Faster Bit-parallel Approximate String Matching

Heikki Hyyrö * and Gonzalo Navarro **

Abstract. We present a new bit-parallel technique for approximate string matching. We build on two previous techniques. The first one [Myers, J. of the ACM, 1999], searches for a pattern of length m in a text of length n permitting k differences in $O(mn/w)$ time, where w is the width of the computer word. The second one [Navarro and Raffinot, ACM JEA, 2000], extends a sublinear-time exact algorithm to approximate searching. The latter technique makes use of an $O(kmn/w)$ time algorithm [Wu and Manber, Comm. ACM, 1992] for its internal workings. This algorithm is slow but flexible enough to support all the required operations. In this paper we show that the faster algorithm of Myers can be adapted to support all those operations. This involves extending it to compute edit distance, to search for any pattern suffix, and to detect in advance the impossibility of a later match. The result is an algorithm that performs better than the original version of Navarro and Raffinot and that is the fastest for several combinations of m, k and alphabet sizes that are useful, for example, in natural language searching and computational biology.

1 Introduction

Approximate string matching is one of the main problems in classical string algorithms, with applications to text searching, computational biology, pattern recognition, etc. Given a text of length n, a pattern of length m, and a maximal number of differences permitted, k, we want to find all the text positions where the pattern matches the text up to k differences. The differences can be substituting, deleting or inserting a character. We call $\alpha = k/m$ the *difference ratio*, and σ the size of the alphabet Σ. All the average case figures in this paper assume random text and uniformly distributed alphabet.

In this paper we consider online searching, that is, the pattern can be preprocessed but the text cannot. The classical solution to the problem is based on filling a dynamic programming matrix and needs $O(mn)$ time [16]. Since then, many improvements have been proposed (see [11] for a complete survey). These can be divided into four types.

The first type is based on dynamic programming and has achieved $O(kn)$ worst case time [7, 9]. These algorithms are not really practical, but there exist also practical solutions that achieve, on the average, $O(kn)$ [20] and even $O(kn/\sqrt{\sigma})$ time [4].

The second type reduces the problem to an automaton search, since approximate searching can be expressed in that way. A deterministic finite automaton (DFA) is used in [20] so as to obtain $O(n)$ search time, which is worst-case optimal. The problem is that the preprocessing time and the space is $O(3^m)$, which makes the approach practical only for very small patterns. In [22] they trade time for space using a Four Russians approach, achieving $O(kn/\log s)$ time on average and $O(mn/\log s)$ in the worst case, assuming that $O(s)$ space is available for the DFAs.

The third approach filters the text to quickly discard large text areas, using a necessary condition for an approximate occurrence that is easier to check than the full condition. The areas that cannot be discarded are verified with a classical algorithm [18, 17, 5, 12, 14]. These algorithms achieve "sublinear" expected time in many cases for low difference ratios, that is, not all text characters are inspected. However, the filtration is not effective for higher ratios. The typical average complexity is $O(kn \log_\sigma m/m)$ for $\alpha = O(1/\log_\sigma m)$. The optimal average complexity is $O((k + \log_\sigma m)n/m)$ for $\alpha < 1 - O(1/\sqrt{\sigma})$ [5], which is achieved in the same paper. The algorithm, however, is not practical.

Finally, the fourth approach is bit-parallelism [1, 21], which consists in packing several values in the bits of the same computer word and managing to update all them in a single operation.

* Dept. of Computer Science, University of Tampere, Finland.
** Dept. of Computer Science, University of Chile.

The idea is to simulate another algorithm using bit-parallelism. The first bit-parallel algorithm for approximate searching [21] parallelized an automaton-based algorithm: a nondeterministic finite automaton (NFA) was simulated in $O(k\lceil m/w\rceil n)$ time, where w is the number of bits in the computer word. We call this algorithm BPA (for Bit-Parallel Automaton) in this paper. BPA was improved to $O(\lceil km/w\rceil n)$ [3] and finally to $O(\lceil m/w\rceil n)$ time [10]. The latter simulates the classical dynamic programming algorithm using bit-parallelism, and we call it BPM (for Bit-Parallel Matrix) in this paper.

Currently the most successful approaches are filtering and bit-parallelism. A promising approach combining both [14] will be called ABNDM in this paper (for Approximate BNDM). The original ABNDM was built on BPA because the latter is the most flexible for the particular operations needed. The faster BPM was not used at that time because of the difficulty in modifying it to be suitable for ABNDM.

In this paper we extend BPM in several ways so as to permit it to be used in the framework of ABNDM. The result is a competitive approximate string matching algorithm. In particular, the algorithm turns out to be the fastest for a range of m and k that includes interesting cases of natural language searching and computational biology applications.

2 Basic Concepts

2.1 Notation

We will use the following notation on strings: $|x|$ will be the length of string x; ε will be the only string of length zero; string positions will start at 1; substrings will be denoted as $x_{i...j}$, meaning taking from the i-th to the j-th character of x, both inclusive; x_i will denote the single character at position i in x. We say that x is a prefix of xy, a suffix of yx, and a substring or factor of yxz.

Bit-parallel algorithms will be described using C-like notation for the operations: bitwise "and" (&), bitwise "or" (|), bitwise "xor" ($\wedge$), bit complementation ($\sim$), and shifts to the left ($<<$) and to the right ($>>$), which are assumed to enter zero bits both ways. We also perform normal arithmetic operations ($+$, $-$, etc.) on the bit masks, which are treated as numbers in this case. Constant bit masks are expressed as sequences of bits, the first to the right, using exponentiation to denote bit repetition, for example $10^3 = 1000$ has a 1 at the 4-th position.

2.2 Problem Description

The problem of approximate string matching can be stated as follows: given a (long) text T of length n, and a (short) pattern P of length m, both being sequences of characters from an alphabet Σ of size σ, and a maximum number of differences permitted, k, find all the segments of T whose *edit distance* to P is at most k. Those segments are called "occurrences", and it is common to report only their start or end points.

The *edit distance* between two strings x and y is the minimum number of *differences* that would transform x into y or vice versa. The allowed differences are deletion, insertion and substitution of characters. The problem is non-trivial for $0 < k < m$. The *difference ratio* is defined as $\alpha = k/m$.

Formally, if $ed()$ denotes the edit distance, we may want to report start points (i.e. $\{|x|,\ T = xP'y,\ ed(P,P') \leq k\}$) or end points (i.e. $\{|xP'|,\ T = xP'y,\ ed(P,P') \leq k\}$) of occurrences.

2.3 Dynamic Programming

The oldest and still most flexible (albeit slowest) algorithm to solve the problem is based on dynamic programming [16]. We first show how to compute the edit distance between two strings

x and y. To compute $ed(x,y)$, a matrix $M_{0..|x|,0..|y|}$ is filled, where $M_{i,j} = ed(x_{1..i}, y_{1..j})$, so at the end $M_{|x|,|y|} = ed(x,y)$. This matrix is computed as follows

$$M_{i,0} \leftarrow i, \qquad M_{0,j} \leftarrow j,$$
$$M_{i,j} \leftarrow \text{if } (x_i = y_j) \text{ then } M_{i-1,j-1} \text{ else } 1 + \min(M_{i-1,j}, M_{i,j-1}, M_{i-1,j-1})$$

where the formula accounts for the three allowed operations. This matrix is usually filled columnwise left to right, and each column top to bottom. The time to compute $ed(x,y)$ is then $O(|x||y|)$.

This is easily extended to approximate searching, where $x = P$ and $y = T$, by letting an occurrence start anywhere in T. The only change is on the initial condition $M_{0,j} \leftarrow 0$. The time is still $O(|x||y|) = O(mn)$. The space can be reduced to $O(m)$ by storing only one column of the matrix at the time, namely, the one corresponding to the current text position (going left to right means examining the text sequentially).

In this case it is more appropriate to think of a column vector $C_{0...m}$, which is initialized at $C_i \leftarrow i$ and updated to C' after reading text character T_j using

$$C'_i \leftarrow \text{if } (P_i = T_j) \text{ then } C_{i-1} \text{ else } 1 + \min(C'_{i-1}, C_i, C_{i-1})$$

for all $i > 0$, and hence we report every end position j where $C_i \le k$.

Several properties of the matrix M are discussed in [19]. The most important for us is that adjacent cells in M differ at most by 1, that is, both $M_{i,j} - M_{i\pm1,j}$ and $M_{i,j} - M_{i,j\pm1}$ are in the range $\{-1, 0, +1\}$. Also, $M_{i+1,j+1} - M_{i,j}$ is in the range $\{0, 1\}$.

Fig. 1 shows examples of edit distance computation and approximate string matching.

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	0	1	2	3	4	5	6
u	2	1	0	1	2	3	4	5
r	3	2	1	0	1	2	3	4
v	4	3	2	1	1	2	3	4
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	**2**

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Fig. 1. The dynamic programming algorithm. On the left, to compute the edit distance between "survey" and "surgery". On the right, to search for "survey" in the text "surgery". The bold entries show the cell with the edit distance (left) and the end positions of occurrences for $k = 2$ (right).

2.4 The Cutoff Improvement

In [20] they consider the dynamic programming algorithm and observe that column values larger than k can be assumed to be $k+1$ without affecting the output of the computation. Cells of C with value not exceeding k are called *active*. In the algorithm, the index ℓ of the last active cell (i.e., largest i such that $C_i \le k$) is maintained. All the values $C_{\ell+1...m}$ are assumed to be $k+1$, so C needs to be updated only in the range $C_{1...\ell}$. Later [4] it was shown that, on average, $\ell = O(k)$ and therefore the algorithm is $O(kn)$.

The value ℓ has to be updated throughout the computation. Initially, $\ell = k$ because $C_i = i$. It is shown that, at each new column, the last active cell can be incremented at most by one, so we check whether $C_{\ell+1} \le k$ and in such a case we increment ℓ. However, it is also possible that which was the last active cell becomes inactive now, that is, $C_\ell > k$. In this case we have to search upwards for the new last active cell. Despite that this search can take $O(m)$ time at a given column, we cannot work more than $O(n)$ overall, because there are at most n increments of ℓ in the whole process, and hence there are no more than $n + k$ decrements. Hence, the last active cell is maintained at $O(1)$ amortized cost per column.

2.5 An Automaton View

An alternative approach is to model the search with a non-deterministic automaton (NFA) [2]. Consider the NFA for $k = 2$ differences shown in Fig. 2. Every row denotes the number of differences seen (the first row zero, the second row one, etc.). Every column represents matching a pattern prefix. Horizontal arrows represent matching a character. All the others increment the number of differences (i.e., move to the next row): vertical arrows insert a character in the pattern, solid diagonal arrows substitute a character, and dashed diagonal arrows delete a character of the pattern. The initial self-loop allows an occurrence to start anywhere in the text. The automaton signals (the end of) a match whenever a rightmost state is active.

It is not hard to see that once a state in the automaton is active, all the states of the same column and higher-numbered rows are active too. Moreover, at a given text position, *if we collect the smallest active rows at each column, we obtain the vector C of the dynamic programming* (in this case $[0,1,2,3,3,3,2]$, compare to Fig. 1).

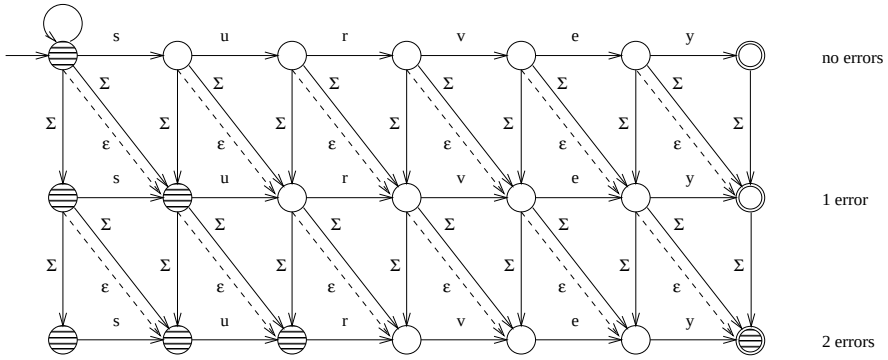

Fig. 2. An NFA for approximate string matching of the pattern "`survey`" with two differences. The shaded states are those active after reading the text "`surgery`".

Note that the NFA can be used to compute edit distance by simply removing the self-loop, although it cannot distinguish among different values larger than k.

2.6 A Bit-Parallel Automaton Simulation (BPA)

The idea of BPA [21] is to simulate the NFA of Fig. 2 using bit-parallelism, so that each row i of the automaton fits in a computer word R_i (each state is represented by a bit). For each new text character, all the transitions of the automaton are simulated using bit operations among the $k+1$ computer words.

The update formula to obtain the new R'_i values at text position j from the current R_i values is as follows:

$$R'_0 \leftarrow ((R_0 << 1) \mid 0^{m-1}1) \ \& \ B[T_j]$$
$$R'_{i+1} \leftarrow ((R_{i+1} << 1) \ \& \ B[T_j]) \ \mid \ R_i \ \mid \ (R_i << 1) \ \mid \ (R'_i << 1)$$

where $B[c]$ is a precomputed table of σ entries such that the r-th bit of $B[c]$ is set whenever $P_r = c$. We start the search with $R_i = 0^{m-i}1^i$. In the formula for R'_{i+1} are expressed, in that order, horizontal, vertical, diagonal and dashed diagonal arrows.

If $m > w$ we need $\lceil m/w \rceil$ computer words to simulate every R_i mask, and have to update them one by one. The cost of this simulation is thus $O(k\lceil m/w \rceil n)$. The algorithm is flexible, for example in order to remove the initial self-loop one has to change the update formula for R_0 to $R'_0 \leftarrow (R_0 << 1) \ \& \ B[T_j]$.

2.7 Myers' Bit-Parallel Matrix Simulation (BPM)

A better way to parallelize the computation [10] is to represent the differences between consecutive rows or columns of the dynamic programming matrix instead of the NFA states. Let us call

$$\begin{aligned} \Delta h_{i,j} &= M_{i,j} - M_{i,j-1} \in \{-1, 0, +1\} \\ \Delta v_{i,j} &= M_{i,j} - M_{i-1,j} \in \{-1, 0, +1\} \\ \Delta d_{i,j} &= M_{i,j} - M_{i-1,j-1} \in \{0, 1\} \end{aligned}$$

the horizontal, vertical, and diagonal differences among consecutive cells. Their range of values come from the properties of the dynamic programming matrix [19].

We present a version [8] that differs slightly from that of [10]: Although both perform the same number of operations per text character, the one we present is easier to understand and more convenient for our purposes.

Let us introduce the following boolean variables. The first four refer to horizontal/vertical positive/negative differences and the last to the diagonal difference being zero:

$$\begin{aligned} VP_{i,j} &\equiv \Delta v_{i,j} = +1 \qquad VN_{i,j} \equiv \Delta v_{i,j} = -1 \\ HP_{i,j} &\equiv \Delta h_{i,j} = +1 \qquad HN_{i,j} \equiv \Delta h_{i,j} = -1 \\ & D0_{i,j} \equiv \Delta d_{i,j} = 0 \end{aligned}$$

Note that $\Delta v_{i,j} = VP_{i,j} - VN_{i,j}$, $\Delta h_{i,j} = HP_{i,j} - HN_{i,j}$, and $\Delta d_{i,j} = 1 - D0_{i,j}$. It is clear that these values completely define $M_{i,j} = \sum_{r=1\ldots i} \Delta v_{r,j}$.

The boolean matrices HN, VN, HP, VP, and $D0$ can be seen as vectors indexed by i, which change their value for each new text position j, as we traverse the text. These vectors are kept in bit masks with the same name. Hence, for example, the i-th bit of the bit mask HN will correspond to the value $HN_{i,j}$. The index $j-1$ refers to the previous value of the bit mask (before processing T_j), whereas j refers to the new value, after processing T_j. By noticing some dependencies among the five variables [8, 15], one can arrive to identities that permit computing their new values (at j) from their old values (at $j-1$) fast.

Fig. 3 gives the pseudo-code. The value $diff$ stores $C_m = M_{m,j}$ explicitly and is updated using $HP_{m,j}$ and $HN_{m,j}$.

```
BPM (P_{1...m}, T_{1...n}, k)
1.    Preprocessing
2.        For c ∈ Σ Do B[c] ← 0^m
3.        For i ∈ 1...m Do B[P_i] ← B[P_i] | 0^{m-i}10^{i-1}
4.        VP ← 1^m, VN ← 0^m
5.        diff ← m
6.    Searching
7.        For j ∈ 1...n Do
8.            X ← B[T_j] | VN
9.            D0 ← ((VP + (X & VP)) ∧ VP) | X
10.           HN ← VP & D0
11.           HP ← VN | ~(VP | D0)
12.           X ← HP << 1
13.           VN ← X & D0
14.           VP ← (HN << 1) | ~(X | D0)
15.           If HP & 10^{m-1} ≠ 0^m Then diff ← diff + 1
16.           If HN & 10^{m-1} ≠ 0^m Then diff ← diff - 1
17.           If diff ≤ k Then report an occurrence at j
```

Fig. 3. BPM bit-parallel simulation of the dynamic programming matrix.

This algorithm uses the bits of the computer word better than previous bit-parallel algorithms, with a worst case of $O(\lceil m/w \rceil n)$ time. However, the algorithm is more difficult to adapt to other related problems, and this has prevented it from being used as an internal tool of other algorithms.

2.8 The ABNDM Algorithm

Given a pattern P, a *suffix automaton* is an automaton that recognizes every suffix of P. This is used in [6] to design a simple exact pattern matching algorithm called BDM, which is optimal on average ($O(n \log_\sigma m/m)$ time). To search for a pattern P in a text T, the suffix automaton of $P^r = P_m P_{m-1} \ldots P_1$ (i.e the pattern read backwards) is built. A window of length m is slid along the text, from left to right. The algorithm scans the window backwards, using the suffix automaton to recognize a factor of P. During this scan, if a final state is reached that does not correspond to the entire pattern P, the window position is recorded in a variable *last*. This corresponds to finding a *prefix* of the pattern starting at position *last* inside the window and ending at the end of the window, because the suffixes of P^r are the reverse prefixes of P. This backward search ends in two possible forms:

1. We fail to recognize a factor, that is, we reach a letter a that does not correspond to a transition in the suffix automaton (Fig. 4). In this case we shift the window to the right so as to align its starting position to the position *last*.

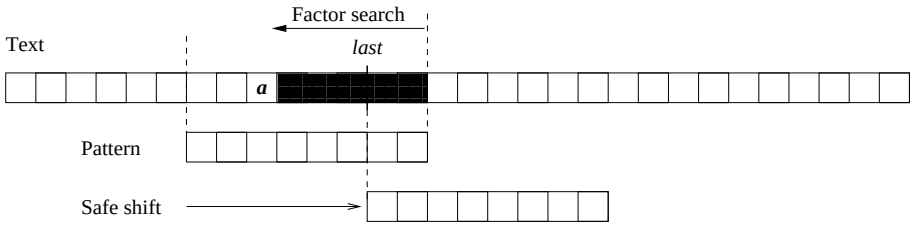

Fig. 4. BDM search scheme.

2. We reach the beginning of the window, and hence recognize P and report the occurrence. Then, we shift the window exactly as in case 1 (to the previous *last* value).

In BNDM [14] this scheme is combined with bit-parallelism so as to replace the construction of the deterministic suffix automaton by the bit-parallel simulation of a nondeterministic one. The scheme turns out to be flexible and powerful, and permits other types of search, in particular approximate search. The resulting algorithm is ABNDM.

We modify the NFA of Fig. 2 so that it recognizes not only the whole pattern but also any suffix thereof, allowing up to k differences. Fig. 5 illustrates the modified NFA. Note that we have removed the initial self-loop, so it does not search for the pattern but recognizes strings at edit distance k or less from the pattern. Moreover, we have built it on the reverse pattern. We have also added an initial state "I", with ϵ-transitions leaving it. These allow the automaton to recognize, with up to k differences, any suffix of the pattern.

In the case of approximate searching, the length of a pattern occurrence ranges from $m-k$ to $m+k$. To avoid missing any occurrence, we move a window of length $m-k$ on the text, and scan backwards the window using the NFA described above.

Each time we move the window to a new position we start the automaton with all its states active, which represents setting the initial state to active and letting the ϵ-transitions flush this activation to all the automaton (the states in the lower-left triangle are also activated to allow initial insertions). Then we start reading the window characters backward.

We recognize a prefix and update *last* whenever the final NFA state is activated. We stop the backward scan when the NFA is out of active states.

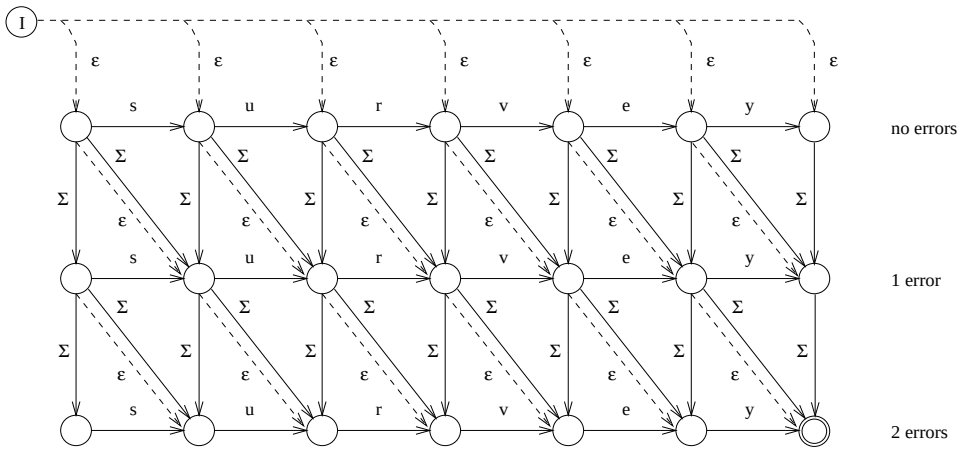

Fig. 5. An NFA to recognize suffixes of the pattern "survey" reversed.

If the automaton recognizes a pattern prefix at the initial window position, then it is possible (but not necessary) that the window starts an occurrence. The reason is that strings of different length match the pattern with k differences, and all we know is that we have matched a prefix of the pattern of length $m - k$.

Therefore, in this case we need to *verify* whether there is a pattern occurrence starting exactly at the beginning of the window. For this sake, we run the traditional automaton that computes edit distance (i.e., that of Fig. 2 without initial self-loop) from the initial window position in the text. After reading at most $m + k$ characters we have either found a match starting at the window position (that is, the final state becomes active) or determined that no match starts at the window beginning (that is, the automaton runs out of active states).

So we need two different automata in this algorithm. A first one makes the *backward scanning*, recognizing suffixes of P^r. A second one makes the *forward scanning*, recognizing P.

The automata can be simulated in a number of ways. In [14] they choose BPA [21] because it is easy to adapt to the new scenario. To recognize all the suffixes we just need to initialize $R_i \leftarrow 1^m$. To make it compute edit distance, we remove the self-loop as explained in Sec. 2.6. The final state is active when $R_k \,\&\, 10^{m-1} \neq 0^m$. The NFA is out of active states whenever $R_k = 0^m$. Other approaches were discarded: an alternative NFA simulation [3] is not practical to compute edit distance, and BPM [10] cannot easily tell when the corresponding automaton is out of active states, or which is the same, when all the cells of the current dynamic programming column are larger than k.

Fig. 6 shows the algorithm.

The algorithm is shown to be good for moderate m, low k and small σ, which is an interesting case, for example, in DNA searching. However, the use of BPA for the NFA simulation limits its usefulness to very small k values. Our purpose in this paper is to show that BPM can be extended for this task, so as to obtain a faster version of ABNDM that works with larger k.

Average Case Analysis of ABNDM. We show that ABNDM inspects on average $O(kn \log_\sigma(m)/m)$ text positions. This is better than what was previously obtained [11]. Using previous results [11], we have that the total number of strings that match a suffix of a pattern of length m with k errors is at least $\binom{m}{k}(\sigma-1)^k$ (assuming only replacements and matching only the whole pattern) and at most $m\binom{m}{k}^2\sigma^k$ (counting every suffix as if it had length m and assuming all the strings different). If we inserted those strings in a trie, the resulting height would be logarithmic (base σ) in the number of strings inserted. This means a height of $\Theta(k + m\log_\sigma m - k\log_\sigma k - (m-k)\log_\sigma(m-k))$, which can be factorized as $\Theta\left(k + m\log_\sigma \frac{m}{m-k} + k\log_\sigma \frac{m-k}{m}\right)$. We have $m\log_\sigma \frac{m}{m-k} = m\log_\sigma\left(1 + \frac{k}{m-k}\right) \leq \frac{m}{m-k}k \leq 2k$. The latter is because we are interested in the case $k < m/2$, as otherwise the algorithm cannot be sublinear time: the window is of length

```
ABNDM (P_{1...m}, T_{1...n}, k)
1.    Preprocessing
2.        Build forward and backward NFA simulations (fNFA and bNFA)
3.    Searching
4.        j ← 0
5.        While pos ≤ n − (m − k) Do
6.            j ← m − k, last ← m − k
7.            Initialize bNFA
8.            While j ≠ 0 AND bNFA has active states Do
9.                Feed bNFA with T_{pos+j}
10.               j ← j − 1
11.               If bNFA's final state is active Then /* prefix recognized */
12.                   If j > 0 Then last ← j
13.                   Else check with fNFA a possible occurrence starting at pos + 1
14.           pos ← pos + last
```

Fig. 6. The generic **ABNDM** algorithm.

$m-k$ and we read at least $k+1$ characters before the NFA can run out of active states. Hence we have that the height is $\Theta(k + k\log_\sigma(m/k))$. This is $\Theta(k\log_\sigma m)$, for example consider the case $k = m^\lambda$.

Traversing the window backwards until the NFA runs out of active states is equivalent to entering the above trie with the reverse window. On average, we reach the end of the trie in $\Theta(k\log_\sigma m)$ steps. Then we shift the window forward in $m - \Theta(k\log_\sigma m)$ positions. Overall, we inspect $O(kn\log_\sigma(m)/m)$ text positions, for $\alpha < 1/2$. If we use BPA, the complexity is $O(k^2 n\log_\sigma(m)/w)$. If we manage to use BPM, this goes down to $O(kn\log_\sigma(m)/w)$.

3 Forward Scanning with the BPM Simulation

We first focus on how to adapt the BPM algorithm to perform the forward scanning required by the ABNDM algorithm. Two modifications are necessary. The first is to make the algorithm compute edit distance instead of performing text searching. The second is making it able to determine when it is not possible to obtain edit distance $\leq k$ by reading more characters.

3.1 Computing Edit Distance

We recall that BPM implements the dynamic programming algorithm of Sec. 2.3 in such a way that differential values, rather than absolute ones, are stored. Therefore, we must consider which is the change required in the dynamic programming matrix in order to compute edit distance. As explained in Sec. 2.3, the only change is that $M_{0,j} = j$. In differential terms (Sec. 2.7), this means $\Delta h_{0,j} = 1$ instead of zero, or which is the same, that the lowest bit of HP should always be 1.

However, since this bit is always zero in the original BPM algorithm, it is not represented. The only place where the assumption $\Delta h_{0,j} = 0$ makes a difference is on line 12 of the algorithm (Fig. 3). On this line, HP is shifted to the left, and the assumed bit zero enters automatically from the right. Hence we change line 12 of the algorithm to $X \leftarrow (HP << 1) \mid 0^{m-1}1$.

Since we will use this technique several times from now on, we give in Fig. 7 the code for a single step of edit distance computation.

3.2 Preempting the Computation

Albeit in the forward scan we could always run the automaton through $m+k$ text characters, stopping only if $diff \leq k$ to signal a match, it is also possible to determine that $diff$ will always be larger than k in the characters to come. This happens when all the cells of the vector C_i are

```
BPMStep (Bc)
1.      X ← Bc | VN
2.      D0 ← ((VP + (X & VP)) ∧ VP) | X
3.      HN ← VP & D0
4.      HP ← VN | ∼(VP | D0)
5.      X ← (HP << 1) | 0^{m-1}1
6.      VN ← X & D0
7.      VP ← (HN << 1) | ∼(X | D0)
```

Fig. 7. Single step of the adaptation of BPM to compute edit distance. It receives the bit mask B of the current text character and shares all the other variables with the calling process.

larger than k, because there is no way in the recurrence to introduce a value smaller than the current ones. In the automaton view, this is the same as the NFA running out of active states (since an active state at column i and row r would mean $C_i = r \leq k$).

This is more difficult in the dynamic programming matrix simulation of BPM. The only column value that is explicitly stored is $diff = C_m$. The others are implicitly represented as $C_i = \sum_{r=1\ldots i} VP_r - VN_r$. It is not easy to check whether $\forall i,\ C_i > k$ using this incremental representation.

Our solution is inspired in the cutoff algorithm of Sec. 2.4. This algorithm permits knowing all the time the largest ℓ such that $C_\ell \leq k$, at constant amortized time per text position. Although designed for text searching, the technique can be applied without any change to the edit distance computation algorithm. Clearly $\exists i,\ C_i \leq k \quad \Leftrightarrow \quad \ell \geq 0$.

So we have to figure out how to compute ℓ using BPM. Initially, since $C_i = i$, we set $\ell \leftarrow k$. Later, we have to update ℓ for each new text character read. Recall that, given that neighboring cells in M differ by at most one, and that by definition $M_{\ell+1,j-1} > k$, we have that $M_{\ell,j-1} = k$.

Since ℓ can increase at most by one at the new text position, we start by effectively increasing it. This increment is correct when $M_{\ell+1,j} \leq k$ *before* doing the increment. Since $M_{\ell+1,j} - M_{\ell,j-1} = \Delta d_{\ell+1,j} \in \{0,1\}$, we have that it was correct to increase ℓ if and only if $D0_{\ell,j}$ *after* the increment. If it was not correct to increase ℓ, we later decrease it as much as necessary to obtain $M_{\ell,j} \leq k$. Since we know that now $M_{\ell,j} = k+1$, we obtain $M_{\ell-1,j} = M_{\ell,j} - VP_{\ell,j} + VN_{\ell,j}$, and so on with $\ell-2$, $\ell-3$, etc. If we reach $\ell = 0$ and still $M_{\ell,j} > k$, then all the rows are larger than k and we stop the scanning process.

The above arguments assume $\ell < m$. Note that, as soon as $\ell = m$, we have that $C_m \leq k$ and therefore the forward scan will then terminate because we have found an occurrence.

Fig. 8 shows the forward scanning algorithm. It scans from text position j and determines whether there is an occurrence starting at j. Instead of P, the routine receives the mask B already computed (see Fig. 3). Note that for efficiency ℓ is maintained in unary.

4 Backward Scanning with the BPM Simulation

The backward scan has the particularity that all the NFA states start active. This is equivalent to initializing C as $C_i = 0$ for all i. The place where this initialization is expressed in BPM is on line 4 of Fig. 3. Since $VP = 1^m$, we have $C_i = i$. We change it to $VP \leftarrow 0^m$ and obtain the desired effect. Also, like in forward scanning, $M_{0,j} = j$, so we apply the same change to line 12 that sets the 0-th bit in HP.

With these tools at hand, we could simply apply the forward scan algorithm with B built on P^r and reading the window backwards. We could use ℓ to determine when the NFA is out of active states. Every time $\ell = m$ we know that we have recognized a prefix and hence update *last*. There are a few changes, though: (i) we start with $\ell = m$ because $M_{i,0} = 0$; and (ii) we have to deal with the case $\ell = m$ when updating ℓ, because now we do not stop the backward scanning in that case but just update *last*.

```
BPMFwd (B, T_{j...n}, k)
1.      VP ← 1^m, VN ← 0^m
2.      ℓ ← 0^{m-k}10^{k-1}
3.      While j ≤ n Do
4.          BPMStep (B[T_j])
5.          ℓ ← ℓ << 1
6.          If D0 & ℓ = 0^m Then
7.              val ← k + 1
8.              While val > k Do
9.                  If ℓ = 0^{m-1}1 Then Return FALSE
10.                 If VP & ℓ ≠ 0^m Then val ← val − 1
11.                 If VN & ℓ ≠ 0^m Then val ← val + 1
12.                 ℓ ← ℓ >> 1
13.         Else If ℓ = 10^{m-1} Then Return TRUE
14.         j ← j + 1
15.     Return FALSE
```

Fig. 8. Adaptation of BPM to perform a forward scan from text position j and return whether there is an occurrence starting at j.

The latter problem is solved as follows. As soon as $\ell = m$ we stop tracking ℓ and initialize $diff \leftarrow k$ as the known value for C_m. We keep updating $diff$ using HP and HN just as in Fig. 3, until $diff > k$. At this moment we switch to updating ℓ again, moving it upwards as necessary.

The above scheme works correctly but it is terribly slow. The reason is that ℓ starts at m and it has to reach zero before we can leave the window. This requires m shifting operations $\ell \leftarrow \ell >> 1$, which is a lot considering that on average one traverses $O(k \log_\sigma m)$ characters in the window. The $O(k+n)$ complexity given in Sec. 2.4 becomes here $O(m + k \log_\sigma m)$. So, the problem is that all the column values reach a value larger than k quite soon, but we take too much time traversing all them to determine that this has happened.

We present two solutions to determine fast that all the C_i values have surpassed k.

4.1 Bit-Parallel Counters

In the original BPM algorithm the integer value $diff = C_m$ is explicitly maintained in order to determine which text positions match. The way is to use the m-th bit of HP and HN to keep track of C_m. This part of the algorithm is not bit-parallel, so in principle one cannot do the same with all the C_i values and still hope to update all of them in a single operation.

However, it is possible to store several such counters in the same computer word MC and use them to upper bound the others. Since the C_i values start at zero and the window is of length $m-k$, we need in principle $\lceil \log_2(m-k) \rceil$ bits to store any C_i value (their value after reading the last window character is not important). Hence we have space for $O(m/\log m)$ counters in MC.

To determine the minimum number Q of bits needed for each counter we must look a bit ahead in our algorithm. We will need to determine that all the counters have exceeded $k' = k + \lfloor Q/2 \rfloor$. For this sake, we initialize the counters at a value b that makes sure that their last bit will be activated when they surpass this threshold. So we need that $b + k' + 1 = 2^{Q-1}$. On the other hand, we have to ensure that the Q-th bit is always set for any counter value up to $b+m-k$ (and that Q bits are still enough to represent the counter), which means $b+m-k < 2^Q$. This can be resolved by first finding the minimal Q^* such that $k+1 \leq 2^{Q^*-1}$ and $m-k \leq 2^{Q^*}$. The solution is $Q^* = 1 + \lceil \log_2(\max(m-2k-1, k+1)) \rceil$. Then $Q = Q^*$ if $2^{Q^*-1} - 1 \geq k'$, and otherwise $Q = Q^* + 1$. Finally we set $b = 2^{Q-1} - k' - 1$.

We decide to store $t = \lceil m/Q \rceil$ counters, for C_m, C_{m-Q}, C_{m-2Q}, $\ldots$, $C_{m-(t-1)Q}$. The counter for C_{m-rQ} uses the bits of the region $m - rQ \ldots m - rQ + Q - 1$, both inclusive. This means that we need $m + Q - 1$ bits for MC[1].

The counters can be used as follows. Since every cell is at distance at most $\lfloor Q/2 \rfloor$ to some represented counter and the difference between consecutive cells is at most 1, it is enough that all the counters are $\geq k' + 1 = k + \lfloor Q/2 \rfloor + 1$, to be sure that all the cells of C exceed k.

So the idea is to traverse the window until all the counters exceed k' and then shift the window. We will examine a few more cells than if we had controlled exactly all the C values: The backward scan will behave as if we permitted $k' = k + \lfloor Q/2 \rfloor$ differences, so the number of characters inspected is $\Theta(n(k + \log m) \log_\sigma m/m)$. Note that we have only m/Q suffixes to test but this does not affect the complexity. Note also that the amount of shifting is not affected because we have C_m correctly represented.

We have to face two problems. The first one is how to update all the counters in a single operation. This is not hard because counter C_{m-rQ} can be updated from its old to its new value by considering the $(m - rQ)$-th bits of HP and HN. That is, we define a mask $sMask = (0^{Q-1}1)^t 0^{m+Q-1-tQ}$ and update MC using $MC \leftarrow MC + (HP \;\&\; sMask) - (HN \;\&\; sMask)$.

The second problem is how to determine that all the counters have exceeded k'. For this sake we have defined b and Q so that the Q-th bits of the counters get activated when they exceed k'. If we define $eMask = (10^{Q-1})^t 0^{m+Q-1-tQ}$, then we can stop the scanning whenever $MC \;\&\; eMask = eMask$.

Finally, note that our assumption that every cell in C is at distance at most $\lfloor Q/2 \rfloor$ to a represented cell may not be true for the first $\lfloor Q/2 \rfloor$ cells. However, we know that, at the j-th iteration, $C_0 = j$, so we may assume there is an implicit counter at row zero. Moreover, since this counter is always incremented, it is larger than any other counter, so it will surely surpass k' when other counters do. The initial $\lfloor Q/2 \rfloor$ cells are close enough to this implicit counter.

Fig. 9 shows the pseudocode of the algorithm. All the bit masks are of length m, except $sMask$, $eMask$ and MC, which are of length $m + Q - 1$.

In case our upper bound turns out to be too loose, we can use several interleaved sets of counters, each set in its own bit-parallel counter. For example we could use two interleaved MC counters and hence the limit would be $Q/4$. In general we could use c counters and have a limit of the form $Q/2^c$. The cost would be $O(nc(k + \log(m)/2^c) \log_\sigma m/m)$, which is optimized for $c = \log_2(\log_\sigma(m)/k)$, where the complexity is $O(nk \log_\sigma m \log\log_\sigma m/m)$ for $k = o(\log_\sigma m)$ and the normal $O(nk \log_\sigma m/m)$ otherwise.

4.2 Bit-Parallel Cutoff

The previous technique, although simple, has the problem that it changes the complexity of the search and inspects more cells than necessary. We can instead produce, using a similar approach, an algorithm with the same complexity as the basic version. This time the idea is to mix the bit-parallel counters with a bit-parallel version of the cutoff algorithm (Sec. 2.4).

Consider regions $m - rQ - Q + 1 \ldots m - rQ$ of length Q. Instead of having the counters fixed at the end of each region (as in the previous section), we let the counters "float" inside their region. The distance between consecutive counters is still Q, so they all float together and all are at the same distance δ to the end of their regions. We use $sMask$ and $eMask$ with the same meanigs as before, but they are displaced so as to be all the time aligned to the counters.

The invariant is that the counters will be as close as possible to the end of their regions, as long as all the cells past the counters exceed k. That is,

$$\delta \;=\; \min\{d \in 0 \ldots Q,\; \forall r \in \{0 \ldots t-1\},\; \gamma \in \{0 \ldots d-1\},\; C_{m-rQ-\gamma} > k\}$$

[1] If sticking to m bits is necessary we can store C_m separately in the *diff* variable, at the same complexity but more cost in practice.

```
ABNDMCounters (P_{1...m}, T_{1...n}, k)
1.    Preprocessing
2.        For c ∈ Σ Do Bf[c] ← 0^m , Bb[c] ← 0
3.        For i ∈ 1...m Do
4.            Bf[P_i] ← Bf[P_i] | 0^{m-i}10^{i-1}
5.            Bb[P_i] ← Bb[P_i] | 0^{i-1}10^{m-i}
6.        Q ← 1 + ⌈log_2(max(m - 2k - 1, k + 1))⌉
7.        If 2^{Q-1} - k - 1 ≥ ⌊Q/2⌋ Then Q ← Q + 1
8.        b ← 2^{Q-1} - k - ⌊Q/2⌋ - 1
9.        t ← ⌈m/Q⌉
10.       sMask ← (0^{Q-1}1)^t 0^{m+Q-1-tQ}
11.       eMask ← (10^{Q-1})^t 0^{m+Q-1-tQ}
12.   Searching
13.       j ← 0
14.       While pos ≤ n - (m - k) Do
15.           j ← m - k, last ← m - k
16.           VP ← 0^m, VN ← 0^m
17.           MC ← [b]^t_Q 0^{m+Q-1-tQ}
18.           While j ≠ 0 AND MC & eMask ≠ eMask Do
19.               BPMStep (Bb[T_{pos+j}])
20.               MC ← MC + (HP & sMask) - (HN & sMask)
21.               j ← j - 1
22.               If MC & 10^{m+Q-2} ≠ 0^{m+Q-1} Then /* prefix recognized */
23.                   If j > 0 Then last ← j
24.                   Else If BPMFwd (Bf, T_{pos+1...n}) Then
25.                       Report an occurrence at pos + 1
26.           pos ← pos + last
```

Fig. 9. The **ABNDM** algorithm using bit-parallel counters. The expression $[b]_Q$ denotes the number b seen as a bit mask of length Q. Note that **BPMFwd** can share its variables with the calling code because these are not needed anymore at that point.

where we assume that C yields values larger than k when accessed at negative indexes. When δ reaches Q, this means that all the cell values are larger than k and we can suspend the scanning. Prefix reporting is easy since no prefix can match unless $\delta = 0$, as otherwise $C_m = C_{m-0 \cdot Q} > k$, and if $\delta = 0$ then the last floating counter has exactly the value C_m.

The floating counters are a bit-parallel version of the cutoff technique, where each counter cares of its region. Consequently the way of moving the counters up and down resembles the cutoff technique. We first move down and use $D0$ to determine if we should have moved down. If not, we move up as necessary using VP and VN. To determine if we should have moved down, we need to know whether there is a counter that exceeds k. Using a similar mechanism as with computing Q^* in the previous section, we let $Q = 1 + \lceil \log_2(\max(m - 2k, k + 1)) \rceil$ and $b = 2^{Q-1} - k - 1$, and use $eMask$. In order to increment and decrement the counters we use $sMask$. We have to deal with the case where the counters are at the end of their region and hence cannot move down further. In this case we update them using HP and HN.

It is possible that the upmost counter goes out of bounds while shifting the counters, which in effect results in that counter being removed. But this is not a problem, because since $C_0 > k$ after the counters are shifted upwards for the first time, and since a disappearing counter will be equal to C_0 just before getting out of bounds, we have that this counter will be inactive. Moreover, it should be inactive from then on, because if all the first block in C is larger than k, it will not become $\leq k$ again.

As for the case of a single counter, we work $O(1)$ amortized time per text position. More specifically, if we read u window characters then we work $O(u + Q)$ because we have to move from $\delta = 0$ to $\delta = Q$. But $O(u + Q) = O(k \log_\sigma m)$ on average because $Q = O(\log m)$, and therefore the classical complexity is not altered.

We also tried a practical version of using cutoff, in which the counters are not shifted. Instead they are updated in a similar fashion to the algorithm of Fig. 9, and when all counters have a value $> k$, we try to shift a *copy* of them up until either a cell with value $\leq k$ is found or $Q-1$ consecutive shifts are made. In the latter case we can stop the search, since then we have covered checking the whole column C. This version has a worse complexity, $O(Qk \log m) = O(k \log^2 m)$, as at each processed character it is possible to make $O(Q)$ shifts. But in practice it turned out to be very similar to the present cutoff algorithm.

Fig. 10 shows the algorithm. The counters are not physically shifted, we use δ instead.

```
ABNDMCutoff (P_{1...m}, T_{1...n}, k)
1.    Preprocessing
2.        For c ∈ Σ Do Bf[c] ← 0^m , Bb[c] ← 0
3.        For i ∈ 1...m Do
4.            Bf[P_i] ← Bf[P_i] | 0^{m-i}10^{i-1}
5.            Bb[P_i] ← Bb[P_i] | 0^{i-1}10^{m-i}
6.        Q ← 2 + ⌈log_2(m - 2k - 2)⌉
7.        b ← 2^{Q-1} - k - 1
8.        t ← ⌈m/Q⌉
9.        sMask ← (0^{Q-1}1)^t 0^{m+Q-1-tQ}
10.       eMask ← (10^{Q-1})^t 0^{m+Q-1-tQ}
11.   Searching
12.       j ← 0
13.       While pos ≤ n - (m - k) Do
14.           j ← m - k, last ← m - k
15.           VP ← 0^m, VN ← 0^m
16.           MC ← [b]_Q^t 0^{m+Q-1-tQ}
17.           δ ← 0
18.           While j ≠ 0 AND δ < Q Do
19.               BPMStep (Bb[T_{pos+j}])
20.               If δ = 0 Then MC ← MC + ((HP & sMask) - (HN & sMask)
21.               Else
22.                   δ ← δ - 1
23.                   MC ← MC + (~(D0 << δ) & sMask)
24.               While δ < Q AND MC & eMask = eMask Do
25.                   MC ← MC - ((VP << δ) & sMask) + ((VN << δ) & sMask)
26.                   δ ← δ + 1
27.               j ← j - 1
28.               If δ = 0 AND MC & 10^{m+Q-2} ≠ 0^{m+Q-1} Then /* prefix recognized */
29.                   If j > 0 Then last ← j
30.                   Else If BPMFwd (Bf, T_{pos+1...n}) Then
31.                       Report an occurrence at pos + 1
32.           pos ← pos + last
```

Fig. 10. The **ABNDM** algorithm using bit-parallel cutoff. The same comments of Fig. 9 apply.

5 Experimental Results

We compared our BPM-based ABNDM against the original BPA-based ABNDM, as well as those other algorithms that, according to a recent survey [11], are the best for moderate pattern lengths. We tested with random patterns and text over uniformly distributed alphabets. Each individual test run consisted of searching for 100 patterns a text of size 10 Mb. We measured total elapsed times.

The computer used in the tests was a 64-bit Alphaserver ES45 with four 1 Ghz Alpha EV68 processors, 4 GB of RAM and Tru64 UNIX 5.1A operating system. All test programs were compiled with the DEC CC C-compiler and full optimization. There were no other active

significant processes running on the computer during the tests. All algorithms were set to use a 64 KB text buffer. The tested algorithms were:

ABNDM/BPA(regular): ABNDM implemented on BPA [21], using a generic implementation for any k.

ABNDM/BPA(special code): Same as before but especially coded for each value of k to avoid using an array of bit masks.

ABNDM/BPM(count): ABDNM implemented using BPM and counters (Sec. 4.1). The implementation differed slightly from Fig. 9 due to optimizations.

ABNDM/BPM(cutoff): ABDNM implemented using BPM and cutoff (Sec 4.2). The implementation differed slightly from Fig. 10 due to optimizations.

ABNDM/BPM(static): The version of ABNDM/cutoff that does not actively shift the counters.

BPM: The sequential BPM algorithm [10]. The implementation was from us and used the slightly different (but practically equivalent in terms of performance) formulation from [8].

BPP: A combined heuristic using pattern partitioning, superimposition and hierarchical verification, together with a diagonally bit-parallelized NFA [3, 13]. The implementation was from the original authors.

EXP: Partitioning the pattern into $k+1$ pieces and using hierachical verification with a diagonally bit-parallelized NFA in the checking phase [12]. The implementation was from the original authors.

Fig. 11 shows the test results for $\sigma = 4$, 13 and 52 and $m = 30$ and 55. This is only a small part of our complete tests, which included $\sigma = 4, 13, 20, 26$ and 52, and $m = 10, 15, 20, \ldots, 55$. We chose $\sigma = 4$ because it behaves like DNA, $\sigma = 13$ because it behaves like English, and $\sigma = 52$ to show that our algorithms are useful even on large alphabets.

First of all it can be seen that ABNDM/BPM(cutoff) is always faster than ABNDM/BPM(counters) by a nonnegligible margin.

It can be seen that our ABNDM/BPM versions are often faster than ABNDM/BPA(special code) when $k = 4$, and always when $k > 4$. Compared to ABNDM/BPA(regular), our version is always faster for $k > 1$. We note that writing down a different procedure for every possible k value, as done for ABNDM/BPA(special code), is hardly a real alternative in practice.

With moderate pattern length $m = 30$, our ABNDM/BPM versions are competitive for low error levels. However, BPP is better for small alphabets and EXP is better for large alphabets. In the intermediate area $\sigma = 13$, we are the best for $k = 4 \ldots 6$. This area is rather interesting when searching natural language text.

When $m = 55$, our ABNDM/BPM versions become much more competitive, being the fastest in many cases: For $k = 5 \ldots 9$ with $\sigma = 4$, and for $k = 4 \ldots 11$ both with $\sigma = 13$ and $\sigma = 52$, with the single exception of the case $\sigma = 52$ and $k = 9$, where EXP is faster (this seems to be a variance problem, however).

6 Conclusions

The most successful approaches to approximate string matching are bit-parallelism and filtering. A promising algorithm combining both is ABNDM [14]. However, ABNDM uses a slow $O(kmn/w)$ time bit-parallel algorithm (BPA [21]) for its internal working because no other alternative exists with the necessary flexibility. In this paper we have shown how to extend BPM [10] to replace BPA. Since BPM is $O(mn/w)$ time, we obtain a much faster version of ABNDM.

For this sake, BPM was extended to permit backward scanning of the window and forward verification. The extensions involved making it compute edit distance, making it able to recognize any suffix of the pattern with k differences, and, the most complicated, being able to tell in advance that a match cannot occur ahead, both for backward and forward scanning. We presented two alternatives for the backward scanning: a simple one that may read more characters

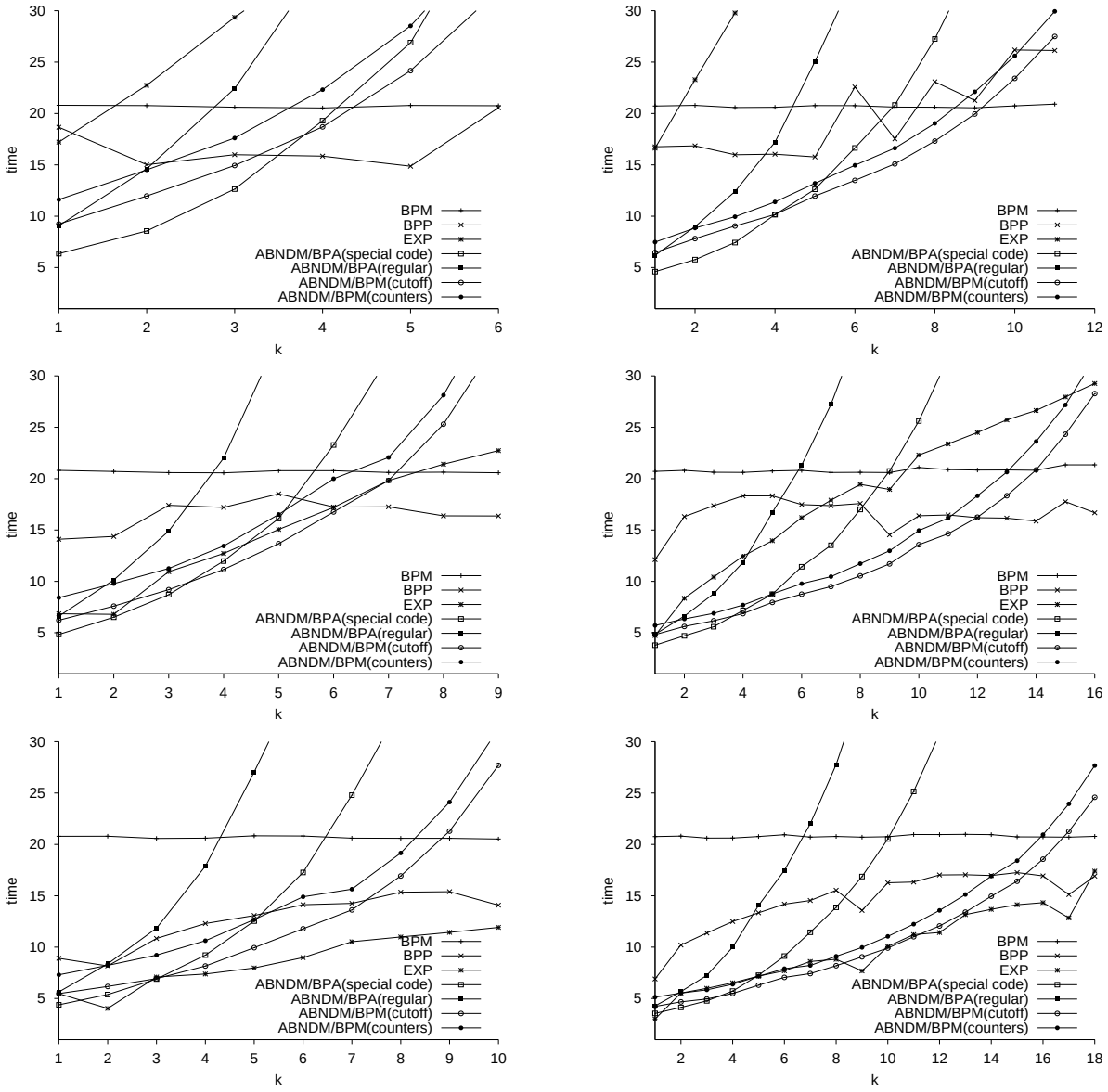

Fig. 11. Comparison between algorithms, showing total elapsed time as a function of the number of differences permitted, k. From top to bottom row we show $\sigma = 4$, 13 and 52. On the left we show $m = 30$ and on the right $m = 55$.

than necessary, and a more complicated (and more costly per processed character) that reads exactly the required characters.

The experimental results show that our new algorithm beats the old ABNDM, even when BPA is especially coded with a different procedure for every possible k value, often for $k = 4$ and always for $k > 4$, and that it beats a general BPA implementation for $k \geq 2$. Moreover it was seen that our version of ABNDM becomes the fastest algorithm for many cases with moderately long pattern and fairly low error level, provided the counters fit in a single computer word. This includes several interesting cases in searching DNA, natural language text, proteic sequences, etc.

References

1. R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, 1992.
2. R. Baeza-Yates. A unified view of string matching algorithms. In *Proc. Theory and Practice of Informatics (SOFSEM'96)*, LNCS 1175, pages 1–15, 1996.

3. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
4. W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. 3rd Combinatorial Pattern Matching (CPM'92)*, LNCS 644, pages 172–181, 1992.
5. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 259–273, 1994.
6. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, Oxford, UK, 1994.
7. Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM Journal on Computing*, 19(6):989–999, 1990.
8. H. Hyyrö. Explaining and extending the bit-parallel algorithm of Myers. Technical Report A-2001-10, University of Tampere, Finland, 2001.
9. G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989.
10. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.
11. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
12. G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, 72:65–70, 1999.
13. G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate string matching. *Algorithmica*, 30(4):473–502, 2001.
14. 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.
15. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. To appear.
16. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.
17. E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. European Symposium on Algorithms (ESA'95)*, LNCS 979, pages 327–340, 1995.
18. J. Tarhio and E. Ukkonen. Approximate Boyer-Moore string matching. *SIAM Journal on Computing*, 22(2):243–260, 1993.
19. E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.
20. E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.
21. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, 1992.
22. S. Wu, U. Manber, and G. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

APPENDIX – To be read at the discretion of the reviewer

We explain in more detail (still less than in the full version) the basic algorithms that have to be understood in order to follow our paper.

6.1 Dynamic Programming

The oldest and still most flexible (albeit slowest) algorithm to solve the problem is based on dynamic programming [16]. We first show how to compute the edit distance between two strings x and y. To compute $ed(x,y)$, a matrix $M_{0..|x|,0..|y|}$ is filled, where $M_{i,j} = ed(x_{1..i}, y_{1..j})$, so at the end $M_{|x|,|y|} = ed(x,y)$. This matrix is computed as follows

$$M_{i,0} \leftarrow i, \qquad M_{0,j} \leftarrow j,$$
$$M_{i,j} \leftarrow \text{if } (x_i = y_j) \text{ then } M_{i-1,j-1} \text{ else } 1 + \min(M_{i-1,j}, M_{i,j-1}, M_{i-1,j-1})$$

where the formula accounts for the three allowed operations. This matrix is usually filled columnwise left to right, and each column top to bottom. The time to compute $ed(x,y)$ is then $O(|x||y|)$.

This is easily extended to approximate searching, where $x = P$ and $y = T$, by letting an occurrence start anywhere in T. The only change is on the initial condition $M_{0,j} \leftarrow 0$. The time is still $O(|x||y|) = O(mn)$. The space can be reduced to $O(m)$ by storing only one column of the matrix at the time, namely, the one corresponding to the current text position (going left to right means examining the text sequentially).

In this case it is more appropriate to think of a column vector $C_{0...m}$, which is initialized at $C_i \leftarrow i$ and updated to C' after reading text character T_j using

$$C'_i \leftarrow \text{if } (P_i = T_j) \text{ then } C_{i-1} \text{ else } 1 + \min(C'_{i-1}, C_i, C_{i-1})$$

for all $i > 0$, and hence we report every end position j where $C_i \leq k$.

Several properties of the matrix M are discussed in [19]. The most important for us is that adjacent cells in M differ at most by 1, that is, both $M_{i,j} - M_{i\pm1,j}$ and $M_{i,j} - M_{i,j\pm1}$ are in the range $\{-1, 0, +1\}$. Also, $M_{i+1,j+1} - M_{i,j}$ is in the range $\{0, 1\}$.

6.2 The Cutoff Improvement

In [20] they consider the dynamic programming algorithm and observe that column values larger than k can be assumed to be $k+1$ without affecting the output of the computation. Cells of C with value not exceeding k are called *active*. In the algorithm, the index ℓ of the last active cell (i.e., largest i such that $C_i \leq k$) is maintained. All the values $C_{\ell+1...m}$ are assumed to be $k+1$, so C needs to be updated only in the range $C_{1...\ell}$. Later [4] it was shown that, on average, $\ell = O(k)$ and therefore the algorithm is $O(kn)$.

The value ℓ has to be updated throughout the computation. Initially, $\ell = k$ because $C_i = i$. It is shown that, at each new column, the last active cell can be incremented at most by one, so we check whether $C_{\ell+1} \leq k$ and in such a case we increment ℓ. However, it is also possible that which was the last active cell becomes inactive now, that is, $C_\ell > k$. In this case we have to search upwards for the new last active cell. Despite that this search can take $O(m)$ time at a given column, we cannot work more than $O(n)$ overall, because there are at most n increments of ℓ in the whole process, and hence there are no more than $n + k$ decrements. Hence, the last active cell is maintained at $O(1)$ amortized cost per column.

6.3 An Automaton View

An alternative approach is to model the search with a non-deterministic automaton (NFA) [2]. Consider the NFA for $k = 2$ differences shown in Fig. 12. Every row denotes the number of differences seen (the first row zero, the second row one, etc.). Every column represents matching a pattern prefix. Horizontal arrows represent matching a character. All the others increment the number of differences (i.e., move to the next row): vertical arrows insert a character in the pattern, solid diagonal arrows substitute a character, and dashed diagonal arrows delete a character of the pattern. The initial self-loop allows an occurrence to start anywhere in the text. The automaton signals (the end of) a match whenever a rightmost state is active.

It is not hard to see that once a state in the automaton is active, all the states of the same column and higher-numbered rows are active too. Moreover, at a given text position, *if we collect the smallest active rows at each column, we obtain the vector C of the dynamic programming* (in this case $[0, 1, 2, 3, 3, 3, 2]$).

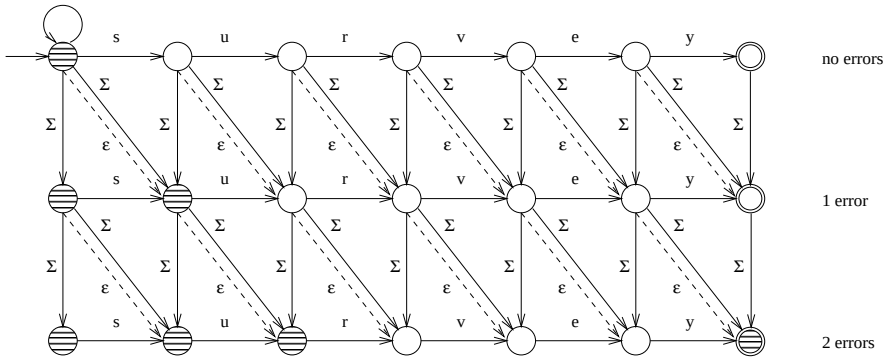

Fig. 12. An NFA for approximate string matching of the pattern "survey" with two differences. The shaded states are those active after reading the text "surgery".

Note that the NFA can be used to compute edit distance by simply removing the self-loop, although it cannot distinguish among different values larger than k.

6.4 The ABNDM Algorithm

Given a pattern P, a *suffix automaton* is an automaton that recognizes every suffix of P. This is used in [6] to design a simple exact pattern matching algorithm called BDM, which is optimal on average ($O(n \log_\sigma m/m)$ time). To search for a pattern P in a text T, the suffix automaton of $P^r = P_m P_{m-1} \ldots P_1$ (i.e the pattern read backwards) is built. A window of length m is slid along the text, from left to right. The algorithm scans the window backwards, using the suffix automaton to recognize a factor of P. During this scan, if a final state is reached that does not correspond to the entire pattern P, the window position is recorded in a variable *last*. This corresponds to finding a *prefix* of the pattern starting at position *last* inside the window and ending at the end of the window, because the suffixes of P^r are the reverse prefixes of P. This backward search ends in two possible forms:

1. We fail to recognize a factor, that is, we reach a letter a that does not correspond to a transition in the suffix automaton (Fig. 13). In this case we shift the window to the right so as to align its starting position to the position *last*.
2. We reach the beginning of the window, and hence recognize P and report the occurrence. Then, we shift the window exactly as in case 1 (to the previous *last* value).

In BNDM [14] this scheme is combined with bit-parallelism so as to replace the construction of the deterministic suffix automaton by the bit-parallel simulation of a nondeterministic one.

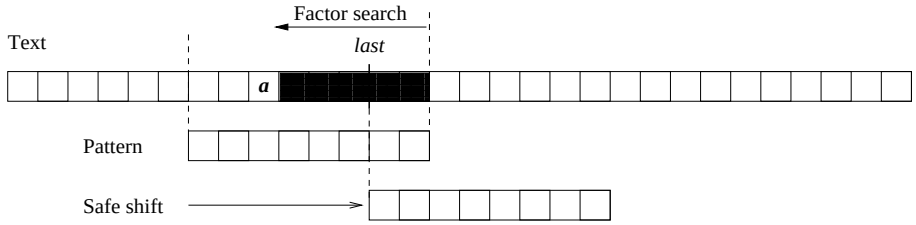

Fig. 13. BDM search scheme.

The scheme turns out to be flexible and powerful, and permits other types of search, in particular approximate search. The resulting algorithm is ABNDM.

We modify the NFA of Fig. 12 so that it recognizes not only the whole pattern but also any suffix thereof, allowing up to k differences. Fig. 14 illustrates the modified NFA. Note that we have removed the initial self-loop, so it does not search for the pattern but recognizes strings at edit distance k or less from the pattern. Moreover, we have built it on the reverse pattern. We have also added an initial state "I", with ϵ-transitions leaving it. These allow the automaton to recognize, with up to k differences, any suffix of the pattern.

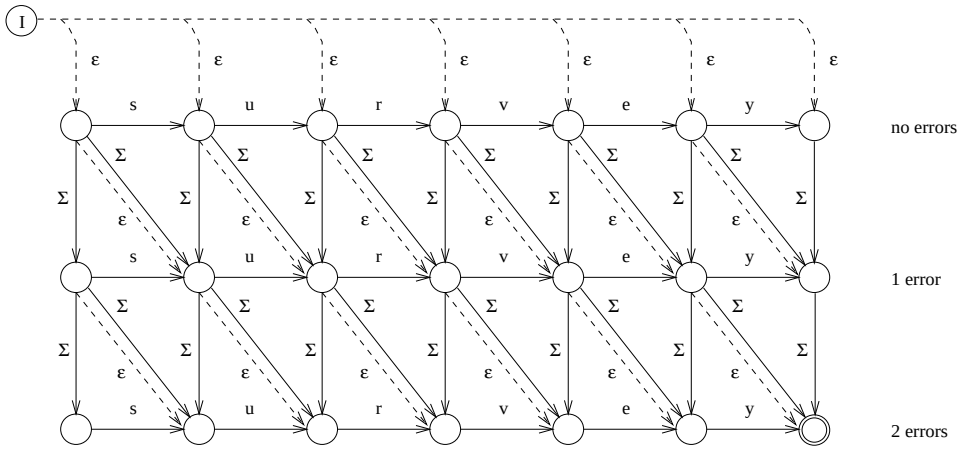

Fig. 14. An NFA to recognize suffixes of the pattern `"survey"` reversed.

In the case of approximate searching, the length of a pattern occurrence ranges from $m-k$ to $m+k$. To avoid missing any occurrence, we move a window of length $m-k$ on the text, and scan backwards the window using the NFA described above.

Each time we move the window to a new position we start the automaton with all its states active, which represents setting the initial state to active and letting the ϵ-transitions flush this activation to all the automaton (the states in the lower-left triangle are also activated to allow initial insertions). Then we start reading the window characters backward.

We recognize a prefix and update *last* whenever the final NFA state is activated. We stop the backward scan when the NFA is out of active states.

If the automaton recognizes a pattern prefix at the initial window position, then it is possible (but not necessary) that the window starts an occurrence. The reason is that strings of different length match the pattern with k differences, and all we know is that we have matched a prefix of the pattern of length $m-k$.

Therefore, in this case we need to *verify* whether there is a pattern occurrence starting exactly at the beginning of the window. For this sake, we run the traditional automaton that computes edit distance (i.e., that of Fig. 12 without initial self-loop) from the initial window position in the text. After reading at most $m+k$ characters we have either found a match starting at the

window position (that is, the final state becomes active) or determined that no match starts at the window beginning (that is, the automaton runs out of active states).

So we need two different automata in this algorithm. A first one makes the *backward scanning*, recognizing suffixes of P^r. A second one makes the *forward scanning*, recognizing P.

The automata can be simulated in a number of ways. In [14] they choose BPA [21] because it is easy to adapt to the new scenario. Other approaches were discarded: an alternative NFA simulation [3] is not practical to compute edit distance, and BPM [10] cannot easily tell when the corresponding automaton is out of active states, or which is the same, when all the cells of the current dynamic programming column are larger than k.

Fig. 15 shows the algorithm.

```
ABNDM (P_{1...m}, T_{1...n}, k)
1.    Preprocessing
2.        Build forward and backward NFA simulations (fNFA and bNFA)
3.    Searching
4.        j ← 0
5.        While pos ≤ n − (m − k) Do
6.            j ← m − k, last ← m − k
7.            Initialize bNFA
8.            While j ≠ 0 AND bNFA has active states Do
9.                Feed bNFA with T_{pos+j}
10.               j ← j − 1
11.               If bNFA's final state is active Then /* prefix recognized */
12.                   If j > 0 Then last ← j
13.                   Else check with fNFA a possible occurrence starting at pos + 1
14.           pos ← pos + last
```

Fig. 15. The generic **ABNDM** algorithm.

The algorithm is shown to be good for moderate m, low k and small σ, which is an interesting case, for example, in DNA searching. However, the use of BPA for the NFA simulation limits its usefulness to very small k values. Our purpose in this paper is to show that BPM can be extended for this task, so as to obtain a faster version of ABNDM that works with larger k.