Average-Optimal String Matching

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

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

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

Abstract

The exact string matching problem is to find the occurrences of a pattern of length m from a text of length n symbols. We develop a novel and unorthodox filtering technique for this problem. Our method is based on transforming the problem into multiple matching of carefully chosen pattern subsequences. While this is seemingly more difficult than the original problem, we show that the idea leads to very simple algorithms that are optimal on average. We then show how our basic method can be used to solve multiple string matching as well as several approximate matching problems in average optimal time. The general method can be applied to many existing string matching algorithms. Our experimental results show that the algorithms perform very well in practice.

Key words: string matching, multiple string matching, optimality, bit-parallelism

1 Introduction

We consider several classical string matching problems, given a text $T[0 \ldots n-1]$ over some finite alphabet Σ, of size σ. The *exact string matching* problem is to find the occurrences of a given pattern $P[0 \ldots m-1]$ from T, i.e. all substrings $T[i \ldots i+m-1]$ such that $P[j] = T[i+j]\ \forall\ j \in \{0, \ldots, m-1\}$. *Multiple (exact) string matching* is as above, but instead of a single pattern P, we are given a set $\mathcal{P}$ of r patterns that are matched simultaneously. The *approximate matching* problem with *Hamming distance* allows up to k *mismatches* in each occurrence, i.e. $|\{P[j] = T[i+j] \mid 0 \leq j \leq m-1\}| \geq m-k$.

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

Numerous efficient algorithms solving the exact matching problem have been obtained. The first linear time algorithm (KMP) [1] was given in [30], and the first sublinear average time algorithm (BM) in [7]. Many practical variants of the BM family have been suggested, see e.g. [24,36]. An average optimal $O(n \log_\sigma(m)/m)$ time algorithm is obtained e.g. in [13] (BDM). Recently bit-parallelism has been shown to lead to the most efficient algorithms for relatively short patterns, in practice. The first algorithm in this class was Shift-Or [15,5,37], which runs in $O(n\lceil m/w \rceil)$ time, where w is the number of bits in a computer word (typically 32 or 64). Shift-Or is extremely simple to implement, and can be easily adapted to more complex search problems; common properties for most of the bit-parallel algorithms. Currently, among the fastest algorithms in practice (for $m \leq w$) are BNDM [33] and SBNDM [32,35]. BNDM is bit-parallel version of BDM, and SBNDM is a simplified version of BNDM. Their common feature is combining bit-parallelism with skipping characters, in the manner of the BM family of algorithms [7].

For multiple matching, the Aho–Corasick algorithm [1] (which can be seen as a generalization of KMP) achieves $O(n)$ (or $O(n \log(\sigma))$, depending on implementation details) time. The Commentz–Walter algorithm [10] is a generalization of the BM algorithm for multiple patterns. This is also sublinear on average, for small pattern sets. BDM algorithm can also be generalized for multiple patterns, resulting in an algorithm that is optimal in both worst and average cases [13].

For the problem of matching under Hamming distance, the asymptotically most efficient algorithms are based on convolutions and Fast Fourier Transform, and one of them achieves $O(n\sqrt{k \log k})$ worst case time [4]. This is a major improvement over a trivial brute-force algorithm which solves the problem in $O(nm)$ worst case time. Still, more practical algorithms for this problem are based on bit-parallelism or filtering. A classic example based on bit-parallelism is the Shift-Add algorithm [5], which 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). This algorithm was recently refined to achieve $O(n\lceil m/w \rceil)$ worst case time, i.e., an optimal parallelization [21]. Besides the algorithms that have good worst case complexity, there is a number of filtering based algorithms [31] 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. The average complexity of this prob-

[1] As a historical remark, we note that essentially the same algorithm was obtained several years earlier by Gosper [6, item 179], although no analysis was given.

lem is $O(n(k + \log_\sigma(m))/m)$, which was shown by Chang and Marr [9] (their result for the edit distance can trivially be translated also to this problem), and a real algorithm with this average complexity was given in the same work. Another average-optimal algorithm was presented in [19].

For more references, see e.g. [13,14,34].

1.1 Our Contributions

In this paper we first introduce a new general technique for skipping text characters, and then apply it to the problem variants mentioned above. Basically, our idea is to translate the given problem into matching a (sparse) subsequence of the text against several sparse subsequences of the pattern. We can say that we transform the original problem into a multiple pattern matching problem. While this may seem more difficult than originally, in fact this idea leads to very simple algorithms that are often optimal on average. Our "building blocks" for the resulting multiple matching problems are based either on bit-parallelism or the classic Aho–Corasick automaton.

Our achievements can be summarized as follows. For the problem of exact string matching we propose an average-optimal (for short patterns) variant of the well-known bit-parallel algorithm, Shift-Or, and we also discuss its practical implementation. Then we apply the same idea to the naïve (brute-force) algorithm; while the obtained algorithm is no longer average-optimal, it is arguably one of the simplest known character-skipping string matching algorithms. Finally, we notice how our technique can be used to generate subpatterns for the Aho–Corasick multiple matching algorithms, which leads to an average-optimal algorithm without any limitation on the pattern length. This solution can be trivially generalized to r input patterns, again with optimal time complexity on average. Bit-parallelism and brute-force search are then used, together with our pattern-splitting idea, for matching under Hamming distance, and the first algorithm is again average-optimal for short patterns. We also consider three other approximate matching problems, where in all cases we reached optimality on average (at least for short patterns).

To expose the potential of our technique, we tested the algorithms for exact string matching (for one or many patterns) and matching under Hamming distance, using various texts (DNA, proteins, English text, random data with alphabet size of 96 characters) and on three different hardware platforms. The experiments showed that our algorithms are very competitive, and their advantage is typically greatest on the most modern of the tested platforms, equipped with Intel Core 2 Duo CPU. For exact string matching for a single pattern the best choice is Average-Optimal Shift-Or, while for multiple patterns the winner is Average-Optimal Aho–Corasick.

The organization of the paper is as follows. Sec. 1.2 presents the notation used throughout the paper. Sec. 2 shows our technique on a high level. The following three sections deal with particular incarnations of this idea in well-known exact string matching algorithms: Shift-Or, brute-force, and Aho–Corasick algorithm, respectively. In Sec. 6 two algorithms for matching under Hamming distance are presented, and in Sec. 7 three other approximate matching problems are considered. Sec. 8 contains results of experiments on real and artificial data, concentrated (but not limited to) the average-optimal Shift-Or algorithm. The last section concludes.

A preliminary version of this paper appeared in [17].

1.2 Preliminaries

We use the following notation. The *pattern* is $P[0 \ldots m-1]$ and the *text* is $T[0 \ldots n-1]$. The symbols of P and T are taken from some finite alphabet Σ, of size σ. 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, $<<$ is shift to left, and $>>$ shift to right, both with zero padding. For brevity, we make the assumption that $m \le w$, unless explicitly stated otherwise. All the logarithms are of base 2, if not stated otherwise.

2 New Filtering Technique

We start with showing a general technique of how to skip text characters, with any (linear time) string matching algorithm that can search for multiple patterns simultaneously. The method 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 $m' = \lfloor m/q \rfloor$, as follows:

$$P^j[i] = P[j + iq], \quad j = 0 \ldots q-1, \quad i = 0 \ldots \lfloor m/q \rfloor - 1.$$

In other words, we generate q different alignments of the original pattern P, each alignment containing only every qth character. The total length of the patterns P^j is $q\lfloor m/q \rfloor \le m$. For example, if $P =$ `abcdef` and $q = 3$, then $P^0 =$ `ad`, $P^1 =$ `be` and $P^2 =$ `cf`.

Assume now that P occurs at $T[i \ldots i+m-1]$. From the definition of P^j it

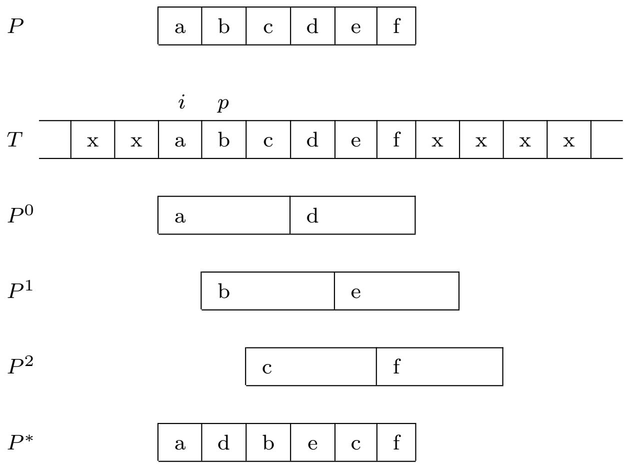

Fig. 1. An example. Assume that $P = \texttt{abcdef}$ occurs at text position $T[i \ldots i+m-1]$, and that $q = 3$. The current text position is $p = 10$, and $T[p] = \texttt{b}$. The next character the algorithm reads is $T[p+q] = T[13] = \texttt{e}$. This triggers a match of $P^{p \bmod q} = P^1$, and the text area $T[p-1 \ldots p-1+m-1] = T[i \ldots i+m-1]$ is verified.

directly follows that

$$P^j[h] = T[i + j + hq], \quad j = i \mod q, \quad h = 0 \ldots m' - 1.$$

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. Fig. 1 illustrates.

The occurrences of the patterns in $\mathcal{P}$ can be searched for simultaneously using any multiple string matching algorithm. Assuming that the selected string matching algorithm runs generally in $O(n)$ time, then the filtering time becomes $O(n/q)$, as only every qth symbol of T is read. The filter searches for the exact matches of q patterns, each of length $\lfloor m/q \rfloor$. Assuming that each character occurs with probability $1/\sigma$, the probability that P^j occurs (triggering a verification) in a given text position is $(1/\sigma)^{\lfloor m/q \rfloor}$. A brute force verification cost is in the worst case $O(m)$. To keep the total time at most $O(n/q)$ on average, we select q so that $nm/\sigma^{m/q} = O(n/q)$. This is satisfied for $q = m/(2 \log_\sigma(m))$, where the verification cost becomes $O(n/m)$ and filtering cost $O(n \log_\sigma(m)/m)$. The total average time is then dominated by the filtering time, i.e. $O(n \log_\sigma(m)/m)$, which is optimal [38].

In the following sections, we describe efficient applications of the above scheme for several string matching problems.

2.1 Relaxing q

The performance of all the algorithms we are going to present depends on the choice of q. Our analyses assume uniform distribution of characters, which is

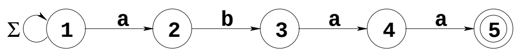

Fig. 2. NFA recognizing the pattern `abaa`.

not a problem in most practical cases. For instance, good results for English language can be obtained by assuming uniform distribution for $\sigma \approx 16$ [34]. The real problem is that for some texts the distribution could change abruptly, and thus there is no single optimal q.

If this becomes an issue, we can proceed as follows (this will work for all our algorithms):

- Compute the value of q assuming uniform distribution.
- Do all the precomputations for $q' \in \{1, 2, 3, \ldots, q, \ldots, m\}$.
- Initialize $q' = q$, and start searching, using q'.
- During the search, if the verification algorithm is reading significantly more characters than the filtering algorithm, then decrease q'.
- In the opposite case, i.e., if the filtering algorithm is reading significantly more characters than the verification algorithm, then increase q'.

The bookkeeping of the above is simple, and adds only a constant factor overhead to the whole algorithm.

3 Shift-Or

The Shift-Or algorithm [15,5] can be seen as a simulation of a nondeterministic automaton, constructed as follows. The automaton has states $0, 1, \ldots, m$. The state 0 is the initial state, state m is the final (accepting) state, and for $i = 0, \ldots, m-1$ 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 and 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 $0 \leq i \leq m-1$, 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. Fig. 2 illustrates.

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 is set to 1. For each text symbol c the vector is updated by $D \leftarrow (D << 1) \mid B[c]$. 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

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

```
for $i \leftarrow 0$ to $\sigma - 1$ do $B[i] \leftarrow \sim 0$
for $i \leftarrow 0$ to $m - 1$ do $B[P[i]] \leftarrow B[P[i]]$ & $\sim(1 << i)$
$D \leftarrow \sim 0; mm \leftarrow 1 << (m-1); i \leftarrow 0$
while $i < n$ do
    $D \leftarrow (D << 1) \mid B[T[i]]$
    if $(D$ & $mm) \neq mm$ then report match
    $i \leftarrow i + 1$
```

Alg. 2 Average-Optimal-Shift-Or(T, n, P, m, q).

```
for $i \leftarrow 0$ to $\sigma - 1$ do $B[i] \leftarrow \sim 0$
$h \leftarrow 0$; $mm \leftarrow 0$
for $j \leftarrow 0$ to $q - 1$ do
    for $i \leftarrow 0$ to $\lfloor m/q \rfloor - 1$ do
        $B[P[iq + j]] \leftarrow B[P[iq + j]]$ & $\sim(1 << h)$
        $h \leftarrow h + 1$
    $mm \leftarrow mm \mid (1 << (h - 1))$
$D \leftarrow \sim 0; i \leftarrow 0$
while $i < n$ do
    $D \leftarrow ((D$ & $\sim mm) << 1) \mid B[T[i]]$
    if $(D$ & $mm) \neq mm$ then VerifyAOSO$(T, i, n, P, m, q, D, mm)$
    $i \leftarrow i + q$
```

occurrence of P. Alg. 1 gives the code. If $m = O(w)$, then the algorithm runs in time $O(n)$.

3.1 Optimal Shift-Or

The set of patterns can be searched for simultaneously using the Shift-Or algorithm, as long as $qm' \leq w$. All the patterns are preprocessed together, as if they were concatenated. For our example pattern, $P =$ `abcdef`, we effectively preprocess a pattern $P^* = P^0\ P^1\ P^2 =$ `adbecf`. Alg. 2 gives the code for preprocessing and filtering algorithms. If the pattern P^j matches, then the $(j + 1)m'$-th bit in D is zero. This is detected with $(D$ & $mm) \neq mm$, where mm has every $(j + 1)m'$-th bit set to 1. These bits have also to be cleared in D before the shift operation $(D$ & $\sim mm)$, to correctly initialize the first bit corresponding to each of the successive patterns.

Whenever an occurrence of P^j is found in the text, we must verify if P also occurs, with the corresponding alignment. To efficiently detect which patterns in $\mathcal{P}$ match, we first set $D \leftarrow (D$ & $mm)$ ^ mm, i.e. the $(j + 1)m'$-th bit in D is now one if P^j matches, and all other bits are zero. Now $s \leftarrow \lfloor \log_2(D) \rfloor$ gives the index of the highest bit set in D, and therefore j is $\lfloor s/m' \rfloor$, which is our alignment offset, see Fig. 1. The corresponding text position is then

Alg. 3 VerifyAOSO$(T, i, n, P, m, q, D, mm)$.

```
$D \leftarrow (D \mathbin{\&} mm) \wedge mm$
while $D \neq 0$ do
    $s \leftarrow \lfloor \log_2(D) \rfloor$
    $c \leftarrow -(\lfloor m/q \rfloor - 1)\ q - \lfloor s / \lfloor m/q \rfloor \rfloor$
    if $P[0 \ldots m-1] = T[i+c \ldots i+c+m-1]$ then report match
    $D \leftarrow D \mathbin{\&} {\sim}(1 << s)$
```

verified. Finally, we clear the bit s in D. This is repeated until D becomes zero, indicating that there are no more matches. Note that computing $\lfloor \log_2(x) \rfloor$ can be done very efficiently in modern computers, e.g. by casting x to real number, and extracting the exponent from the standardized floating point representation. Alg. 3 gives the verification code. By using the optimal q, the algorithm clearly runs in $O(n \log_\sigma(m)/m)$ average time.

3.2 Handling longer patterns

If $qm' > w$, we must use more computer words, and the running time must be multiplied by $O(\lceil qm'/w \rceil) = O(\lceil m/w \rceil)$, i.e. the average time becomes $O(n \log_\sigma(m)/w)$.

However, the trick used in [35] to make BNDM work with $m > w$ can be applied to our algorithms too. The idea is to partition the pattern into $r = \lfloor m/h \rfloor$ consecutive parts. The length of each part is now $h = \lfloor (m-1)/w \rfloor + 1$. All the h characters of each part are then superimposed into a single "supercharacter". The resulting r supercharacters are then concatenated to form a single pattern of length r. This pattern fits into a single computer word, and it can be searched for by reading only every hth character of the text. This turns any algorithm, where it is applied to, into a filter, so the potential matches must be verified. See [35] for more details. This technique permits long patterns for the average optimal Shift-Or as well. The result is an algorithm with $O(n \log_{\sigma/h}(m)/m)$ time on average. For $m < w\sigma^\varepsilon$, where ε is a constant, $0 < \varepsilon < 1$, the complexity becomes $O(\frac{1}{1-\varepsilon} n \log_\sigma(m)/m)$, which is still optimal up to a constant factor. In practice the technique should work well for $\sigma \gg h$.

3.3 Linear worst case time

The worst case running time of Alg. 2 is $O(nm)$. However, the verification algorithm is easy to combine with standard Shift-Or, so that the verifications take at most $O(n)$ total time. This is done as follows. Whenever we must verify a pattern occurrence, we do it with Shift-Or. The last text position verified is saved in a variable, as well as the state vector D (for plain Shift-Or). If

Alg. 4 Fast-Shift-Or(T, n, P, m, U).

```
for $i \leftarrow 0$ to $\sigma - 1$ do $B[i] \leftarrow ((1 << m) - 1) << (w - U - m)$
for $i \leftarrow 0$ to $m - 1$ do $B[P[i]] \leftarrow B[P[i]]$ & $\sim(1 << (w - U - m + i))$
$D \leftarrow \sim 0; i \leftarrow 0$
while $i < n$ do
    for $r \leftarrow 0$ to $U - 1$ do $D \leftarrow (D << 1) \mid B[T[i + r]]$
    if $\sim D >> (w - U) \neq 0$ then report matches
    $i \leftarrow i + U$
```

the next verification area overlaps with the previous, we restore the Shift-Or search state from the previous verification. Otherwise, if the next verification area starts after the previous ended, we reinitialize the Shift-Or search state. The verification algorithm then reads every text character at most once, and therefore the time is at most $O(n)$ (or $O(n\lceil m/w \rceil)$ for long patterns). However, if the verification time becomes an issue, the filter does not work well, and one could use plain Shift-Or just as well.

3.4 Implementation

In modern pipelined CPUs branching is costly. In Alg. 1 there are two conditionals in the search code; to detect the matches, and to check the end of the input. A simple way to avoid these to some degree is to unroll the line 5, i.e. repeat the code

$$D \leftarrow (D << 1) \mid B[T[i]]$$

inline several, say U, times (with increasing offsets for the variable i). This means that the bit $m - 1$ of D, indicating an occurrence, will be overflowed due to the repeated shifts, and hence in line 6 we must detect if any of the bits $m - 1 \ldots m + U - 1$ is zero. This means that we need $U - 1$ extra bits, and the pattern length is therefore limited to $m \leq w - U + 1$.

The second optimization involves detecting the matches. Line 6 in Alg. 1 involves a variable mm. This can be avoided if the bit-vectors are aligned so that the highest bit is in position $w - U + 1$, instead of in position $m + U - 1$. This means that the matches can be detected with $\sim D >> (w - U) \neq 0$, which is efficient if U is constant.

These two simple optimizations (shown in Alg. 4) give about $2 - 5\times$ speed-up for standard Shift-Or (Alg. 1), depending on the architecture. The line 5 in Alg. 4 is automatically inlined by compilers, for small constant U. Although the speed-up is considerable, note that this can depend on the architecture.

Unrolling speeds-up also the Optimal Shift-Or, but the second optimization cannot be applied in this case, since the bit positions indicating the matches are not consecutive. The unrolling technique uses $U - 1$ extra bits per pattern,

Alg. 5 Fast-Average-Optimal-Shift-Or(T, n, P, m, q, U).

```
for $i \leftarrow 0$ to $\sigma - 1$ do $B[i] \leftarrow \sim 0$
$h \leftarrow 0$; $mm \leftarrow 0$
for $j \leftarrow 0$ to $q - 1$ do
    for $i \leftarrow 0$ to $\lfloor m/q \rfloor - 1$ do
        $B[P[iq + j]] \leftarrow B[P[iq + j]]$ & $\sim(1 << h)$
        $h \leftarrow h + 1$
    for $r \leftarrow 0$ to $U - 1$ do
        $mm \leftarrow mm \mid (1 << (h - 1))$
        $h \leftarrow h + 1$
    $h \leftarrow h - 1$
for $i \leftarrow 0$ to $\sigma - 1$ do $B[i] \leftarrow B[i]$ & $\sim(mm$ & $(mm << 1))$
$D \leftarrow \sim mm; i \leftarrow 0$
while $i < n$ do
    for $r \leftarrow 0$ to $U - 1$ do $D \leftarrow (D << 1) \mid B[T[i + rq]]$
    if $(D$ & $mm) \neq mm$ then VerifyAOSO'$(T, i, n, P, m, q, U, D)$
    $D \leftarrow D$ & $\sim mm$
    $i \leftarrow i + Uq$
```

so we need $q(U-1+\lfloor m/q \rfloor)$ bits in total, which is $O(m(U+\log_\sigma(m))/\log_\sigma(m))$ with the optimal q. Note that we must conceptually add $U - 1$ wild cards to the end of each piece P^i, so that the bits denoting the matches are preserved; i.e., the symbols $P^i[m'] \ldots P^[m' + U - 2]$ should match every symbol in the alphabet. This is implemented by zeroing the corresponding $U - 1$ bits in the B table. Alg. 5 gives the code.

Finally, observe that while unrolling is well suited to Shift-Or, the benefits are negligible e.g. for BNDM algorithm, since the more complex control logic cannot be avoided.

4 Almost Optimal Bit-Parallel Naïve Search

Consider the basic brute-force search algorithm: align P against the current text window, $T[k \ldots k+m-1]$; compare the characters of P from left to right against the text window until first mismatch, or the whole pattern is matched; shift the text window to right by one position. This algorithm is extremely simple, yet it runs in $O(n)$ average time, albeit the worst case complexity is $O(nm)$.

We now show that by using our pattern partitioning technique, combined with bit-parallelism, we can improve the above naïve and slow search algorithm. Recall that we partition P to q patterns, $P^0, \ldots, P^{q-1}$, where $P^i[j] = P[i+jq]$.

The preprocessing algorithm builds a two-dimensional table B of bit-vectors,

Alg. 6 Almost-Average-Optimal-Naïve(T, n, P, m, q).

```
for $j \leftarrow 0$ to $\lfloor m/q \rfloor - 1$ do
    for $i \leftarrow 0$ to $\sigma - 1$ do $B[j][i] \leftarrow 0$
    for $i \leftarrow 0$ to $q - 1$ do
        $c \leftarrow P[i + j \times q]$
        $B[j][c] \leftarrow B[j][c] \mid (1 << i)$
$i \leftarrow 0$
while $i < n - m + 1$ do
    $D \leftarrow \sim 0$; $j \leftarrow 0$
    while $j < \lfloor m/q \rfloor$ AND $D \neq 0$ do
        $D \leftarrow D \ \& \ B[j][T[i + j \times q]]$
        $j \leftarrow j + 1$
    if $j = \lfloor m/q \rfloor$ then VerifyAAON$(T, i, n, P, m, q, D)$
    $i \leftarrow i + q$
```

so that the ith bit of $B[j][c]$ is 1 iff the jth symbol of $P^i = c$. Using B we can easily search for all the q patterns simultaneously with the above brute-force method. That is, assume that the pattern is aligned against the text window $T[k \ldots k + m - 1]$. We use a bit-vector D of q bits, initialized to all 1s. The invariant is that the ith bit of D is 1, if the currently read text string matches (the prefix of) the pattern P^i. Hence the state vector D is updated simply as

$$D \leftarrow D \ \& \ B[j][T[k + j \times q]],$$

for $j = 0, 1, 2, \ldots$, until D becomes all zeros, or $j = \lfloor m/q \rfloor - 1$. If D becomes all zeros, then no pattern in the set matches in the current window. In the opposite case, if j becomes $\lfloor m/q \rfloor - 1$, then some patterns may match, and verification is needed. In either case, the window is shifted by q positions. The verification alignments are easily read from D; if the ith bit of D is set, then P^i matches, and the corresponding alignment must be verified. Alg. 6 shows the pseudocode.

Now consider the average case running time of the algorithm. The analysis is similar to the baseline method, see Sec. 2. The difference here is that one window position is not handled in constant time; therefore the filtering time becomes $O(n/q \times m') = O(nm/q^2)$ in the *worst case*, i.e. the filter reads $m' = \lfloor m/q \rfloor$ symbols in each window. We want to select q so that this dominates the verification time, $O(nm/\sigma^{\lfloor m/q \rfloor})$, that is $nm/\sigma^{\lfloor m/q \rfloor} = O(nm/q^2)$. The solution is again $q = m/(2\log_\sigma(m))$, and the total average time becomes $O(n \log_\sigma^2(m)/m)$. This is worse than for the Shift-Or variant, but it has the property of working for longer patterns using just a single computer word, namely for $q = O(w)$, i.e. for $m/\log_\sigma(m) = O(w)$. This means that the asymptotic running time is $O(n \log_\sigma(m)/w)$, which is the same as for Shift-Or for large m.

5 Aho–Corasick

The Aho–Corasick (AC) algorithm [1] is a multiple string matching algorithm that runs in $O(n)$ worst case time. Our application of the AC technique may serve both for single and multiple exact string matching. We briefly review how the algorithm works, for more details refer to [1]. The algorithm builds a finite state automaton recognizing the input pattern set. Basically, the automaton is a trie of all the r patterns, augmented with "fail" transitions. Hence the automaton has $O(rm)$ states. Let $label(s)$ be the label (substring) spelled out by the path from the initial state (root of the trie) to state s (a node of the trie). For a state s, the fail transition leads to state (node in the trie) s', such that $label(s')$ is the longest suffix of $label(s)$ that is also a prefix of some pattern in the set. The resulting automaton can be used to search for every occurrence of the stored patterns from text T; the text symbols are read one by one, and for each symbol we advance using the trie transitions if a matching transition exists, if not, we follow the fail transitions until we reach the initial state, or a matching transition is found. If the automaton goes through a node that corresponds to a stored pattern, an occurrence is found. It is easy to see that the whole process takes only $O(n)$ steps.

Note that for each "fail" transition and alphabet symbol we can precompute the state where the symbol leads. This complicates the preprocessing, but searching algorithm becomes simpler and more efficient (in practice) as every state has an outgoing transition for every alphabet symbol and fail transitions become obsolete. However, the space becomes $O(rm\sigma)$.

Alg. 7 shows the pseudocode for preprocessing. The code builds the trie breadth-first, and simultaneously builds the full automaton directly. In the end, state 0 is the initial state. Each state s has a transition with symbol c stored as $AC.\delta[s][c]$. The set $AC.id[s]$ stores the set of pattern numbers that match for the state s.

5.1 Optimal Aho–Corasick

Again recall that we partition P to q patterns, $P^0, \ldots, P^{q-1}$. It should be clear that we can search for each P^i using AC, and adapting it for skipping text symbols can be done precisely as for Shift-Or. That is, the automaton is built for the q patterns P^i, and only every qth text symbol is read. If some P^i occurs in the text, we invoke verification. The analysis is also the same as for Shift-Or, i.e. the average time becomes $O(n \log_\sigma(m)/m)$. The only difference is that no assumption is made on the pattern length.

We note that we can apply our technique to any number of patterns simulta-

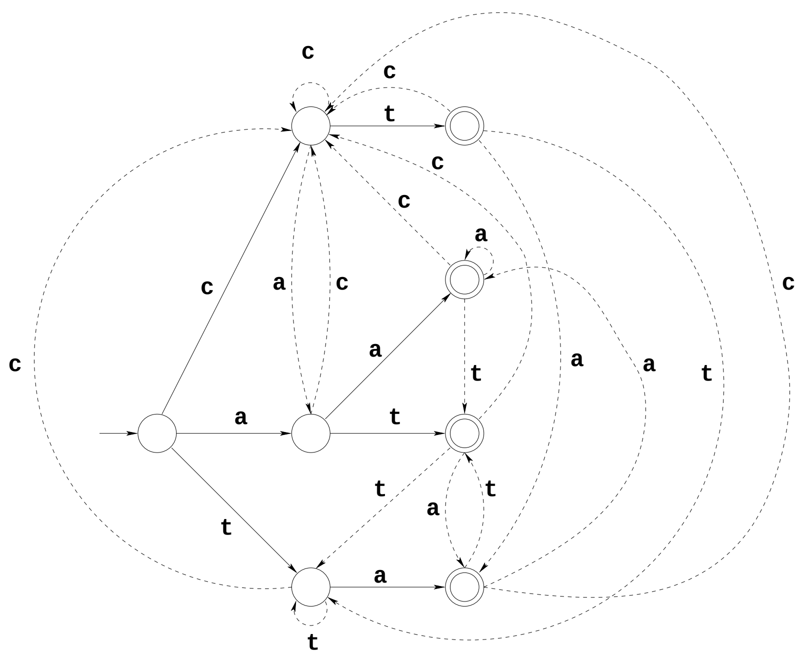

Fig. 3. The full AC-automaton for patterns $\{\texttt{atataa}, \texttt{acatta}\}$, using the parameter value $q = 3$, giving the pattern (multi-)set $\{\texttt{at}, \texttt{ta}, \texttt{aa}, \texttt{at}, \texttt{ct}, \texttt{aa}\}$. The solid arrows correspond to the trie of the set.

neously, as AC can search for any number of patterns. (Obviously, the same is true for the bit-parallel algorithms as well, but in practice the number of bits is too small.) The algorithm itself does not change much, the partitioning technique is simply applied to all the r given patterns, and searched for together. However, the verification probability increases, i.e. it is multiplied by r, and hence we must choose $q = O(m/\log_\sigma(rm))$, resulting in $O(n\log_\sigma(rm)/m)$ average time, which is again optimal. Fig. 3 illustrates the automaton, Alg. 8 gives the pseudocode for filtering, and Alg. 9 for verification.

The number of states is still $O(rm)$ in the worst case, as in the case of standard AC automaton. However, if r is large as compared to σ, many pattern pieces can share the same prefix, which reduces the number of states in practice. In our case we have qr prefixes, and the pattern lengths are only $\lfloor m/q \rfloor$, and hence in practice the automaton has fewer states than plain AC.

Finally, we note that the worst case time can be improved from the $O(nmr)$ to just $O(n)$, by using the same trick as for Shift-Or (see Sec. 3.3). That is, it is simple to use standard AC algorithm for the verifications, and by saving the search state the total worst case time can be made $O(n)$.

Alg. 7 Build-AC($\mathcal{P}, r$).

```
State ← 0
queue1 ← 1; queue2 ← 0
for $i ← 0$ to $r - 1$ do $L[i] ← 0$; $ps[i] ← 0$
done ← false
while NOT(done) do
    done ← true
    for $i ← 0$ to $r - 1$ do
        if $L[i] < |\mathcal{P}^i|$ then
            done ← false
            $c ← \mathcal{P}^i[L[i]]$
            if $AC.\delta[ps[i]][c]$ = fail then
                State ← State + 1
                $AC.\delta[ps[i]][c] ←$ State
            $ps[i] ← AC.\delta[ps[i]][c]$
            $L[i] ← L[i] + 1$
            if $L[i] = |\mathcal{P}^i|$ then $AC.id[ps[i]] ← AC.id[ps[i]] \cup \{i\}$
    while queue1 ≤ queue2 do
        for $c ← 0$ to $\sigma - 1$ do
            $s ← AC.\delta[queue1][c]$
            if $s \neq$ fail then
                $AC.fail[s] ← AC.\delta[AC.fail[queue1]][c]$
                $AC.id[s] ← AC.id[s] \cup AC.id[AC.fail[s]]$
            else
                $AC.\delta[queue1][c] ← AC.\delta[AC.fail[queue1]][c]$
        queue1 ← queue1 + 1
    queue2 ← State
return AC
```

Alg. 8 Average-Optimal-AC(T, n, P, r, m, q).

```
for $i ← 0$ to $r - 1$ do
    for $j ← 0$ to $q - 1$ do
        for $k ← 0$ to $\lfloor m/q \rfloor - 1$ do
            $\mathcal{P}^{iq+j}[k] ← P^i[kq + j]$
$AC ←$ Build-AC($\mathcal{P}, rq$)
$s ← 0; i ← 0$
while $i < n$ do
    $s ← AC.\delta[s][T[i]]$
    if $AC.id[s] \neq \emptyset$ then VerifyAOAC($T, i, n, P, m, q, AC, s$)
    $i ← i + q$
```

6 Approximate Matching for the k-Mismatches Problem

In this section we present two bit-parallel algorithms for approximate string matching.

Alg. 9 VerifyAOAC$(T, i, n, P, m, q, AC, s)$.

```
1   for $i \leftarrow 0$ to $|AC.id[s]| - 1$ do
2       $id \leftarrow \lfloor AC.id[s][i]/q \rfloor$
3       $os \leftarrow AC.id[s][i] \mod q$
4       $c \leftarrow -(\lfloor m/q \rfloor - 1)\ q - os$
5       if $P^{id}[0 \ldots m-1] = T[i+c \ldots i+c+m-1]$ then report match
```

6.1 Optimal Shift-Add

Shift-Add [5] is a bit-parallel algorithm for approximate searching under Hamming distance, i.e. it allows at most k mismatches of pattern characters in the occurrences. Shift-Add is very similar to Shift-Or. Shift-Or reserves only one bit per pattern character in the state vector D. If some bit is 0 in the vector, it means that the corresponding pattern prefix matches with 0 mismatches the current text position, while bit 1 means that the prefix matches with one or more mismatches. This is possible to extend to allow k mismatches by reserving $\ell = \lceil \log_2(k+1) \rceil + 1$ bits for each character, and replacing the OR operation with addition operation [5].

More precisely, the ith ℓ-bit field in $B[c]$ is ℓ bit binary number 0, if the ith character of P matches the character c, and 1 otherwise. Then we can accumulate the mismatches as

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

If the mth field of D has a value less than $k+1$, 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 $\ell = O(\log(k))$ 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:

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

The bit mask om has bit 1 in the highest bit position of each ℓ-bit field, and 0s elsewhere. Note that if o has bit 1 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 mth field. Shift-Add clearly works in $O(n)$ time, if $m(\lceil \log_2(k+1) \rceil + 1) = O(w)$.

Our method of skipping text characters with Shift-Or clearly works with Shift-Add as well. The pattern is again splitted to q partitions. If some of our q patterns occur with at most k mismatches, then we verify if the whole pattern occurs with at most k mismatches. Note that this is different from most of the other pattern partitioning based approaches, that partition the pattern into q' pieces, and then search for the pieces with $\lfloor k/q' \rfloor$ errors. This latter approach leads to $O(nk \log_\sigma(m)/m)$ average time in general, and works for $k = O(m/\log_\sigma(m))$. This time is not optimal, whereas our approach leads to $O(n(k + \log_\sigma(m))/m)$ optimal average time, see below.

Adapting the Shift-Add algorithm to multiple patterns requires some modifications on the preprocessing and searching algorithms. The problem is how to detect the matches of several patterns in parallel. This is solved by initializing the counters to $2^{\ell-1} - (k+1)$, instead of to zero. This trick has been used before, e.g. in [12]. This ensures that the overflow bit is activated immediately when the counter reaches a value $k+1$, and is therefore easy to detect for all patterns in parallel. This could be implemented explicitly, by setting the first field in D of each pattern to this value after the shift operation. Instead, we add $2^{\ell-1} - (k+1)$ to all fields of the $B[c]$ vectors that correspond to the first character of each of the patterns. This ensures that the counters in D get correctly initialized, assuming the first counters of each pattern were zero before the addition. This zeroing is done explicitly with a bit mask. Alg. 10 gives the code.

Consider now the average case time. The analysis is similar to that of ABNDM [26]. The filtering time is again $O(n/q)$, assuming that $m\log(k) = O(w)$. Recall that we have pattern subsequences of length $m' = \lfloor m/q \rfloor$, that is, the filtering time is $O(nm'/m)$, and we require that $q \geq 2$. The probability that such a piece matches a given text position with at most k mismatches is $O(\gamma^{m'}/\sqrt{m'})$ [20], where $\gamma < 1$, whenever $k/m' < 1 - e/\sigma$. This gives the condition that $m' > k/(1 - e/\sigma)$. There are q pieces and $O(n/q)$ possibilities to trigger a verification corresponding to one of the pieces. The cost of the verification is $O(m)$ in the worst case. Hence the total verification cost is on average $O((\gamma^{m'}/\sqrt{m'})mqn/q) = O(nm\gamma^{m'}/\sqrt{m'})$. By choosing m' large enough, we can guarantee that this does not dominate the filtering cost. In particular, we require that verifications take at most $O(n/m)$ total time on average, i.e. $\gamma^{m'}/\sqrt{m'} \leq 1/m^2$. More strict condition is to require that $\gamma^{m'} \leq 1/m^2$, giving the condition that $m' \geq 2\log_{1/\gamma}(m)$. On the other hand,

$$\gamma(\alpha) \; = \; \frac{1}{\sigma^{1-\alpha}\alpha^\alpha(1-\alpha)^{1-\alpha}},$$

where $\alpha = k/m'$ [20]. For $\alpha = 0$ we get $\gamma = 1/\sigma$. Notice that $\gamma(\alpha)$ is monotonically increasing on α in the range $0 \leq \alpha \leq 1/2$, and hence γ is maximized with $\alpha = 1/2$ (in that range, for larger α the algorithm cannot be sublinear),

Alg. 10 Average-Optimal-Shift-Add(T, n, P, m, q, k).

```
$\ell \leftarrow \lceil \log_2(k+1) \rceil + 1$
$iv \leftarrow 0$
**for** $i \leftarrow 0$ **to** $m-1$ **do** $iv \leftarrow iv \mid (1 << (i\ell))$
**for** $i \leftarrow 0$ **to** $\sigma - 1$ **do** $B[i] \leftarrow iv$
$iv \leftarrow (1 << (\ell - 1)) - (k+1)$
$h \leftarrow 0$; $mm \leftarrow 0$; $hm \leftarrow 0$; $om \leftarrow 0$
**for** $j \leftarrow 0$ **to** $q-1$ **do**
    **for** $i \leftarrow 0$ **to** $\lfloor m/q \rfloor - 1$ **do**
        $B[P[iq+j]] \leftarrow B[P[iq+j]]$ ^ $(1 << h)$
        $h \leftarrow h + \ell$
    $hm \leftarrow hm \mid (((1 << \ell) - 1) << (h - \ell))$
    $mm \leftarrow mm \mid (1 << (h-1))$
    $iv \leftarrow iv \mid (iv << h)$
**for** $i \leftarrow 0$ **to** $\sigma - 1$ **do** $B[i] \leftarrow B[i] + iv$
**for** $i \leftarrow 0$ **to** $\lfloor m/q \rfloor q - 1$ **do** $om \leftarrow om \mid (1 << (((i+1)\ell) - 1))$
$D \leftarrow 0; o \leftarrow om; i \leftarrow 0$
**while** $i < n$ **do**
    $D \leftarrow (D << \ell) + B[T[i]]$
    $o \leftarrow (o << \ell) \mid (D \mathbin{\&} om)$
    $D \leftarrow D \mathbin{\&} {\sim}hm \mathbin{\&} {\sim}om$
    **if** $(o \mathbin{\&} mm) \neq mm$ **then** VerifyAOSA($T, i, n, P, m, q, k, o, mm$)
    $o \leftarrow o \mathbin{\&} {\sim}hm$
    $i \leftarrow i + q$
```

giving $\gamma \leq 1/(\sigma^{1-1/2}(1/2)^{1/2}(1-1/2)^{1-1/2}) = 2/\sqrt{\sigma}$. Thus it holds that

$$1/\sigma \leq \gamma \leq 2/\sqrt{\sigma} \quad \Longleftrightarrow \quad \sqrt{\sigma}/2 \leq 1/\gamma \leq \sigma$$

for $\sigma > e/(1 - k/m')$, and it is sufficient that [2]

$$m' \geq 2\log_{1/\gamma}(m) = \frac{2\log_\sigma(m)}{\log_\sigma(1/\gamma)} = \Theta(\log_\sigma(m)).$$

Combining the bounds we obtain $m' > \max(k/(1-e/\sigma), 2\log_{1/\gamma}(m)) = \Theta(k + \log_\sigma(m))$, and $k/m < (1/2)(1 - e/\sigma)$. Noticing that we want the smallest possible m', and substituting the previous result in $m' = \lfloor m/q \rfloor$, we get $O(n(k + \log_\sigma(m))/m)$ total average time.

Linear worst case time (for short patterns) can be obtained in similar way as in the case of Shift-Or. For long patterns all the bounds must be multiplied by $O(m \log(k)/w)$.

[2] Note that in a similar analysis in [26] only the lower bound $(1/\sigma)$ for γ (called a in there) was given; however their end result is correct, as $\log(1/a) = \Theta(\log(\sigma))$ again, only the constant is different (larger) as they used different matching model (edit distance), giving slightly larger upper bound for a.

6.2 *Almost Optimal Naïve Algorithm for the k-Mismatches Problem*

The naïve algorithm we used in Sec. 4 can be easily extended for Hamming distance as well. Basically, the naïve algorithm works as for the exact matching case, the window is just scanned until $k+1$ mismatches (or a complete match) are encountered. The parallelization for the pattern pieces is also similar. Basically, we maintain q counters (of $O(\log(k))$ bits each), each one corresponding to one of the possible alignments. The counters are updated until all overflow (more than k mismatches), or some counter survived $\lfloor m/q \rfloor$ symbols, in which case a verification is needed. Either case, the window is shifted by q symbols. This is easy to implement by applying the techniques developed in Sec. 6.1. Alg. 11 shows the pseudocode.

We again follow the analysis of ABNDM [26]. For simplicity, we analyze a simplified algorithm that never performs better than our real algorithm. The complexity of the simplified algorithm then upper bounds the real one. Recall that the length of a pattern piece is $m' = \lfloor m/q \rfloor$. The simplified algorithm always reads exactly (while the real algorithm may stop sooner) m' symbols for each text window, for some $m' \leq m$. If any of the q patterns have accumulated at most k mismatches after m' symbols are read, we verify them all (while the real algorithm may verify less), with a worst case cost of $O(mq)$. Whether or not verification is invoked, the window is shifted by q positions (as in the real algorithm). Hence the filtering algorithm needs $O(n/q \times m') = O(n/m \times m'^2)$ time. The verifications cost is at most $O(n/q \times mq \times \gamma^{m'}/\sqrt{m'}) = O(nm\gamma^{m'}/\sqrt{m'})$, where $\gamma < 1$, when $m' > k/(1 - e/\sigma)$. We again want to make this at most $O(n/m)$, and from here on following the steps of the previous analysis we obtain that $m' > \max(k/(1 - e/\sigma), 2\log_{1/\gamma}(m))$, and the final complexity of $O(n(k + \log_\sigma(m))^2/m)$, for $k/m < (1/2)(1 - e/\sigma)$. For long patterns the time must be multiplied by $O(q \log_2(k)/w)$, which gives $O(n \log(k)(k+\log_\sigma(m))/w)$. Again, this is (asymptotically) the same as for Alg. 10.

Finally, note that we talk about k-mismatches rather than Hamming matching, since our algorithms only find end positions for approximate matches in the text, and not return the Hamming distance for each text position (the latter problem, of course, cannot be solved in $o(n)$ time in character model even in the best case).

7 Other approximate matching models

The technique presented in this article has applications also for other approximate matching problems. In the following subsections, we show how to apply it for the problem of matching with swaps [3], circular pattern matching [22],

Alg. 11 Almost-Average-Optimal-Naïve-Hamming(T, n, P, m, q, k).

```
$\ell \leftarrow \lceil \log_2(k+1) \rceil + 1$
$iv \leftarrow 0$; $ivv \leftarrow 0$; $om \leftarrow 0$
for $i \leftarrow 0$ to $q-1$ do
    $iv \leftarrow iv \mid (1 << (i\ell))$
    $ivv \leftarrow (ivv << \ell) \mid (1 << (\ell-1)) - (e+1)$
$om \leftarrow iv << (\ell-1)$
for $j \leftarrow 0$ to $\lfloor m/q \rfloor - 1$ do
    for $i \leftarrow 0$ to $\sigma - 1$ do $B[j][i] \leftarrow iv$
    for $i \leftarrow 0$ to $q-1$ do
        $c \leftarrow P[i + j \times q]$
        $B[j][c] \leftarrow B[j][c]$ ^ $(1 << (i\ell))$
$i \leftarrow 0$
while $i < n - m + 1$ do
    $D \leftarrow ivv$; $o \leftarrow 0$; $j \leftarrow 0$
    while $j < \lfloor m/q \rfloor$ AND $o \neq om$ do
        $D \leftarrow (D + B[j][T[i + jq]]) \mid o$
        $o \leftarrow D$ & $om$
        $D \leftarrow D$ & $\sim om$
        $j \leftarrow j + 1$
    if $o \neq om$ then VerifyAAONH$(T, i, n, P, m, q, o)$
    $i \leftarrow i + q$
```

and (δ, γ)-matching.

7.1 Pattern matching with swaps

The problem of matching with local swaps (also called transpositions) is to report all text substrings $T[i \ldots i+m-1]$ such that every text symbol $T[i+j]$ matches either $P[j]$, or $P[j-1]$, or $P[j+1]$. In other words, any pair of adjacent symbols of P is allowed to be swapped, but no symbol can participate in more than one swap. A related problem is to limit the number of swaps to up to k. There are bit-parallel algorithms for this latter problem [32,25], that work in $O(nk\lceil m/w \rceil)$ time, or in $O(n\lceil m/w \rceil)$ time [25], if other edit-operations (mismatches and indels) are allowed as well.

The best result currently, when mismatches and indels are not allowed, is $O(n \log m \log \sigma)$ [3], and recently also a bit-parallel algorithm was shown [28], with $O(n \log m \lceil m/w \rceil)$ worst case time complexity. The best bit-parallel algorithms [16] solve the problem in $O(n\lceil m/w \rceil)$ and $O(n \log_\sigma(m)/m)$ worst and average case (for $m = O(w)$) times, respectively.

The average-optimal Shift-Or algorithm can be adapted for this problem as well. The only essential change is in the preprocessing; the mask $B[c]$ has hth bit set to 0, iff $P[iq + j] = c$, or $P[iq + j - 1] = c$, or $P[iq + j + 1] = c$ (cf.

Alg. 2, line 5). In this way, more verifications are needed compared to the exact matching problem, but it is easy to notice that on average the match probability for a pair of symbols, at any sampled position of T, is upper-bounded by $3/\sigma$, i.e., grows only by a constant, hence the $O(n \log_\sigma(m)/m)$ average time complexity for the simpler problem remains (for $m = O(w)$), as each verification takes $O(m)$ time in the worst case, and only $O(1)$ time on average. (Note that we cannot use the probability estimation of $3/\sigma$ for the case of $\sigma < 4$. The precise probability formula, $p = 1 - (1 - 1/\sigma)^3$, should then be used, which makes showing the average time complexity equally trivial.) For longer patterns, the trick described in Sec. 3.2 can be used again, leading to $O(n \log_{\sigma/h}(m)/m)$ average time.

Another view of the above is to first build the swap automaton, used to achieve the $O(n\lceil m/w \rceil)$ worst case time [16] (see Fig. 4), and then "crop" it (Fig. 5) to create a filter. The resulting automaton then corresponds to a pattern P' such that $P'[i] \subseteq \Sigma$. This can then be processed as in Alg. 2, which the above description does, by processing the class $P'[i]$, instead of the symbol $P[i]$.

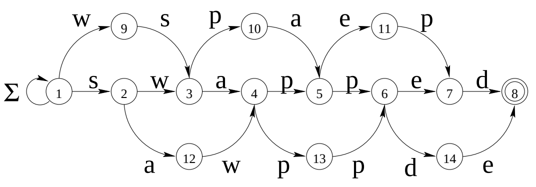

Fig. 4. Non-deterministic finite swap automaton for $P =$ `swapped`.

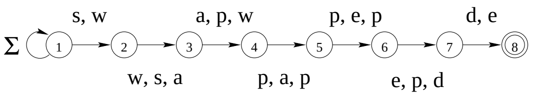

Fig. 5. "Cropped" (filter) automaton for $P =$ `swapped`.

7.2 Pattern matching with all circular shifts

In the problem of matching with all circular shifts, we are interested in reporting matches between a substring of the text and any rotation $P[i \ldots m]P[0 \ldots i-1]$ of the pattern [27]. The best known solution requires $O(n \log \sigma)$ time.

Again, we care for the average case rather than the worst case. To this end, we can partition each of the m rotations into q evenly spaced subsequences, and search for all the mq strings with a multiple matching algorithm, namely AC again. Any match (of length $\lfloor m/q \rfloor$) is verified naïvely in $O(m)$ time per matching piece. The analysis is then the same as before, but using $r = mq$, which gives $q = O(m/\log_\sigma(m))$, and $O(n \log_\sigma(m)/m)$ average time.

The problem with that analysis is that now the patterns (i.e., rotations of P)

are not independent. Still, we can make use of the analysis of several problems with transposition invariance [18]. In transposition invariance, P matches (exactly) the text substring $T[j \dots j + m - 1]$, if there exists a $t \in \{-\sigma \dots \sigma\}$ such that $P[i] + t = T[j + i]$ for every i. The problem was solved by generating all the $O(\sigma)$ possible transpositions and resorting to multiple matching. Our case is similar: the generated patterns are not random, but depend on the original pattern. However, they showed [18, Sec. 5.3] that the average case complexity analysis that assumes uniform distribution of the pattern symbols is still valid, even if the generated patterns are not independent. This analysis generalizes straightforwardly to our case. We do not repeat the proof here, but just give the necessary modifications: Instead of $O(\sigma)$ patterns, we use mq patterns, i.e. $O(m^2)$ in the worst case; the "ℓ-gram" length they used is now m/q, i.e. $O(\log_\sigma(m))$ (actually the same value that they used); the naïve verification cost is not $O(\sigma mn)$, but $O(nm^2)$ now. All the steps of the proof can then be repeated (even when our problem is not the same), showing that our $O(n \log_\sigma(m)/m)$ average case bound is valid. This is also the lower bound for the problem, as otherwise we could solve the exact string matching problem faster, by using some faster algorithm for circular shifts as a filter. Hence our algorithm is optimal on average.

Finally, we note that the average-optimal Shift-Or algorithm could be adapted as well, by generating a pattern $P' = P[0 \dots m - 1]P[0 \dots m - 2]$. The substrings of P' include all the substrings of all the circular rotations of P, and hence we could use P' as a filter (but still using text windows of length m only; note that this trick does not work with all algorithms, but AOSO poses no problems). As $|P'| = O(m)$, the complexity remains the same as for the exact matching.

7.3 Efficient (δ, γ)-Matching

Yet another problem where we can apply our idea is (δ, γ)-matching [8,12], where an integer alphabet is assumed, the matching symbols in T may differ by at most δ to their respective symbols in P, and the total sum of the absolute values of those differences does not exceed γ (obviously, $\gamma < m\delta$). We modify an algorithm from [12], which runs in $O(n\lceil m(1 + \log(\gamma + 1))/w\rceil)$ time. Using our techniques gives us an $O(n\lceil m(1 + \log(\gamma + 1))/w\rceil/q)$ time filtering algorithm. Assuming uniform random distribution of characters, we can select $q = O(m/\log_{\sigma/\delta}(m))$, which gives $O(\lceil\log(\gamma)m/w\rceil n \log_{\sigma/\delta}(m)/m)$ time, which is optimal on average [18] if $\lceil\log(\gamma)m/w\rceil = O(1)$. Asymptotically the time becomes $O(n \log(\gamma) \log_{\sigma/\delta}(m)/w)$ on average. The same kind of result was achieved in [12] as well (but the complexity was not given there), by using a different technique.

The average case analysis does not profit from the γ condition, but assumes that it is at least $m\delta$. Our algorithm is filter, while the original algorithms [12] were not. Hence the original algorithm is needed to maintain the sum of the differences, to avoid verifications. In our case we can simply use only the δ condition in the filtering phase, which means that the algorithm degenerates to Shift-Or with character classes, and the $O(\log(\gamma))$ term becomes 1, and the algorithm has optimal average case time for $m = O(w)$. The verification phase must of course use the γ condition as well.

8 Experimental results

In this section we provide some experimental results. These are not meant to be exhaustive, but to show the potential of our techniques. Nevertheless, the results show that we can easily beat some of the best previous algorithms. The experiments concentrate on the average-optimal Shift-Or algorithm, but we give also some results for the average-optimal Shift-Add and Aho–Corasick algorithms. The "naïve" almost-optimal variants of Shift-Or and Shift-Add were not competitive for the reasonably short pattern lengths we used, and the exact results are not given here.

All the algorithms were implemented in C. For the experiments we used 3.0GHz Intel Core 2 Duo (E6850) with 2 GB RAM, 4 MB cache, running GNU/Linux 2.6.22.4 and `icc 10.0` compiler. For experiments in Pentium 4 and UltraSPARC IIIi architectures, refer to [17].

We performed the experiments using random ASCII ($\sigma = 96$, $n = 10$MB), and several real texts. These are: the E.coli DNA sequence (4,638,690 characters) from Canterbury Corpus[3], real protein data (5,050,292 characters) from TIGR Database (TDB)[4], and natural language text (the collected works of Charles Dickens, 10,192,446 characters), from Silesia Corpus[5]. The patterns were randomly extracted from the texts, and each test was repeated 100 times. We report the average speed in megabytes per second.

8.1 Shift-Or experiments

The algorithms included in the experiments were the following:

[3] `http://corpus.canterbury.ac.nz/descriptions/`
[4] `http://www.tigr.org/tdb`
[5] `http://sun.aei.polsl.pl/~sdeor/silesia.html`

BNDM: The baseline algorithm [33], one of the best known and most effective of the bit-parallel algorithms.
SBNDM: Simplified version of BNDM [32,35]; in practice faster, but examines more text characters.
BNDM2: The fastest algorithm of the BNDM family [23].
AOSO: Our Average-Optimal Shift-Or algorithm.
FAOSO: Fast variant of AOSO, using unroll factor of $U = 4$.
AOSOA: Adaptive version of AOSO.
Shift-Or: Plain classical Shift-Or algorithm [5].
Fast Shift-Or: Fast variant of Shift-Or, using unroll factor of $U = 4$.

We also compared against the Boyer–Moore–Horspool algorithm [24] (including optimized variants, using fast skip-loops), and Boyer–Moore–Horspool–Sunday algorithm [36], but these were not competitive, so we do not report the results here. Besides BNDM2, they present several other BNDM variants [23], but BNDM2 was clearly the best in our experiments. All algorithms were implemented by ourselves.

For AOSO and FAOSO the optimal q value was found experimentally (however, especially FAOSO is not too sensitive to the exact value of q, for long patterns slightly too small q values do not degrade the performance much). AOSOA uses adaptive method (see Sec. 2.1), implemented as follows: the initial value is $q = m$; every time a verification is invoked, q is decremented by one; after every 256th text access by the filter, q is incremented by one. This simple strategy works well in practice. AOSOA also uses a simple unrolling method (inner loop is simply repeated 4 times), but not the more advanced and efficient method used by FAOSO.

Table 1 gives the speeds in megabytes per second for all the texts. The q values reported correspond both to AOSO and FAOSO. As it can be seen, our algorithms are clearly the fastest on DNA in all the cases. Interestingly, the fast variant of the plain Shift-Or algorithm beats our average optimal Shift-Or for short patterns. FAOSO is the best alternative also for natural language, but in some cases the gap against BNDM2 is small. The results for proteins and random ASCII are worse for us; in some cases BNDM2 wins by a wide margin.

In general, the best algorithms are FAOSO and BNDM2, with only a few exceptions. The main problem with FAOSO (and AOSO) is that q must be an integer, and this forces too small values in some cases. The problem with BNDM2 is that, assuming that they unroll U times, they can shift only after reading U characters, and the maximum shift is reduced to $m - U + 1$. Our algorithms do not have such limitations.

DNA						
Shift-Or: 478, Fast Shift-Or: **1164**						
m, q	AOSO	FAOSO	AOSOA	BNDM	SBNDM	BNDM2
$4, 2$	329	395	508	334	400	567
$8, 2$	802	**1474**	851	592	707	819
$12, 4$	983	**2011**	1301	825	1001	1079
$16, 4$	1474	**2458**	1638	1022	1286	1382
$20, 4$	1695	**3276**	2011	1222	1563	1638
$24, 4$	1762	**3597**	2107	1427	1843	1923
$28, 4$	1777	**3717**	2458	1602	2116	2212

proteins						
Shift-Or: 473, Fast Shift-Or: 1151						
m, q	AOSO	FAOSO	AOSOA	BNDM	SBNDM	BNDM2
$4, 2$	882	**1605**	892	753	917	1473
$8, 4$	1553	2603	1417	1120	1294	**2992**
$12, 4$	1720	3676	2189	1416	1738	**4081**
$16, 8$	2676	3853	2676	1852	2271	**4816**
$20, 8$	2676	3853	3010	2189	2816	**5235**
$24, 8$	3211	**5873**	3705	2675	3368	5473
$28, 8$	3440	**5873**	4014	3211	3947	5734

natural language						
Shift-Or: 476, Fast Shift-Or: 1151						
m, q	AOSO	FAOSO	AOSOA	BNDM	SBNDM	BNDM2
$4, 2$	798	**1450**	817	710	816	1350
$8, 4$	1495	**2492**	1369	1020	1171	2430
$12, 4$	1735	**3600**	2160	1275	1555	3240
$16, 4$	1767	**3641**	2627	1502	1948	3600
$20, 6$	2558	**4628**	2859	1714	2337	3888
$24, 8$	3240	**5143**	3600	1921	2730	4050
$28, 8$	3240	**5282**	4050	2118	3076	4226

random ASCII						
Shift-Or: 477, Fast Shift-Or: 1152						
m, q	AOSO	FAOSO	AOSOA	BNDM	SBNDM	BNDM2
$4, 2$	901	2000	1149	1316	**2150**	1339
$8, 4$	1786	**3636**	2041	2222	3125	2967
$12, 6$	2564	**4878**	2941	2941	3636	4367
$16, 8$	3330	**5556**	3571	3300	4000	5495
$20, 10$	4000	5714	4348	3703	4348	**5814**
$24, 12$	4546	5882	4546	4167	4444	**6024**
$28, 12$	4546	5882	4762	4348	4651	**6289**

Table 1
Searching speed in megabytes per second for different algorithms.

Finally, Table 2 gives the speeds for the average-optimal Shift-Add (AOSA) for Core 2. Our character skipping technique clearly speeds-up Shift-Add as well, the exception being short patterns or large k on DNA alphabet, where our algorithm essentially degenerates to plain Shift-Add.

8.2 Aho–Corasick experiments

We implemented also AC-automaton, and its average-optimal version (AOAC). The implementation uses a full automaton without the fail-

Alg	AOSA	AOSA	AOSA	AOSA	AOSA	Shift-Add
m k	$m = 8$ $k = 1$	$m = 12$ $k = 1$	$m = 16$ $k = 1$	$m = 8$ $k = 2$	$m = 16$ $k = 2$	$m = 8 \ldots 16$ $k = 1 \ldots 2$
DNA	379	702	834	379	681	379
Proteins	816	944	1554	438	860	379
NL (ASCII)	784	875	1519	397	860	379
Rnd (ASCII)	855	1613	1724	826	1587	379

Table 2
Searching speed in megabytes per second for Average-Optimal Shift-Add.

DNA			
m, r, q	AOAC	AC	BSOM
8, 1, 2	**691**	421	394
16, 1, 4	**1341**	421	504
64, 1, 16	**3403**	421	1593
8, 16, 1	385	**417**	127
16, 16, 2	**737**	413	234
64, 16, 9	**1580**	395	725
8, 64, 1	323	**369**	61
16, 64, 2	**481**	357	153
64, 64, 9	**714**	297	433

proteins			
m, r, q	AOAC	AC	BSOM
8, 1, 4	**1338**	422	708
16, 1, 8	**2535**	419	1170
64, 1, 16	**5351**	419	3265
8, 16, 2	**764**	415	303
16, 16, 5	**1505**	412	554
64, 16, 16	**2349**	385	1517
8, 64, 2	**645**	395	216
16, 64, 4	**958**	376	396
64, 64, 16	**1021**	295	775

natural language			
m, r, q	AOAC	AC	BSOM
8, 1, 4	**1312**	419	729
16, 1, 8	**2090**	419	1098
64, 1, 20	**5064**	419	2682
8, 16, 2	**741**	415	250
16, 16, 4	**1361**	414	457
64, 16, 16	**2576**	400	1234
8, 64, 2	**525**	375	141
16, 64, 4	**894**	360	279
64, 64, 16	**1225**	315	680

random ASCII			
m, r, q	AOAC	AC	BSOM
8, 1, 4	**1520**	420	1210
16, 1, 8	**2860**	420	1980
64, 1, 32	**5710**	418	3830
8, 16, 4	**1390**	417	510
16, 16, 8	**2250**	415	1030
64, 16, 32	**3300**	398	2340
8, 64, 4	**940**	405	376
16, 64, 8	**1300**	394	680
64, 64, 20	**1590**	345	1120

Table 3
Searching speed in megabytes per second for different algorithms.

transitions. For a comparison, we used the Backward Set Oracle Matching (BSOM) algorithm [2,34] (implemented by its authors). This is a simplified version of multiple BDM algorithm, but it has been experimentally shown that BSOM is always faster than BDM [34], and one of the fastest algorithms for moderate to long patterns, the competitiveness increasing also for increasing alphabet sizes.

The experiments were run using pattern set sizes $r \in \{1, 16, 64\}$ and pattern lengths $m \in \{8, 16, 64\}$. The results are shown in Table 3 for Core 2 Duo. Our algorithm is always faster than BSOM, but loses to plain AC for short DNA patterns for $r = 16, 64$. The reason is that in these cases the optimal q is 1, i.e. AOAC degenerates to plain AC, with additional complexity. However, for larger alphabets and longer patterns our approach is better by far.

9 Conclusions

We have presented an extremely simple filtering technique which has a surprising number of applications in string matching algorithms. The resulting new algorithms often have optimal running times on average, and have simple implementations, which helps them achieve very competitive speeds in practice. The simplicity comes from a novel forward matching technique (as opposed to backward matching as in most competing algorithms) and from the fact that the pattern shifts are constant. This also leads to simple unrolling trick that boosts the search in modern hardware. This trick cannot be applied so successfully to more complex backward matching algorithms.

We started with the classic exact string matching problem, and showed how our technique can be used to modify the well-known bit-parallel algorithm, Shift-Or, to achieve the optimal running time on average, for short patterns. Interestingly, the same idea can be used for other exact string matching algorithms, as we showed on the example of modifying the brute-force algorithm. Our best result for exact matching, in asymptotic terms, is based on building the Aho–Corasick automaton for a number of subsequences of the given pattern. The algorithm achieves the optimal $O(n \log_\sigma(m)/m)$ average time, without any limitation on the pattern length. Generalizing this algorithm for multiple patterns is straightforward and again optimal average search time is achieved for this problem. We note now that another application along these lines can be modifying the Karp–Rabin algorithm to improve its average time from $O(n)$ (for a "reasonable" pattern length; see [29] and [11, Sec. 34.2] for details) to the optimal $O(n \log_\sigma(m)/m)$. The key idea of the original algorithm is to calculate a signature (hash) from all pattern symbols, and compare it to a respective signature for a text substring. Matches have to be verified (usually with a brute-force algorithm), but mismatches with equal signatures are very rare. An important property of the algorithm is that the hash function can be calculated incrementally, paying $O(1)$ time per a text character. Our technique of skipping characters in regular intervals can also be applied to the KR algorithm. The only problem is that matching a signature built over m/q text symbols against all q signatures for subsequences of the pattern cannot be done, in brute-force manner, as it would not lead to any improvement in the average time complexity. Instead, we can use a hash table with signatures as keys, and restrict the signatures to, e.g., $\min(m, \log_2(n)/2)$ bits, instead of the standard whole machine word (which has at least $\log n$ bits). In this way, the hash table gets quite cheap both in space and initialization time, and the average lookup time remains $O(1)$.

We also consider several approximate matching problems. One is matching under Hamming distance, where we present two algorithms, the first based on the Shift-Add algorithm, the other on brute-force; the former of them achieves the optimal average case complexity for short patterns. Other models are pattern matching with swaps, and pattern matching with all circular shifts, where again our technique easily allows to achieve optimal average case complexity, for short patterns in case of the former problem, and for any patterns in case of the latter one. Finally, the (δ, γ)-matching problem, motivated by music information retrieval, is discussed. The obtained average case time is again optimal for short patterns. Our technique could be used with the classic pattern partitioning technique [31] as well, to solve string matching under Levenshtein distance (allowing k insertions, deletions or substitution of characters) in $O(nk \log_\sigma(m)/m)$ average time, but this is optimal only for $\log_\sigma(m) = O(1)$.

Some of the proposed algorithms have been shown in thorough experiments, with various real-world and artificial texts, to be very competitive in practice.

Acknowledgments

We thank the anonymous reviewers for many helpful comments.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[2] C. Allauzen and M. Raffinot. Factor oracle of a set of words. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.

[3] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlap matching. In *Proceedings of the 12th ACM-SIAM Annual Symposium on Discrete Algorithms*, pages 279–288, 2001.

[4] 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.

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

[6] M. Beeler, R. W. Gosper, and R. Schroeppel. HAKMEM. MIT AI Memo 239, 1972. `http://www.inwap.com/pdp10/hbaker/hakmem/algorithms.html` (link verified Feb 4, 2008).

[7] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[8] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In *Proceedings of the 10th Australasian Workshop on Combinatorial Algorithms*, volume 3, pages 114–128. Curtin University Press, 1999.

[9] W.I. Chang and T. Marr. Approximate string matching with local similarity. In M. Crochemore and D. Gusfield, editors, *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, number 807 in Lecture Notes in Computer Science, pages 259–273, Asilomar, CA, 1994. Springer-Verlag, Berlin.

[10] B. Commentz-Walter. A string matching algorithm fast on the average. In H. A. Maurer, editor, *Proceedings of the 6th International Colloquium on Automata, Languages and Programming*, number 71 in Lecture Notes in Computer Science, pages 118–132, Graz, Austria, 1979. Springer-Verlag, Berlin.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, London, England, 1st edition, 1990.

[12] 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.

[13] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

[14] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings.* Cambridge University Press, 2007. 392 pages.

[15] B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics*, 3:29–46, 1964.

[16] K. Fredriksson. Fast algorithms for string matching with and without swaps. Unpublished manuscript, `http://www.cs.uku.fi/~fredriks/pub/papers/sm-w-swaps.pdf`, 2000.

[17] K. Fredriksson and Sz. Grabowski. Practical and optimal string matching. In *Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE'2005)*, LNCS 3772, pages 374–385. Springer–Verlag, 2005.

[18] K. Fredriksson, V. Mäkinen, and G. Navarro. Flexible music retrieval in sublinear time. *International Journal of Foundations of Computer Science (IJFCS)*, 17(6):1345–1364, 2006.

[19] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics (JEA)*, 9(1.4):1–47, 2004.

[20] K. Fredriksson, G. Navarro, and E. Ukkonen. Sequential and indexed two-dimensional combinatorial template matching allowing rotations. *Theoretical Computer Science A*, 347(1–2):239–275, 2005.

[21] Sz. Grabowski and K. Fredriksson. Bit-parallel string matching under Hamming distance in $O(n\lceil m/w \rceil)$ worst case time. *Information Processing Letters*, 105(5):182–187, 2008.

[22] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, Cambridge, 1997.

[23] J. Holub and B. Durian. Fast variants of bit parallel approach to suffix automata. Talk given in *The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation*, 2005. `http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf` (link verified Jan 17, 2008).

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

[25] H. Hyyrö. Bit-parallel approximate string matching algorithms with transposition. *Journal of Discrete Algorithms*, 3(2–4):215–229, 2005.

[26] H. Hyyrö and G. Navarro. Bit-parallel witnesses and their applications to approximate string matching. *Algorithmica*, 41(3):203–231, 2005.

[27] C. S. Iliopoulos and M. S. Rahman. Indexing circular patterns. In *Proceedings of the Workshop on Algorithms and Computation*, Dhaka, Bangladesh, February 2008.

[28] C. S. Iliopoulos and M. S. Rahman. A new model to solve the swap matching problem and efficient algorithms for short patterns. In *Proceedings of SOFSEM'2008*, volume 4910 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 2008.

[29] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

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

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

[32] G. Navarro. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience (SPE)*, 31:1265–1312, 2001.

[33] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000. `http://www.jea.acm.org/2000/NavarroString`.

[34] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

[35] H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE2003)*, LNCS 2857, pages 80–94. Springer–Verlag, 2003.

[36] D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.

[37] S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

[38] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.