphabets. This gives a speed-up by a factor s, where s is the number of characters processed simultaneously. The algorithm is implemented, and we show that it works well in practice too. The result is the fastest known algorithm for exact string matching for short patterns and small alphabets.

Key words: Algorithms, bit-parallelism, string matching

1 Introduction

We address the well studied exact string matching problem. The problem is to search the occurrences of the pattern $P[1 \ldots m]$ from the text $T[1 \ldots n]$, where the symbols of P and T are taken from some finite alphabet Σ, of size σ. Numerous efficient algorithms solving the problem have been obtained. The first linear time algorithm was given in (Knuth et al., 1977), and the first sublinear expected time algorithm in (Boyer and Moore, 1977). The sublinearity is obtained by skipping some characters of the input text by *shifting* the pattern over some text positions by using the information obtained by matching only a few characters of the pattern.

[1] Work done while the author was working in Department of Computer Science, University of Helsinki. Supported by the Academy of Finland. Preliminary version has appeared in (Fredriksson, 2002).

We apply a super-alphabet method to the well-known shift–or algorithm (Baeza-Yates and Gonnet, 1992; Wu and Manber, 1992). Shift–or algorithm uses bit-parallelism to simulate a non-deterministic automaton efficiently. It is therefore natural to extend this parallelism to another dimension, to process several characters at a single step. The algorithm is not only of theoretical interest. We have implemented it, and give experimental results. The performance is clearly improved, and in some cases even more than the theory predicts.

Super-alphabet methods have appeared before, and the idea was mentioned already in (Knuth et al., 1977; Boyer and Moore, 1977). In (Baeza-Yates, 1989) the BMH algorithm (Horspool, 1980) was modified to use a super-alphabet. This effectively reduced the probability of match of two random characters, thus yielding longer shifts. Similar methods were presented in (Tarhio and Peltola, 1997; Navarro and Tarhio, 2000). In (Masek and Paterson, 1980) the edit distance between two strings is computed using precomputed and tabled blocks of the dynamic programming matrix. This technique achieves a speed-up by a factor of $\mathcal{O}(\log n)$. This comes close to our approach.

Our 'super-alphabet' symbols can be seen as well-known q–grams (substrings of lenght q) treated as single blocks. A related method is presented in (Kytöjoki et al., 2003).

2 Preliminaries

Let $T[1 \ldots n]$ and $P[1 \ldots m]$ be arrays of symbols taken from a finite alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. The array T is called *text*, and the array P is called *pattern*. Usually $m \ll n$. We want to find all the exact occurrences of P in T. This is called an exact string matching problem.

Consider now processing several, say s, symbols of T at a time. That is, we use a super-alphabet of size σ^s, packing s symbols of T to a single super-symbol. Each symbol in T and P requires $b = \lceil \log_2 \sigma \rceil$ bits. Often eight bits are allocated for each symbol, even if $\lceil \log_2 \sigma \rceil < 8$ (8 bit bytes). This means that we can process $\lfloor w/b \rfloor$ symbols at the same time. Assume that we want to process s symbols at the same time. We map T to T', such that

$$\begin{aligned} T'[i] = {} & T[is] + \\ & T[is+1]2^b + \ldots + \\ & T[is+s-1]2^{((s-1)b)}. \end{aligned} \tag{1}$$

That is, we pack as many symbols of T as possible to one word with sb bits. Thus the input for our automaton is a "compressed" version of T, called T'.

Note that the "packing" can be implicit. For example, if T is 8 bit ASCII text, we may just address the string as if it was, say an array of $\lfloor |T|/2 \rfloor$ 16bit integers. This is straight-forward for example in `C/C++`. For simplicity, and without loss of generality, we assume that the packing is implicit.

3 Super-alphabet shift–or

In this section we build a non-deterministic finite-state automaton (NFA). Each step of the automaton is simulated using bit-parallel techniques (Baeza-Yates and Gonnet, 1992; Wu and Manber, 1992). Fig. 1 gives an example automaton. The automaton gets T for input, and whenever the accepting state becomes active, there is a match.

We use the following notation. A machine word has w bits, numbered from the least significant bit to the most significant bit. For bit-wise operations of words a C–like notation is used, & is `and`, | is `or`, and $\ll$ is shift to left, with zero padding. The notation 1^i denotes a bit-string of i consecutive 1–bits.

The automaton is constructed as follows. The automaton has states $1, 2, \ldots, m+1$. The state 1 is the initial state, state $m+1$ is the final (accepting) state, and for $i = 1, \ldots, m$ 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 and to the initial state.

The preprocessing algorithm builds a table B. The table have one bit-mask entry for each character in the alphabet. For $1 \leq i \leq m$, the mask $B[c]$ has ith bit set to 0, if and only if $P[i] = c$, and to 1 otherwise. The bit-mask table 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.

We also need a bit-vector D for the states of the automaton. The ith bit of the state vector is set to 0, iff the state i is active. Initially each bit in the state vector is set to 1.

For each new text symbol $T[i]$, we update the vector as follows:

$$D \leftarrow (D \ll 1) \mid B[T[i]] \tag{2}$$

Each state gets its value from the previous one ($D \ll 1$), which remains active only if the text character matches the corresponding transition ($|B[T[i]]$). The first state is set active automatically by the shift operation, which sets the least

significant bit to 0. If after the simulation step (2), the mth (least significant) bit of D is zero, then there is an occurrence of P.

Clearly each step of the automaton is simulated in time $\mathcal{O}(\lceil m/w \rceil)$, which leads to $\mathcal{O}(n\lceil m/w \rceil)$ total time. For small patterns, where $m = \mathcal{O}(w)$, this is a linear time algorithm.

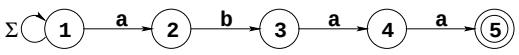

Fig. 1. Forward transitions of the non-deterministic finite automaton for $P = \texttt{abaa}$.

It is possible to improve the NFA simulation by using bit-parallelism also in the indexing of the table B. In effect we use a super-alphabet of size σ^s. It must be emphasized that we do *not* modify the automaton, that is, the automaton does *not* use any super-alphabet, but the super-alphabet is only used to simulate the original automaton faster. Now our simulation algorithm is

$$D \leftarrow (D \ll s) \mid B'[T'[i]]. \tag{3}$$

To compute the table B', we must simulate, i.e. track the movements of the bits for s steps, that we bit-wise `or` into the vector Din the basic algorithm. Let $C = c_s \ll (s-1)b + c_{s-1} \ll (s-2)b + \ldots + c_1$, where $c_1, \ldots, c_s$ form one super-symbol. The original shift-or first sees the symbol c_1, and at the time it sees the next symbol c_2, the information in $B[c_1]$ is already shifted by one bit position. At the time it sees c_s, the bit vector of $B[c_1]$ is shifted $s-1$ bit positions. The (shifted) bit vectors are `or`ed together, and with the state vector. As the `or` operation is commutative, we may compute B' as follows:

$$\begin{aligned} B'[C] \leftarrow & ((B[c_1] \;\&\; 1^m) \ll (s-1)) \mid ((B[c_2] \;\&\; 1^m) \ll (s-2)) \mid \ldots \mid \\ & (B[c_s] \;\&\; 1^m). \end{aligned} \tag{4}$$

Theorem 1 *Equation (3) correctly evaluates the value of the state-vector D, using the table B' of Equation (4).*

Proof. Follows from the fact that bit-wise `or` is commutative operation. □

$B'[C]$ therefore pre-shifts and bit-wise `or`s the state transition information for s consecutive original symbols, and the state vector D is then updated with this precomputation.

After the simulation step any of the bits numbered $m \ldots m+s-1$ may be zero in D. This indicates an occurrence at location $T[is - d + m - 1]$, where $d = m \ldots m+s-1$, corresponding to the offset of the zero bit in D.

The size of the new table B' is $\mathcal{O}(\sigma^s)$. This restricts the full potential of the method. Our implementation uses at most $w/2 = 16$ bits for the super-alphabet. The bit-vectors have length $m + s - 1$, due to the need of $s - 1$ extra bits to handle the super-alphabet (the overflow of the final state). The simulation algorithm works in time $\mathcal{O}(n/s\lceil(m + s - 1)/w\rceil + t)$, where t is the number of matches reported. This is $\mathcal{O}(\frac{nm}{sw} + t)$ for large m. If we select $s = \mathcal{O}(w/\log_2 \sigma)$, the bound becomes $\mathcal{O}(nm \log_2 \sigma/w^2 + t)$.

Theorem 2 *The super-alphabet shift–or algorithm runs in time* $\mathcal{O}(\frac{nm}{sw} + t)$, *using* $\mathcal{O}(\sigma^s)$ *space.* □

4 Experimental results

We have implemented the super-alphabet shift–or algorithm. The implementation is in C, compiled with `gcc` 2.96. The computer was 1333MHz Athlon, 512Mb RAM, running Linux 2.4 operating system.

The test data was a 64Mb generated dna file, in ascii format. We experimented with the shift–or algorithm using super-alphabets $s = 1$ (the original algorithm) and $s = 2$. As the alphabet size for the dna is 4, we also packed the same data, such that each byte contains four characters. For the packed dna we run experiments with $s = 4$ and $s = 8$. The algorithm only counted the number of matches, without explicitly reporting each of them. We experimented with different pattern lengths, but as expected the length did not affect the timings (all the states of the whole automaton fitted in one machine word in each case).

Table 1 gives the timings for the shift–or algorithm. The experimental results show that the method gives a speed-up in practice too. The simplicity of the super-alphabet shift–or method makes it particularly fast in practice. In fact, for packed dna the shift–or method can work faster than predicted by the theory. For example, with $s = 4$, the algorithm was over 5 times faster than with $s = 1$. This is probably due to the smaller input size, so the total number of cache misses is smaller than in the original method. However, using $s = 8$ did not improve the performance that much anymore. This may be partly attributed to the larger table B', and therefore more probable cache misses. The preprocessing time is neglible, less than 0.001 seconds in all cases.

super-alphabet	$s = 1$ (ascii)	$s = 2$ (ascii)	$s = 4$ (packed)	$s = 8$ (packed)
time (s)	0.97	0.52	0.21	0.15

Table 1
Shift–or execution times in seconds for various super-alphabets. Pattern length $m = 16$.

We also implemented the super-alphabet method for the *shift–add* algorithm (Baeza-Yates and Gonnet, 1992). The algorithm computes the number of mismatches (Hamming distance) between the pattern and each text location bit-parallelly. The algorithm is very similar to the shift–or algorithm. The results are presented in Table 2. The parameter k tells the maximum number of mismatches allowed. The times for the super-alphabet grow slightly as k increases. This is due to our implementation; the checking if some position has at most k mismatches is not done entirely parallelly. However, this could be easily fixed by table look-ups. Even so, the super-alphabet is quite effective.

super-alphabet	$s = 1$ (ASCII)	$s = 2$ (ASCII)	$s = 4$ (packed)
$k = 0$, time (s)	1.82	0.99	0.54
$k = 1$, time (s)	1.82	1.00	0.68
$k = 2$, time (s)	1.83	1.00	0.71
$k = 4$, time (s)	1.83	1.15	0.72

Table 2
Shift–add execution times in seconds for various super-alphabets. Pattern length $m = 12$.

The problem with the bit-parallel algorithms is that the word length of the computer architecture (typically 32 or 64) restricts the maximum pattern length [2]. Simulating longer words is easy, but it also causes performance penalty. The shift–or algorithm needs $m + s - 1$ bits for patterns of length m, and super-alphabets of size σ^s. The shift–add algorithm needs $(m + s - 1)(\lceil \log_2(k + s) \rceil + 1)$ bits. Table 2 shows maximal m for various s and k for $w = 64$.

	$k = 1$	$k = 2$	$k = 4$	$k = 8$
$s = 1$	32	21	16	12
$s = 2$	20	20	15	11
$s = 3$	19	14	14	10
$s = 4$	13	13	13	9

Table 3
Maximal m for various s and k for the shift–add algorithm, when the word length is 64 bits.

[2] However, e.g. Intel and AMD have added to their processors SIMD extensions that allow fast (i.e. constant time) operations for longer than 32 bit words, even when the processors are 32 bit. For example, Pentium 4 have instructions that work on 128 bit words in constant time.

5 Conclusions

We have shown that using super-alphabet in sift-or algorithm results in large speed-up, while retaining the simplicity of the original method. The super-alphabet method can be applied to many other algorithms, both to bit-parallel and character comparison based methods.

References

Baeza-Yates, R. A., 1989. Improved string searching. Softw. Pract. Exp. 19 (3), 257–271.

Baeza-Yates, R. A., Gonnet, G. H., 1992. A new approach to text searching. Commun. ACM 35 (10), 74–82.

Boyer, R. S., Moore, J. S., 1977. A fast string searching algorithm. Commun. ACM 20 (10), 762–772.

Fredriksson, K., 2002. Faster string matching with super–alphabets. In: Proceedings of the 9th International Symposium Symposium on String Processing and Information Retrieval (SPIRE'2002). LNCS 2476. Springer–Verlag, pp. 44–57.

Horspool, R. N., 1980. Practical fast searching in strings. Softw. Pract. Exp. 10 (6), 501–506.

Knuth, D. E., Morris, Jr, J. H., Pratt, V. R., 1977. Fast pattern matching in strings. SIAM J. Comput. 6 (1), 323–350.

Kytöjoki, J., Salmela, L., Tarhio, J., 2003. Tuning string matching for huge pattern sets. In: Proceedings of CPM'03. To Appear.

Masek, W. J., Paterson, M. S., 1980. A faster algorithm for computing string edit distances. J. Comput. Syst. Sci. 20 (1), 18–31.

Navarro, G., Tarhio, J., 2000. Boyer-Moore string matching over ziv-lempel compressed text. In: Giancarlo, R., Sankoff, D. (Eds.), Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching. No. 1848 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Montréal, Canada, pp. 166–180.

Tarhio, J., Peltola, H., 1997. String matching in the DNA alphabet. Softw. Pract. Exp. 27 (7), 851–861.

Wu, S., Manber, U., 1992. Fast text searching allowing errors. Commun. ACM 35 (10), 83–91.