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

Szymon Grabowski [a] Kimmo Fredriksson [b,*,1]

[a] *Technical University of Łódź, Computer Engineering Department, Al. Politechniki 11, 90–924 Łódź, Poland*

[b] *Department of Computer Science, University of Kuopio, P.O. Box 1627, 70211 Kuopio, Finland*

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 result 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.

Key words: algorithms, approximate string matching, bit-parallelism, Hamming distance

1 Introduction

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

* Corresponding author.
Email address: kimmo.fredriksson@cs.uku.fi (Kimmo Fredriksson).

[1] Supported by the Academy of Finland. This work was partially done while the author worked in the Department of Computer Science and Statistics, University of Joensuu.

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 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, we talk about 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`, $\sim$ 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 an 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:

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

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

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

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 bottom 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 need to effectively add the counters in D and D' as $D + x$. Note that if the counters in D' consume a whole machine word, i.e., w bits, then the counters in D need $O(w \log(k)/\log\log(k))$ bits, and then the simplest solution for computing x is to use look-up tables to do that conversion in $O(\log(k)/\log\log(k))$ time. It may seem it requires $O(2^w)$ space and even more preprocessing time. In the RAM model of computation it is assumed that $w = O(\log_2(n))$, where n, roughly speaking, corresponds to the length of the longest addressable text, and hence we may use e.g. $\log_2(n)/2$ bit words for indexing the table, and construct the final answer from two pieces. The space is then just $2^{\log_2(n)/2} = \sqrt{n}$ words, which is negligible (in the asymptotic sense) compared to the length of the text. In practice we may use e.g. $w/2$ or $w/4$ bit indexes, depending on w. Clearly, transforming (*expanding*) the bit-vector D' cannot be done in constant time, but fortunately it is performed only every $O(\log(k))$ steps, hence in the amortized sense the whole operation is constant-time per input character. For long patterns the cost becomes $O(\lceil m \log\log(k)/w \rceil)$ per character. Observe that this solution assumes that $w = O(\log(n))$. In Sec. 3.1 we show another method not based on precomputation, and hence it removes the above assumption.

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

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

```
1   ℓ' ← ⌈log2(log2(k + 1) + 1)⌉
2   ℓ ← ⌈log2(k + 1 << ℓ')⌉ + 1
3   iv ← 0
4   for i ← 0 to m − 1 do iv ← iv | (1 << (iℓ'))
5   for i ← 0 to σ − 1 do B[i] ← iv
6   for i ← 0 to m − 1 do B[p_i] ← iv ^ (1 << (iℓ'))
7   om ← 0
8   for i ← 0 to m − 1 do om ← om | 1 << ((i + 1)ℓ − 1)
9   D' ← 0; D ← om; j ← 0
10  while j < n do
11      for i ← 1 to 2^ℓ' − 1 do
12          D' ← (D' << ℓ') + B[t_j]
13          if D'_[m−1] + D_[m−1−j mod ℓ'] ≤ k then report match
14          j ← j + 1
15      x ← Expand(D')
16      D ← D << (2^ℓ' − 1)ℓ
17      D ← ((D & ∼om) + x) | (D & om)
18      D' ← 0
```

string matching under Hamming distance. Alg. 1 shows the pseudo code.

3.1 Expanding counters

We now show how to compute $x_{[i]\ell} = D'_{[i]\ell'}$ parallely without precomputed tables. This requires that the counters are arranged in a different way. Let us call the new vector D^*, replacing D'. The counters in D^* are grouped in *blocks* of ℓ bits. Hence each block contains $c = \lfloor \ell/\ell' \rfloor = O(\log(k)/\log\log(k))$ counters, and the total number of blocks is $b = \lceil m/c \rceil$. Note that the total number of bits is still $O(m \log\log(k))$. The possible $\ell - \lfloor \ell/\ell' \rfloor \ell'$ extra bits within each block are unused. The counters have a different arrangement now. Let us denote the ith ℓ'-bit counter in the jth block of D^* as $D^*_{[i,j]\ell'}$. Then $D^*_{[i/b,\, i \bmod b]\ell'} = D'_{[i]\ell'}$. For example, if $c = 4$ and $b = 5$ we use an arrangement

$$(0, 5, 10, 15)(1, 6, 11, 16)(2, 7, 12, 17)(3, 8, 13, 18)(4, 9, 14, 19),$$

where the parentheses represent blocks of ℓ bits, and the numbers denote the permutation of the counters $0 \ldots 19$ within the blocks. The shift is no longer by ℓ' bits, but by ℓ bits, hence we shift a whole block at a time. The last block needs a special treatment; the last counter of the last block is dropped out, and the rest is shifted by ℓ' bits and the block is moved to the first block position (cleared by the shift operation). Hence we obtain

$$(-1, 4, 9, 14)(0, 5, 10, 15)(1, 6, 11, 16)(2, 7, 12, 17)(3, 8, 13, 18),$$

where -1 denotes a new counter introduced by the shift. This does not pose any problems to the rest of the algorithm as long as we permute the preprocessed $B[\cdot]$ vectors in the same way.

The benefit of this permutation is that now D^* is easy to expand without precomputation. Note that the first counter of each block is aligned to start at a ℓ bit boundary, and they are already in the order we use for the D vector. Hence we need only to mask the rest of the counters (bit-parallely) away, and $1/c$ of the work is done. The counters $2 \ldots c$ are obtained in the same way, we shift the D' vector to align each of the counters in the block in turn to the first position, and mask the rest out. The final answer is then a concatenation of the resulting c vectors. The cost of the shifting and masking is $O(1)$ per counter position (assuming that D' fits to w bits), and we have c counter positions for the blocks. Hence the total cost is $O(\log(k)/\log\log(k))$, including the concatenation. But again, this happens only every $O(\log(k))$ steps, so the amortized cost is $o(1)$, not affecting the total time.

3.2 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 $O(n\lceil m/w \rceil)$.

However, the method we used for detecting the occurrences is too costly in this case (i.e. $O(L)$ per text position, where L is the number of counter levels). Our solution is to delay the occurrence reporting. The second highest level counters have $\lfloor \log_2(k) \rfloor + O(1)$ bits, so the last level is touched every $I = O(k)$ steps of the algorithm, and at precisely these steps we can detect the occurrence in $O(1)$ time, examining only one counter. But at this time $I-1$ highest counters positions have been lost. We therefore add $I - 1$ new (high) counter positions for each level, which are shifted and added together as any other counter, but do not accumulate any errors through the $B[\cdot]$ vectors. In other words, these $I - 1$ new counters are just to preserve the accumulated error values without shifting them out. Hence at every Ith step we detect the possible I occurrences for the last I text positions using the "overflow" counter positions. This costs

only $O(1)$ time per text character, even if implemented naïvely. Finally, note that the overflow counters increase the vector lengths only by a constant factor even in the worst case ($k = m - 1$), and hence the time bound is preserved.

Expanding the Matryoshka counters without precomputation is possible as well, using a method similar to that of Sec. 3.1. We do not provide the details here, because of scarcity of space, only the basic idea. This time we use a constant number of bits per counter in the first level, and double it in each next level, i.e. we set $\ell_{i+1} = 2\ell_i$. The initial counter arrangement is simply the identity permutation, and in the second lowest level we use a prebuilt mask to separate the odd and even counter positions, and shift the counters in odd positions to the end of the row of counters (possibly several machine words from the original position). The process for the higher counter levels is similar, but the counters are wider. The invariant of the counter shifting is that the set of counters is divided in three groups only (one of them having always a single counter, the last one, which is dropped out), and the counters of each group are shifted by the same number of bits, i.e., the overall work per input machine word is constant. The summation is analogous to the one above, with a new formula for ℓ_i, which again leads to $O(n\lceil m/w \rceil)$ time.

3.3 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 $\delta \leq \gamma$. 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

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

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 Experimental comparisons and conclusions

We have presented an 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 (with precomputed expand function), 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, where $w = 32$. The code was written in C and compiled with `icc 9.1`.

If $m\ell \leq w$, the Shift-Add is about twice as fast as our algorithm, but if $m\ell' \leq w$ and $w < m\ell \leq 2w$, then our algorithm is about 40% faster. The experiments were run with 4.4MB DNA text (E.coli), and 9.8MB English text (collected works of Charles Dickens) using randomly extracted patterns. Note that the type of the text, nor the value of m or k do not affect the performance of the algorithms (unless increasing m or k induces a use of more computer words), since the number of operations for both algorithms depends only on n. However, in the cases of $m\ell > w$, Shift-Add is typically ran using only a pattern prefix that fits into a single computer word, and whenever a prefix occurrence is found, the pattern suffix is verified using e.g. brute–force. For small k and typical texts this results in $O(n)$ average time algorithm. For such small k values we do not expect our algorithm to be competitive in practice, but we note that the same trick works for our algorithm as well. For our algorithm this trick is usable for larger k values than for Shift-Add, since we can search longer pattern prefixes, and hence the number of verifications and false positives is smaller.

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.