UNIVERSITY OF JOENSUU

DEPARTMENT OF COMPUTER SCIENCE AND STATISTICS

Report Series A

Bit-parallel string matching under Hamming distance in $O(n\lceil m/w \rceil)$ worst case time

Szymon Grabowski and Kimmo Fredriksson

A-2007-2

ACM F.2.2, H.3.3

ISBN 978-952-458-960-4

ISSN 1796-7317

May 2, 2007

Bit-parallel string matching under Hamming distance in $O(n\lceil m/w\rceil)$ worst case time

Szymon Grabowski
Computer Engineering Department
Technical University of Łódź
sgrabow@kis.p.lodz.pl

Kimmo Fredriksson*
Department of Computer Science and Statistics
University of Joensuu
kfredrik@cs.joensuu.fi

Abstract

Given two strings, a pattern P of length m and a text T of length n over some alphabet Σ, we consider the string matching problem under k mismatches. The well–known Shift-Add algorithm (Baeza-Yates and Gonnet, 1992) solves the problem in $O(n\lceil m\log(k)/w\rceil)$ worst case time, where w is the number of bits in a computer word. We present two algorithms that improve this to $O(n\lceil m\log\log(k)/w\rceil)$ and $O(n\lceil m/w\rceil)$, respectively. The algorithms make use of nested varying length bit-strings, that represent the search state. We call these *Matryoshka counters*. The techniques we developed are of more general use for string matching problems.

Keywords: algorithms, approximate string matching, bit-parallelism, Hamming distance

ACM Classification: F.2.2 [Analysis of algorithms and problem complexity]: Nonnunmerical algorithms and problems — *Pattern matching, Computations on discrete structures*; H.3.3 [Information storage and retrieval]: Information Search and Retrieval — *Search process.*

1 Introduction

Approximate string matching is a classical problem [8], with myriad of applications e.g. in text searching, computational biology, pattern recognition, etc. Given a text $T = t_0 \ldots t_{n-1}$, a pattern $P = p_0 \ldots p_{m-1}$, over some alphabet Σ, and a threshold k, we want to find all the text positions where the pattern matches the text with at most k mismatches. This is often called approximate string matching under Hamming distance.

A trivial brute–force algorithm solves the problem in $O(mn)$ worst case time. Several more efficient algorithms have been proposed, improving the worst case time to $O(kn + m\log m)$ [6]. Convolutions and Fast Fourier Transform can be used to obtain $O(n|\Sigma|\log m)$ [5], and with a more refined technique, just $O(n\sqrt{k\log k})$ time [1].

Bit-parallelism [2, 3] has become one of the most popular techniques in the field of string matching. Basically, it makes use of wide machine words (CPU registers) to parallelize the work of other algorithms, e.g., filling the matrix in a dynamic programming algorithm or simulating a non-deterministic automaton. In many cases, doubling the machine word width translates to doubling algorithm performance (at least in theory). The most practical string matching algorithm for Hamming distance is Shift-Add [3], based on bit-parallelism. This algorithm

*Supported by the Academy of Finland.

achieves $O(n\lceil m \log(k)/w \rceil)$ worst case time, where w is the number of bits in a computer word (typically 32 or 64).

Besides the algorithms that have good worst case complexity, there are vast number of filtering based algorithms [8] that achieve good average case time. These are usually developed for a different model, namely searching under edit distance (allowing also insertions and deletions of symbols), but they work for Hamming distance as well. In general these algorithms work well for large alphabets and small k/m. The filtering algorithms are based on simple techniques to quickly eliminate the text positions that cannot match with k differences, and the rest of the text is verified using some slower algorithm. Hence algorithms that are efficient in the worst case are still needed.

We propose two modified versions of the Shift-Add algorithm, improving its $O(n\lceil m \log(k)/w \rceil)$ worst case time to $O(n\lceil m \log\log(k)/w \rceil)$ and $O(n\lceil m/w \rceil)$. This matches the time of the best known algorithm for searching under edit distance [7], obtaining optimal parallelization.

2 Preliminaries

Let the pattern $P = p_0 p_1 p_2 \ldots p_{m-1}$ and the text $T = t_0 t_1 t_2 \ldots t_{n-1}$ be strings over alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. The pattern has an exact occurrence in some text position j, if $p_i = t_{j-m+1+i}$ for $i = 0 \ldots m-1$. If $p_i \neq t_{j-m+1+i}$ for at most k positions, then the pattern has an approximate occurrence with at most k mismatches. The number of mismatches is called *Hamming distance* (or, alternatively, the k-mismatches problem). We want to report all text positions j where the Hamming distance is at most k.

As our algorithms belong to the family of bit-parallel techniques, some additional symbols need to be presented. Let w denote the number of bits in computer word (typically 32 or 64). We number the bits from the least significant bit (0) to the most significant bit ($w-1$). C–like notation is used for the bit-wise operations of words; & is bit-wise `and`, | is `or`, ^ is `xor`, ~ negates all bits, << is shift to left, and >> shift to right, both with zero padding.

For brevity, we sometimes use the notation $V_{[i]\ell}$ to denote the ith ℓ bit field of the bit-vector V, that is, the bits $i\ell \ldots (i+1)\ell - 1$ interpreted as ℓ bit binary number. It is easy to extract $V_{[i]\ell}$ in $O(1)$ time using bit-shifts and masks, independent of the length of V, assuming that $\ell \leq w$.

2.1 Shift-Add algorithm

Shift-Add [2, 3] is a bit-parallel algorithm for approximate searching under Hamming distance. Shift-Add reserves a *counter* of $\ell = \lceil \log_2(k+1) \rceil + 1$ bits for each pattern character in a bit-vector D of length $m\ell$ bits. This bit-vector denotes the search state: the ith counter tells the number of mismatches for the pattern prefix $p_0 \ldots p_i$ for some text position j. If the $(m-1)$th counter is at most k at any time, i.e. $D_{[m-1]\ell} \leq k$, then we know that the pattern occurs with at most k mismatches in the current text position j.

The preprocessing algorithm builds an array B of bit-vectors. More precisely, we set $B[c]_{[i]\ell} = 0$ iff $p_i = c$, and 1 otherwise. Then we can accumulate the mismatches as

$$D \leftarrow (D << \ell) + B[t_j].$$

I.e. the shift operation moves all counters at position i to position $i+1$, and effectively clears the counter at position 0. Recall that the counter i corresponds to the number of mismatches for a pattern prefix $p_0 \ldots p_i$. The $+\ B[t_j]$ operation then adds 0 or 1 to each counter, depending on whether the corresponding pattern characters match t_j.

If $D_{[m-1]\ell} < k+1$, the pattern matches with at most k mismatches. Note that since the pattern length is m, the number of mismatches can also be m, but we have allocated only $\ell = O(\log_2 k)$ bits for the counters. This means that the counters can overflow. The solution is to store the highest bits of the fields in a separate computer word o, and keep the corresponding bits cleared in D:

$$D \leftarrow (D << \ell) + B[t_j]$$
$$o \leftarrow (o << \ell) \mid (D \,\&\, om)$$
$$D \leftarrow D \,\&\, {\sim}om$$

The bit mask om has bit one in the highest bit position of each ℓ-bit field, and zeros elsewhere. Note that if o has bit one in some field, the corresponding counter has reached at least value $k+1$, and hence clearing this bit from D does not cause any problems. There is an occurrence of the pattern whenever

$$(D + o) \,\&\, mm \;<\; (k+1) << ((m-1)\ell),$$

i.e. when the highest field is less than $k+1$. The bit mask mm selects the $(m-1)$th field. Shift-Add clearly works in $O(n)$ time, if $m(\lceil \log_2(k+1) \rceil + 1) \leq w$. Otherwise, $\lceil m\ell/w \rceil$ computer words have to be allocated for the counters, and the time becomes $O(n\lceil m \log(k)/w \rceil)$ in the worst case. Note that on average the time is better, since only the words that are "active", i.e. the words that have at least one counter with a value at most k have to be updated. This is possible since the counters can only increase.

Note that the seemingly harder problem, string matching under edit distance, can be solved more efficiently with bit-parallelism, in $O(n\lceil m/w \rceil)$ worst case time [7]. Unfortunately, this algorithm cannot be modified for Hamming distance.

3 Counter-splitting

In this section we show how the number of bits for Shift-Add can be reduced. The idea is simple. We use two levels of counters. The top level is as in plain Shift-Add, i.e. we use $\ell = O(\log(k))$ bits. For the second level we use only $\ell' = \log(\log(k+1)+1)$ bits. The basic idea is then to use a bit-vector D' of $m\ell'$ bits, and accumulate the mismatches as before. However, these counters may overflow every $2^{\ell'}$ steps. We therefore add D' to D at every $2^{\ell'} - 1$ steps, and clear the counters in D'. The result is that updating D' takes only $O(\lceil m \log\log(k)/w \rceil)$ worst case time per text character, and updating D takes only $O(\lceil m \log(k)/w \rceil / 2^{\ell'}) = O(m/w)$ amortized worst case time. The total time is then dominated by computing the D' vectors, leading to $O(n\lceil m \log\log(k)/w \rceil)$ total time. It is easy to notice that no $\ell' = o(\log\log(k))$ can improve the overall complexity.

Note that as we add now values of at most $2^{\ell'} - 1$ to the counters in the D vector, instead of just 0 or 1 (as for D'), we must allocate $\ell = \lceil \log_2(k + 2^{\ell'}) \rceil + 1$ bits for them. However, this is asymptotically the same as before, i.e. $\ell \leq \lceil \log_2(2k) \rceil + 1 = \lceil \log_2(k) \rceil + 2 = O(\log(k))$ bits.

Now adding the two sets of counters can be done without causing an overflow, but the problem is how to add them in parallel. The difficulty is that the counters have different number of bits, and hence are unaligned. The vector D' must therefore be expanded so that we insert $\ell - \ell'$ zero bits between all counters prior to the addition, i.e. we must obtain a bit-vector x, so that

$$x_{[i]\ell} = D'_{[i]\ell'}.$$

Then we can effectively add the counters in D and D' as $D + x$. The simplest solution for computing x is to use look-up tables to do that conversion in constant time, but it seems that this requires $O(2^w)$ space and even more preprocessing time. In the RAM model of computation it is assumed that $w = O(\log_2(n))$, and hence we may use e.g. $\log_2(n)/2$ bit words for indexing the table, and construct the final answer from 2 pieces. The space is then just $2^{\log_2(n)/2} = \sqrt{n}$ words, which is negligible compared to the length of the text. In practice we may use e.g. $w/2$ or $w/4$ bit indexes, depending on w. Hence this solution is $O(1)$ both theoretically and practically.

Alg. 1 Shift-Add-Log-Log-k(T, n, P, m, k).

$\ell' \leftarrow \lceil \log_2(\log_2(k+1)+1) \rceil$
$\ell \leftarrow \lceil \log_2(k + 1 << \ell') \rceil + 1$
$iv \leftarrow 0$
for $i \leftarrow 0$ **to** $m-1$ **do** $iv \leftarrow iv \mid (1 << (i\ell'))$
for $i \leftarrow 0$ **to** $\sigma - 1$ **do** $B[i] \leftarrow iv$
for $i \leftarrow 0$ **to** $m-1$ **do** $B[p_i] \leftarrow iv \wedge (1 << (i\ell'))$
$om \leftarrow 0$
for $i \leftarrow 0$ **to** $m-1$ **do** $om \leftarrow om \mid 1 << ((i+1)\ell - 1)$
$D' \leftarrow 0; D \leftarrow om; j \leftarrow 0$
while $j < n$ **do**
    for $i \leftarrow 1$ **to** $2^{\ell'} - 1$ **do**
        $D' \leftarrow (D' << \ell') + B[t_j]$
        if $D'_{[m-1]} + D_{[m-1-j \mod \ell']} \leq k$ **then** report match
        $j \leftarrow j + 1$
    $x \leftarrow Expand(D')$
    $D \leftarrow D << (2^{\ell'} - 1)\ell$
    $D \leftarrow ((D \ \& \sim om) + x) \mid (D \ \& \ om)$
    $D' \leftarrow 0$

Note that we cannot shift the vector D at each step as this would cost $O(\lceil m\ell/w \rceil)$ time. Instead, we shift it only each $2^{\ell'} - 1$ steps in one shot prior to adding the two counter sets:

$$D \leftarrow D << (2^{\ell'} - 1)\ell.$$

As in plain Shift-Add, we must take care not to overflow the counters. The overflow bit is therefore cleared before the addition, and restored afterwards if it was set:

$$D \leftarrow ((D \ \& \sim om) + x) \mid (D \ \& \ om).$$

The final obstacle is the detection of the occurrences, but this is easy to do. At each step j, we just add $D'_{[m-1]\ell'}$ and $D_{[m-1-j \mod \ell']\ell}$. This constitutes the true sum of mismatches for the whole pattern at text position j. If this sum is at most k, we report an occurrence. Note that this takes only constant time since we only add up two counters, one from each of the two vectors (the whole counter sets are added only each $2^{\ell'} - 1$ steps). Note that as the vector D is not shifted at each step, we simulate the shift by selecting the $(m - 1 - j \mod \ell')$th field when detecting the possible occurrences.

Summing up, we have $O(n\lceil m \log\log(k)/w \rceil)$ worst case time algorithm for string matching under Hamming distance. Alg. 1 shows the pseudo code.

3.1 Matryoshka counters

The above scheme can be improved by using more counter levels. We call these *Matryoshka counters*, to reflect their nested nature. Assume that we use $\ell_1 = 2$ bits in the first level, so this requires $O(\lceil m/w \rceil)$ time per text character. The second level uses $\ell_2 = \ell_1 + 1 = 3$ bits, and so on, in general the level i has $\ell_{i-1} + 1 = i + 1$ bits. The ith level is touched every $2^{\ell_{i-1}} - 1$ steps, and costs $O(\lceil \ell_i m/w \rceil / (2^{\ell_{i-1}} - 1))$ amortized time. The total time is then of the form

$$O\left(\sum_{i}^{\log_2(m)} \frac{\lceil \ell_i m/w \rceil}{2^{\ell_{i-1}} - 1} \right) = O\left((m/w) \sum_{i}^{\infty} \frac{i+1}{2^i - 1} \right) = O(m/w).$$

Hence, the total amortized worst case time is $O(n\lceil m/w \rceil)$.

3.2 Other algorithms

There exist many algorithms based on techniques similar to Shift-Add. For example, the $O(n\lceil m\log(\gamma)/w\rceil)$ worst case time algorithm for (δ,γ)-matching [4] can be improved to $O(n\lceil m\log(\delta)/w\rceil)$ worst case time. In (δ,γ)-matching the differences added are not 0 or 1, but rather $|p_i - t_j|$, whenever this difference is at most δ. Otherwise we add $\gamma+1$. Note that $\delta \le \gamma$. The pattern matches if the accumulated differences do not exceed γ. The number of bits reserved for each counter is therefore $O(\log(\gamma))$. This can be improved by reserving only $O(i\log(\delta))$ bits for a level i in the hierarchy of counters. The only obstacle is how to represent the mismatches. In the original algorithm the value $\gamma+1$ is used, but this is not possible as we do not have enough bits. This can be solved by using a separate "flag" bit-vector to denote the mismatches. The time then becomes

$$O\left(\sum_{i}^{\log_2(\gamma)} \frac{\lceil \ell_i m/w\rceil}{2^{\ell_{i-1}}/\delta}\right) = O\left(\frac{m\log(\delta)}{w}\sum_{i}^{\infty}\frac{\delta(i+1)}{\delta^i}\right) = O\left(\frac{m\log(\delta)}{w}\right)$$

per text character, and $O(n\lceil m\log(\delta)/w\rceil)$ in total.

Another problem example is δ-matching under the Hamming distance. A trivial solution is to modify the Shift-Add algorithm so that the array B is preprocessed with respect to δ-matching of characters. In this way, $O(n\lceil m\log(k)/w\rceil)$ worst case time is achieved. Just as trivially, we can apply our technique to improve the time complexity to $O(n\lceil m/w\rceil)$. Note that in the RAM model of computation this is $O(nm/\log(n))$. We are not aware of any earlier $o(mn)$ algorithms solving this problem.

We believe that many other algorithms can be improved similarly.

4 Conclusions

We have presented improved version of the well–known Shift-Add algorithm, removing its dependence on the parameter k, obtaining optimal parallelization. Our techniques have applications in other algorithms as well.

We have also implemented the simpler of our algorithms, having $O(n\lceil\log\log(k)/w\rceil)$ worst case time, and experimentally compared against plain Shift-Add algorithm using 2.4Ghz P4 computer, having $w=32$. The code was written in C and compiled with `icc 9.1`. If $m\ell \le w$, the Shift-Add is about twice as fast as our algorithm, but if $m\ell' \le w$ and $w < m\ell \le 2w$, then our algorithm is about 40% faster.

References

[1] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. In *Proceedings of the 11th ACM-SIAM Annual Symposium on Discrete Algorithms*, pages 794–803, San Francisco, CA, 2000.

[2] R. A. Baeza-Yates. *Efficient text searching.* Ph. D. Thesis, Department of Computer Science, University of Waterloo, Ontario, 1989.

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

[4] M. Crochemore, C. Iliopoulos, G. Navarro, Y. Pinzon, and A. Salinger. Bit-parallel (δ,γ)-matching suffix automata. *Journal of Discrete Algorithms (JDA)*, 3(2–4):198–214, 2005.

[5] M. J. Fischer and M. Paterson. String matching and other products. In R. M. Karp, editor, *Proceedings SIAM-AMS Complexity of Computation*, pages 113–125, Providence, RI, 1974.

[6] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.

[7] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. Assoc. Comput. Mach.*, 46(3):395–415, 1999.

[8] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.