Bit-parallel Witnesses and their Applications to Approximate String Matching

Heikki Hyyrö * Gonzalo Navarro †

Abstract

We present a new bit-parallel technique for approximate string matching. We build on two previous techniques. The first one, BPM [Myers, J. of the ACM, 1999], searches for a pattern of length m in a text of length n permitting k differences in $O(\lceil m/w \rceil n)$ time, where w is the width of the computer word. The second one, ABNDM [Navarro and Raffinot, ACM JEA, 2000], extends a sublinear-time exact algorithm to approximate searching. ABNDM relies on another algorithm, BPA [Wu and Manber, Comm. ACM, 1992], which makes use of an $O(k\lceil m/w \rceil n)$ time algorithm for its internal workings. BPA is slow but flexible enough to support all operations required by ABNDM. We improve previous ABNDM analyses, showing that it is average-optimal in number of inspected characters, although the overall complexity is higher because of the $O(k\lceil m/w \rceil)$ work done per inspected character. We then show that the faster BPM can be adapted to support all the operations required by ABNDM. 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 solution to those challenges is based on the concept of a *witness*, which permits sampling some dynamic programming matrix values so as to bound, deduce, or compute others fast. The resulting algorithm is average-optimal for $m \leq w$, assuming the alphabet size is constant. In practice, it performs better than the original ABNDM and is the fastest algorithm for several combinations of m, k and alphabet sizes that are useful, for example, in natural language searching and computational biology. To show that the concept of witnesses can be used in further scenarios, we also improve a recent bit-parallel algorithm based on Myers [Fredriksson, SPIRE 2003]. The use of witnesses greatly improves the running time of this algorithm too.

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

*Dept. of Computer Sciences, University of Tampere, Finland. Supported by the Academy of Finland and Tampere Graduate School in Information Science and Engineering.

†Dept. of Computer Science, University of Chile. Partially supported by Fondecyt Project 1-020831.

and needs $O(mn)$ time [18]. Since then, many improvements have been proposed (see [13] 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 [8, 11]. These algorithms are not really practical, but there exist also practical solutions that achieve, on the average, $O(kn)$ [22] 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 [22] so as to obtain $O(n)$ search time, which is worst-case optimal. The problem is that the preprocessing time and the space is $O(\min(3^m, (m\sigma)^k)$ in the worst case, which makes the approach practical only for very small patterns. In [25] 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 [20, 19, 5, 14, 16]. 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 the fastest in practice.

Finally, the fourth approach is bit-parallelism [1, 24], which consists in packing several values in the bits of the same computer word and managing to update them all in a single operation. The idea is to simulate another algorithm using bit-parallelism. The first bit-parallel algorithm for approximate searching [24] 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 [12]. 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 in practice are filtering and bit-parallelism. A promising approach combining both [16] will be called ABNDM in this paper (for Approximate BNDM, where BNDM stands for Backward Nondeterministic DAWG Matching). 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 yet 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. We show that, for $m \leq w$, the algorithm has average-optimal complexity $O((k + \log m)n/m)$ for $\alpha < 1/2 - O(1/\sqrt{\sigma})$. Note that optimality holds provided we assume σ is a constant. For longer patterns it becomes $O((k+\log m)n/w)$. In practice, 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. For our analysis, we prove that ABNDM inspects a (truly) optimal number of characters, despite not having an optimal overall complexity.

Among the extensions needed by BPM, the most challenging one is making it detect whether or not the characters read up to now can lead to a match. Under the automaton approach (BPA) this is easy because it is equivalent to the automaton having run out of active states. BPM, however, does not simulate an automaton but rather a dynamic programming matrix. In this case, the condition sought is that all matrix values in the last column exceed k. Since BPM handles differential rather than absolute matrix values, this kind of check is difficult and has prevented using BPM instead of BPA for ABNDM.

We solve the problem by introducing the *witness* concept. A witness is a matrix cell whose absolute value is known. Together with the differential values, we update one or more witness values in parallel. Those witnesses are used to deduce, bound or compute all the other matrix values.

The usefulness of the witness concept goes well beyond the application we developed it for. To demonstrate this, we show how it can be used to improve a recently proposed algorithm [7] where the main idea is to compute the dynamic programming matrix, using BPM, in row-wise rather than the usual column-wise fashion. One of the subproblems addressed in [7] is how to determine that it is not necessary to compute more rows. Again, the condition is that all current values exceed k. We show that our witness technique yields large improvements over the solution presented in [7].

The structure of the paper is as follows. Section 2 presents the background necessary to follow the paper. Section 3 analyzes the classical ABNDM algorithm, because previous analyses [13, 10] are pessimistic. Section 4 shows how BPM algorithm can be adapted to meet the requirements of ABNDM verification. Section 5 gives the changes to BPM that are necessary for ABNDM scanning. Section 6 gives experimental results on the improved ABNDM algorithm. Section 7 shows how the witness technique can be used to improve the row-wise BPM algorithm. Finally, Section 8 gives our conclusions and future work directions.

An earlier partial version of this work appeared in [10].

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 a factor of yxz.

Bit-parallel algorithms will be described using C-like notation for the operations: bitwise "and" (&), bitwise "or" (|), bitwise "xor" (^), 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 [18]. We first show how to compute the edit distance between two strings x and y. To compute $ed(x, y)$, a $(|x| + 1) \times (|y| + 1)$ dynamic programming matrix $M_{0..|x|,0..|y|}$ is filled so that eventually $M_{i,j} = ed(x_{1..i}, y_{1..j})$. The desired solution is then obtained as $M_{|x|,|y|} = ed(x, y)$. Matrix M can be filled by using the well-known dynamic programming recurrence

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

where the formula accounts for the three allowed operations. After setting the boundary conditions (first line of the recurrence), it is common to fill the remaining cells of M in a column-wise manner from left to right: The columns are processed in the order $j = 1 \ldots |y|$, and column j is filled from top to bottom in the order $i = 1 \ldots |x|$ before moving to the next column $j + 1$. Dynamic programming requires $O(|x||y|)$ time to compute $ed(x, y)$. The space can be reduced to $O(m)$ by storing only one column of M at a time, namely, the one corresponding to the current character of y (going left to right means examining y sequentially).

The preceding method is easily extended to approximate searching, where $x = P$ and $y = T$, by letting the comparison between P and T start anywhere in T. The only change is the initial condition $M_{0,j} \leftarrow 0$. The time is still $O(|x||y|) = O(mn)$.

Throughout this paper we assume that M is processed in column-wise manner. Let the vector $C_{0\ldots m}$ correspond to the values in the currently processed column of M. Then the equality $C_i = M_{i,j}$ holds whenever we have just processed the text character T_j. Initially $C_i \leftarrow M_{i,0} = i$. When we move on to process the next text character T_j, vector C first corresponds to column $j - 1$. Let C' denote its updated version that corresponds to the values in column j. When we move to the next column $j+1$, C' becomes C, and the new C' will correspond to the updated values in column $j+1$, and so on. Following the recurrence for M, the update formula for C is

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

for all $i > 0$. We report an occurrence ending at text position j whenever $C'_m \leq k$ immediately after processing the column corresponding to T_j.

Several properties of matrix M are discussed in [21]. 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 set $\{-1, 0, +1\}$. Also, $M_{i+1,j+1} - M_{i,j}$ is in the set $\{0, 1\}$.

Figure 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**

Figure 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 [22] Ukkonen observed that dynamic programming matrix values larger than k can be assumed to be $k+1$ without affecting the output of the computation. Moreover, once $M_{i,j} > k$, it is known that $M_{i+1,j+1} > k$. Cells of C with value not exceeding k are called *active*. In the algorithm, the row index ℓ of the last active cell (i.e., largest i such that $C_i \leq k$) is maintained (let us assume $\ell = -1$ if $C_i > k$ for all i). All the values $C_{\ell+1\ldots m}$ are assumed to be $k+1$, and we know that also the updated values $C'_{\ell+2\ldots m}$ will be larger than k. So C' needs to be updated only in the range $C'_{1\ldots\ell+1}$.

The value ℓ has to be updated throughout the computation. Initially $\ell = k$ because $C_i = M_{i,0} = i$. The row index of the last active cell can increase by at most one when moving to the next column. So we may first check whether $C'_{\ell+1} \leq k$, and in such a case we increment ℓ. If this is not the case, we search upwards for the new last active cell by decrementing ℓ as long as $C'_\ell \leq k$. Despite that this search can take $O(m)$ time at a given column, we cannot work more than $O(n)$ overall. There are at most n increments of ℓ in the whole process, and hence there cannot be more than $n+k$ decrements. Thus the row index ℓ of the last active cell is maintained at $O(1)$ amortized cost per column.

In [4] it was shown that on average $\ell = O(k)$, and therefore Ukkonen's cutoff scheme runs in $O(kn)$ expected time.

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 Figure 2. Each of the $k+1$ rows 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 the last column of the right table in Figure 1).

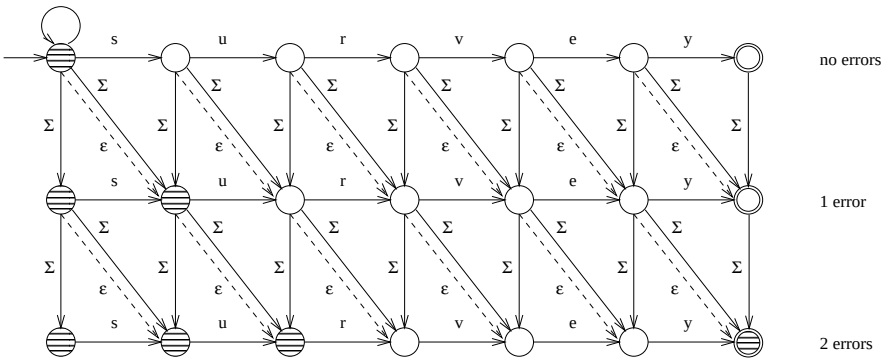

Figure 2: An NFA for approximate string matching of the pattern `"survey"` with two differences. The shaded states are those active after reading the text `"surgery"`.

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 [24] is to simulate the NFA of Figure 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:

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

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

If $m + 1 > w$, we need $\lceil m/w \rceil$ computer words to simulate every R_i mask[1] and have to update

[1]With slightly more complicated formulas, the simulation can be done using m bits, instead of $m+1$.

them one by one. The cost of this simulation is thus $O(k\lceil m/w\rceil n)$. The algorithm is flexible. For example the initial self-loop can be removed by changing the update formula into:

$$\begin{aligned} R'_0 &\leftarrow (R_0 << 1) \;\&\; B[T_j], \\ R'_{i+1} &\leftarrow ((R_{i+1} << 1) \;\&\; B[T_j]) \;\mid\; R_i \;\mid\; (R_i << 1) \;\mid\; (R'_i << 1). \end{aligned}$$

2.7 Myers' Bit-Parallel Matrix Simulation (BPM)

A better way to parallelize the computation [12] 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 [21].

We present a version [9] that differs slightly from that of [12]: 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 [9, 17], one can arrive to identities that permit computing their new values (at j) from their old values (at $j-1$) fast.

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

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.

```
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
```

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

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 (Figure 4). 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 [16] 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.

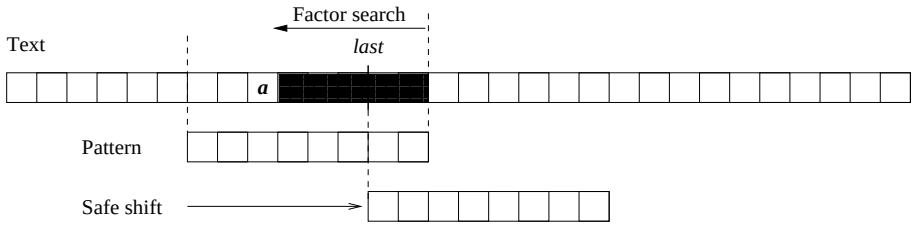

Figure 4: BDM search scheme.

We modify the NFA of Figure 2 so that it recognizes not only the whole pattern but also any suffix thereof, allowing up to k differences. Figure 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.

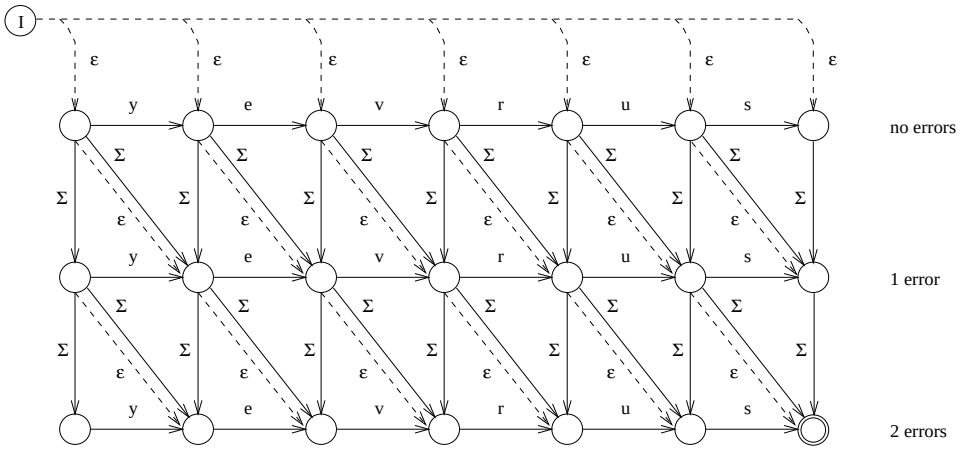

Figure 5: An NFA to recognize suffixes of the pattern `"survey"` reversed.

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 propagate 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 complete pattern occurrence starting at the beginning of the window. For this sake, we run the traditional automaton that computes edit distance (i.e., that of Figure 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. The first one makes the *backward scanning*, recognizing suffixes of P^r. The second one makes the *forward scanning*, recognizing P.

The automata can be simulated in a number of ways. In [16] they choose BPA [24] because it is easy to adapt to the new scenario. To recognize all the suffixes, we just need to initialize $R_i \leftarrow 1^{m+1}$. To make it compute edit distance, we remove the self-loop as explained in Section 2.6. The final state is active when $R_k \ \& \ 10^m \neq 0^{m+1}$. The NFA is out of active states whenever $R_k = 0^{m+1}$. Other approaches were discarded: an alternative NFA simulation [3] is not practical to compute edit distance, and BPM [12] cannot easily tell when the corresponding automaton is out of active states, or similarly, when all the cells of the current dynamic programming column are larger than k.

Figure 6 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.      pos ← 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
```

Figure 6: 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.

3 Average Case Analysis of ABNDM

The best previous analysis of ABNDM [10] (which improved the first one [13]) has shown that the algorithm inspects on average $O(kn \log_\sigma(m)/m)$ text positions. We show now that those analyses are pessimistic, and that the number of character inspections made by ABNDM is indeed the optimal $O((k + \log_\sigma m)n/m)$, and this holds for $\alpha < 1/2 - O(1/\sqrt{\sigma})$. This will be essential to analyze the new algorithms we present in the following sections.

We analyze a simplified algorithm that can never inspect less characters than the real ABNDM algorithm. We will show that even this simplified algorithm is optimal. The simplified algorithm always inspects ℓ characters from the window (ℓ will be determined later), and only then it checks whether the string read matches inside P with k errors or less. If the string does not match, the window is shifted by $m - k - \ell$ characters. If it matches, the whole window is verified and shifted by one position. It is clear that this algorithm can never perform better than the original in any possible text window. If the original algorithm stops the scanning before reading ℓ characters (and hence shifts more than $m - k - \ell$ positions), the current algorithm reads ℓ characters and shifts $m - k - \ell$ positions. Otherwise, the simplified algorithm goes to the worst possible situation: it checks the whole window and shifts by one position.

Let us consider the $n-(m-k)+1 \leq n$ text windows of length $m-k$. We divide them into *good* and *bad* windows. A window is good if its last ℓ characters do not match inside P with k errors or less, otherwise it is bad. We will consider separately the cost to process good and bad windows.

When the search encounters a good window, by definition, it inspects ℓ characters and shifts $m - k - \ell$ positions. Therefore, we cannot process more than $\lceil n/(m - k - \ell) \rceil$ good windows, at $O(\ell)$ cost each. Therefore, the overall number of inspected characters inside good windows is $O(\ell n/(m - k - \ell))$.

In order to handle the bad windows, we must bound the probability of a window being bad. In [3, 13] it is shown that the probability that a given string of length ℓ matches at a given (final) position inside a longer string is a^ℓ/ℓ, where $a < 1$ whenever $k/\ell < 1 - e/\sqrt{\sigma}$, that is, we need at least $\ell > k/(1 - e/\sqrt{\sigma})$. An upper bound to the probability of the string of length ℓ matching inside P is obtained by adding up the m possible final match positions of the string inside P, as if the events of matching at different final positions were disjoint and the first $\ell + k - 1$ positions did not have lower probability of finishing a match. Hence, an upper bound to the probability of a window being bad is ma^ℓ/ℓ.

When the window is bad, we pay at most $(m - k) + (m + k) = 2m$ character inspections for scanning and verifications, and then shift by one. Since there are at most n bad windows in the text, an upper bound to the overall average number of characters inspected on bad windows is $n \cdot ma^\ell/\ell \cdot 2m = O(m^2 n a^\ell/\ell)$. This upper bound is obtained by assuming that we will consider *all* the text windows, and pay $2m$ for all the bad ones.

We choose ℓ large enough so that the cost of bad windows does not exceed $O(n/m)$, so as to ensure that the cost on good windows dominates. For this to hold, we need $a^\ell/\ell \leq 1/m^3$, or more strictly, $a^\ell \leq 1/m^3$. This is equivalent to $\ell \geq 3 \log_{1/a} m$. Since $a \geq 1/\sigma$ [3], a sufficient condition is $\ell \geq 3 \log_\sigma m$.

Therefore, we have that the overall number of characters inspected is $O(\ell n/(m-k-\ell))$ provided $\ell > k/(1 - e/\sqrt{\sigma})$ and also $\ell \geq 3 \log_\sigma m$. The complexity is $O(\ell n/m)$ provided $m - k - \ell \geq cm$ for some constant $0 < c < 1$, that is, $\ell \leq (1 - c)m - k$. So we have lower and upper bounds

on ℓ. The first condition we can derive from both is $k/(1 - e/\sqrt{\sigma}) < (1 - c)m - k$, that is, $\alpha < (1 - e/\sqrt{\sigma})/(2 - e/\sqrt{\sigma}) = 1/2 - O(1/\sqrt{\sigma})$. Since $k < m/2$, $(1 - c)m - k > (1/2 - c)m$ and therefore this upper bound on ℓ does not clash, asymptotically, with the lower bound $\ell \geq 3 \log_\sigma m$.

So we have that the complexity is $O(\ell n/m)$ provided $\alpha < 1/2 - O(1/\sqrt{\sigma})$ and $\ell > \max(k/(1 - e/\sqrt{\sigma}), 3 \log_\sigma m)$. Choosing an appropriate ℓ we obtain complexity $O(\max(k, \log_\sigma m)n/m) = O((k + \log_\sigma m)n/m)$, which is optimal [5]. This shows that our pessimistic analysis is tight and that ABNDM inspects an optimal (on average) amount of characters.

ABNDM, however, is not optimal in terms of overall complexity. The reason is that, for each character inspected, the BPA automaton needs time $O(k)$ to process it if $m \leq w$, and $O(mk/w)$ in general. This gives an overall complexity of $O((k + \log_\sigma m)kn/m)$ if $m \leq w$, and $O((k + \log_\sigma m)kn/w)$ in general.

In this paper we manage to use BPM instead of BPA. This simulation takes $O(1)$ per inspected character if $m \leq w$, and $O(m/w)$ in general. In this case the complexity would be the optimal $O((k + \log_\sigma m)n/m)$ for $m \leq w$ and $O((k + \log_\sigma m)n/w)$ in general. However, as we show later, different complications make the real complexities $O((k + \log m)n/m)$ and $O((k + \log m)n/w)$. These are optimal for constant alphabet size σ.

4 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.

4.1 Computing Edit Distance

We recall that BPM implements the dynamic programming algorithm of Section 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 Section 2.3, the only change is that $M_{0,j} = j$. In differential terms (Section 2.7), this means $\Delta h_{0,j} = 1$ instead of zero.

When $\Delta h_{0,j} = 0$, its value does not need to be explicitly present in the BPM algorithm. The value makes a difference only when HP or HN is shifted left, which happens on lines 12 and 14 of the algorithm (Figure 3). On these occasions the assumed bit zero enters automatically from the right, thereby implicitly using a value $\Delta h_{0,j} = 0$. To use a value $\Delta h_{0,j} = 1$ instead, 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 Figure 7 the code for a single step of edit distance computation.

4.2 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 larger

```
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)
```

Figure 7: The procedure used for performing the variable updates per scanned character in the adaptation of BPM to edit distance computation. It receives the bit mask Bc of the current text character and shares all the other variables with the calling process.

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 $C_i = \sum_{r=1\ldots i}(VP_r - VN_r)$. Using this incremental representation, it is not easy to check whether $C_i > k$ for all i.

Our solution is inspired by the cutoff algorithm of Section 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 $\ell \geq 0$ holds if and only if $C_i \leq k$ for some i. Hence we will maintain a *witness* in the current column of the dynamic programming matrix that will tell which is the last cell not exceeding k.

So we have to figure out how to maintain ℓ using BPM. Initially, since $C_i = M_{i,0} = i$, we set $\ell \leftarrow k$. Later, we have to update ℓ for each new text character read. Recall that neighboring cells in M (and hence in C) differ by at most one. Since, by definition of ℓ, $C_{\ell+1} > k$ and $C_\ell \leq k$, we have that $C_\ell = M_{\ell,j-1} = k$ as long as $\ell < m$. We may assume that $k < m$, so the condition $\ell < m$ holds initially. We consider now how to move from column $j-1$ to column j in M.

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 the bit $D0_{\ell,j}$ is set *after* the increment. If it was not correct to increase ℓ, we decrease it as much as necessary to obtain $M_{\ell,j} \leq k$. In this case we know that $M_{\ell,j} = k+1$, which enables us to obtain the cell values $M_{\ell-1,j} = M_{\ell,j} - VP_{\ell,j} + VN_{\ell,j}$, 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 procedure assumed that $\ell < m$. Note that, as soon as $\ell = m$, we have $C_m \leq k$, and the forward scan will terminate because we have found an occurrence.

Figure 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 table B

already computed (see Figure 3). Note that for efficiency ℓ is maintained in unary.

```
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
```

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

5 Backward Scanning with the BPM Simulation

In this section we address the main obstacle to use BPM instead of BPA inside algorithm ABNDM: the backward scanning. As explained, the problem is that the backward scanning algorithm should be able to tell, as early as possible, that the string read up to now cannot be contained in any pattern occurrence, so as to shift the window as early as possible. Under BPA simulation it turns out that the condition is equivalent to the simulated NFA not having any active state, and this can be directly checked because the NFA states are explicitly represented in BPA. BPM, on the other hand, does not simulate an automaton, but the dynamic programming matrix. In this case, the condition is equivalent to all matrix values in the last column exceeding k. The problem is that BPM does not store absolute matrix values, but differential ones, and this makes it difficult to tell fast whether all cell values exceed some threshold.

We solve the problem by introducing the *witness* concept. A witness is a matrix cell whose absolute value is known. Several witnesses are spread along the current matrix column. All the witness values are maintained in a single computer word and updated in bit-parallel fashion. By knowing the absolute values of some column cells, we can efficiently compute, bound, or deduce all the other column values. When all the values can be proven to exceed k, we know that the current window can be abandoned.

We first develop a naive solution, based on the forward scanning developed in Section 4. This method uses one witness to stop the scanning, and we will show why this cannot be efficient in

this scenario. This fact will motivate the use of several witnesses, which will be developed in depth next.

5.1 A Naive Solution

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 Figure 3: $VP = 1^m$ corresponds to $C_i = i$. 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 to line 12 the same change as with forward scanning in order to use the value $\Delta h_{0,j} = 1$.

With these tools at hand, we could simply apply the forward scan algorithm with B built on P^r and read the window backwards. We could use witness 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*.

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 Figure 3, until $diff > k$. At this moment we switch to updating ℓ again, moving it upwards as necessary.

The above scheme works correctly but 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 to maintain the last active cell, given in Section 2.4, becomes here $O(m + k + \log_\sigma m)$, since now ℓ starts at m instead of k and the "text" length is $O(k + \log_\sigma m)$. Hence, all the column cells reach a value larger than k quite soon, and ℓ goes down to zero, correspondingly. The problem is that ℓ needs too much time to go down to zero. That is, our witness has to traverse all the m cells to determine that all of them exceed k.

We present two solutions to determine fast that all the C_i values have surpassed k. Both solutions rely on maintaining several witnesses at the same time along the matrix column. The general idea is to maintain a denser sample of the absolute values in order to reduce the time needed to traverse all the non-sampled cells. In the first version we develop, it might be that we inspect more window characters than necessary in order to determine that we can shift the window. In the second, we will examine the minimum number of characters required, but will have to work more per character, in order to make use of the witnesses. Both will obtain the same search complexity by different means.

5.2 Bit-Parallel Witnesses

In the original BPM algorithm, the integer value $diff = C_m$ (a witness) is explicitly maintained in order to determine which text positions match. This is accomplished by using 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 witnesses in the same computer word MC and use them to bound the others.

General mechanism. Let Q denote the space, in bits, that will be reserved for a single witness. We set up $t = \lceil m/Q \rceil$ consecutive witnesses into MC. The witnesses keep track of the values C_m, C_{m-Q}, C_{m-2Q}, $\ldots$, $C_{m-(t-1)Q}$, and the witness for C_{m-rQ} uses the bits $m - rQ \ldots m - rQ + Q - 1$ in MC. This means that we need $m + Q - 1$ bits for MC[2]. We discuss later how to determine a suitable value Q. For now we assume that such a Q has already been determined.

The witnesses can be used as follows. We note that every cell is at most $\lfloor Q/2 \rfloor$ positions away from some represented witness, and it is known that the difference between consecutive cell values is at most 1. Thus we can be sure that all the cell values of C exceed k when all the witness values are larger than $k' = k + \lfloor Q/2 \rfloor$.

The preceding assumption that every cell in C is at a distance of at most $\lfloor Q/2 \rfloor$ to a represented cell may not be true for the first $\lfloor Q/2 \rfloor$ cells. But we know that $C_0 = j$ at the j-th iteration, and so we may assume there is an implicit witness at row zero. Moreover, since this witness is always incremented, it is at least as large as any other witness, and so it will surely surpass k' when the other witnesses do. The initial $\lfloor Q/2 \rfloor$ cells are close enough to this implicit witness.

So the idea is to traverse the window until all the witnesses exceed k', and then shift the window. We will examine a few more cells than if we had controlled exactly all the C values. We analyze later the resulting complexity.

Figure 9 shows the pseudocode of the algorithm. The name of the algorithm owes to the fact that the witness positions are fixed, as opposed to the next section.

Implementation. In implementing this idea we face two problems. The first one is how to update all the witnesses in a single operation. This is not hard because each witness 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 all witnesses in parallel by setting $MC \leftarrow MC + (HP \;\&\; sMask) - (HN \;\&\; sMask)$ (lines 10 and 20 in Figure 9).

The second problem is how to determine that all the witnesses have exceeded k'. For this sake we store each witness with excess $b = 2^{Q-1} - 1 - k'$. That is, when $C_{m-rQ} = x$, the corresponding witness holds the value $x + b$. This way the Q-th bit of a witness is activated when the cell value it represents exceeds k'. Thus if we define $eMask = (10^{Q-1})^t 0^{m+Q-1-tQ}$, then we can stop the scanning whenever $MC \;\&\; eMask = eMask$, that is, when all witnesses have their Q-th bits activated (lines 11 and 18 in Figure 9).

Determining Q. Let us now explain how to determine the value Q for the number of bits reserved for each witness. Clearly Q should be as small as possible. The criteria for Q are as follows. First of all we need that (i) $b + k' + 1 = 2^{Q-1}$, where b is the excess value. Initializing the witnesses to b allows us to determine, from their Q-th bits, that a witness has exceeded $k' = k + \lfloor Q/2 \rfloor$. On the other hand, we have to ensure that the Q-th bit remains set for any witness value larger than k', and that Q bits are still enough to represent the witness. Since the upper limit for a cell value in the window is $m - k$, the preceding is guaranteed by the condition (ii) $b + m - k < 2^Q$. Finally, the excess cannot be negative, and so we need (iii) $b \geq 0$.

[2]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.

```
ABNDMFixedWitnesses (P_{1...m}, T_{1...n}, k)
1.    Preprocessing
2.        For c ∈ Σ Do Bf[c] ← 0^m , Bb[c] ← 0^m
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 ← ⌈log_2(m - k + 1)⌉
7.        If 2^{Q-1} < max(m - 2k - ⌊Q/2⌋, 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.       pos ← 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
```

Figure 9: The **ABNDM** algorithm using bit-parallel witnesses. 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 any more at that point.

By replacing (i) in (ii) we get (i') $b = 2^{Q-1} - k' - 1$ and (ii') $m - k - k' \leq 2^{Q-1}$. By (iii) and (i') we get (iii') $k' + 1 \leq 2^{Q-1}$. Hence the solution to the new system of inequalities is $Q = 1 + \lceil \log_2(\max(m - k - k', k' + 1)) \rceil$, and $b = 2^{Q-1} - k' - 1$.

The problem with the above solution is that $k' = k + \lfloor Q/2 \rfloor$, so the solution is indeed a recurrence for Q. Fortunately, it is easy to solve. Since $(X + Y)/2 \leq \max(X, Y) \leq X + Y$ for any nonnegative X and Y, if we call $X = m - k - k'$ and $Y = k' + 1$, we have that $X + Y = m - k + 1$. So $Q \leq 1 + \lceil \log_2(m - k + 1) \rceil$, and $Q \geq 1 + \lceil \log_2((m - k + 1)/2) \rceil = \lceil \log_2(m - k + 1) \rceil$. This gives a 2-integer range for the actual Q value. If $Q = \lceil \log_2(m - k + 1) \rceil$ does not satisfy (ii') and (iii'), we use $Q + 1$ (lines 6–8 in Figure 9).

This scheme works correctly as long as $X, Y \geq 0$, that is, $\lfloor Q/2 \rfloor \leq m - 2k$, or $m - k \geq k'$. If this does not hold, our method is anyway useless since in that case it will have to verify every text window.

Example. Figure 10 shows an example of how the vector MC is set up. All the bit masks are of length m, except $sMask$, $eMask$ and MC, which are of length $m + Q - 1$.

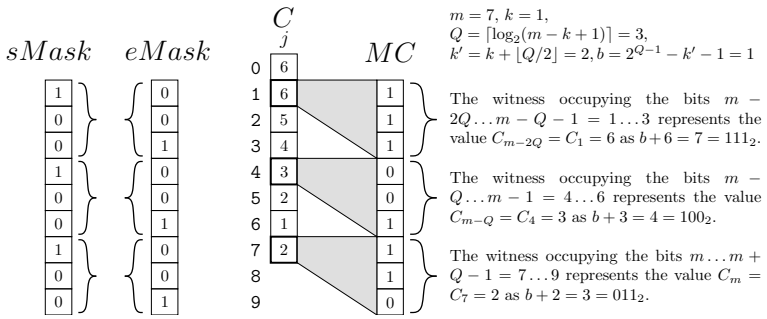

Figure 10: An example of vectors $sMask$, $eMask$ and MC when $m = 7$ and $k = 1$. In this case $Q = 3$, $k' = 2$ and $b = 1$. In the middle we show a possible column C at position j, on the left the vectors $sMask$ and $eMask$, and on the right the corresponding composition of the vector MC at column j. The curly braces point out the bit-regions of the witnesses. The only witness whose Q-th bit is not activated corresponds to value $C_7 = 2 \leq k'$.

Complexity. Let us analyze the complexity of the resulting algorithm. The backward scan will behave as if we permitted $k' = k + \lfloor Q/2 \rfloor$ differences, so the number of characters inspected is $\Theta(n(k + \log m + \log_\sigma m)/m) = \Theta(n(k + \log m)/m)$. 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.

In case our upper bound $k' = k + \lfloor Q/2 \rfloor$ turns out to be too loose, we can use several interleaved sets of witnesses, each set in its own bit-parallel mask. For example we could use two interleaved MC masks and hence the limit would be $k + \lfloor Q/4 \rfloor$. In general we could use c masks and have a limit of the form $k + \lfloor Q/2^c \rfloor$. The cost would be $O(c(k + \log(m)/2^c + \log_\sigma m)n/m)$, which is optimized for $c^* = \log_2(\log(m)/(k + \log_\sigma m))$. Using this optimum, the complexity is $O((k + \log_\sigma m)c^* n/m)$, which means the almost optimal $O((k + \log_\sigma m) \log\log(\sigma) n/m)$ when $k = O(\log_\sigma m)$, the almost optimal $O((k + \log_\sigma m) \log(\log(m)/k)n/m)$ when $\Omega(\log_\sigma m) = k = O(\log m)$, and the optimal $O(kn/m)$ for $k = \Omega(\log n)$. Hence we are very close to optimal under this scheme. Indeed, the algorithm is optimal if we assume that σ is constant.

5.3 Bit-Parallel Cutoff

The previous technique, although simple, has the problem of inspecting more characters than necessary. We can instead produce, using a similar approach, an algorithm that inspects the optimal number of characters. This time the idea is to mix the bit-parallel witnesses with a bit-parallel version of the cutoff algorithm (Section 2.4). The final complexity, however, will be the same as for the previous technique, for reasons that will be clear soon.

General mechanism. Consider regions $m - rQ - Q + 1 \ldots m - rQ$ of length Q. Instead of having the witnesses fixed at the end of each region (as in the previous section), we let the witnesses "float"

inside their region. The distance between consecutive witnesses 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 meanings as before, but they are displaced so as to be all the time aligned to the witnesses.

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

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

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 witness has exactly the value C_m.

In the following we present the details of the witness processing after each text window character is read. Figure 11 shows the pseudocode for the whole algorithm.

Implementation. The floating witnesses are a bit-parallel version of the cutoff technique, where each witness takes care of its region. Consequently the way of moving the witnesses up and down resembles the cutoff technique (Section 2.4). We first move down and use $D0$ to update MC accordingly (lines 22–23 in Figure 11). But maybe we should not have moved down. Moreover, maybe we should move up several times. So, after having moved down, we move up as much as necessary by using VP and VN (lines 24–26 in Figure 11). To determine whether we should move up further, we need to know whether there is a witness that exceeds k. We proceed as in Section 5.2, using $eMask$ to determine whether some witness exceeds k. We also use $sMask$ to increment and decrement the witness values. Q is computed as in Section 5.2, except that $k' = k$ and hence no recurrence arises (lines 6–10 in Figure 11).

Note that we have to deal with the case where the witnesses are at the end of their region and hence cannot move down further. In this case we update them using HP and HN (line 20 in Figure 11).

Finally, it is also possible that the upmost witness goes out of bounds while shifting the witnesses, which in effect results in that witness being removed. For this to happen, however, all the area in C covered by the upmost witness must have values larger than k, and it is not possible that a cell in this area gets a value $\leq k$ later. So this witness can be safely removed from the set, and hence we remove it from $eMask$ as soon as it gets out of bounds for the first time (line 27 in Figure 11). Note that ignoring this fact leads to inspecting slightly more characters (an almost negligible amount) but one instruction is saved, which in practice is convenient.

Example. Figure 12 shows an example of floating the witnesses upwards in vector MC.

Complexity. Let us consider the resulting time complexity. As for the case of a single witness, 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 $O(u + Q) = O(k + \log m)$ on average because $Q = O(\log m)$, and therefore we obtain the same complexity of Section 5.2 (without the possibility of tuning c).

```
ABNDMFloatingWitnesses (P_{1...m}, T_{1...n}, k)
1.  Preprocessing
2.      For c ∈ Σ Do Bf[c] ← 0^m , Bb[c] ← 0^m
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, k + 1))⌉
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.     pos ← 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]^t_Q 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.                 If δ = m − (t − 1)Q Then eMask ← eMask & 1^{(t-1)Q}0^{m+2Q-1-tQ}
28.             j ← j − 1
29.             If δ = 0 AND MC & 10^{m+Q-2} ≠ 0^{m+Q-1} Then /* prefix recognized */
30.                 If j > 0 Then last ← j
31.                 Else If BPMFwd (Bf, T_{pos+1...n}) Then
32.                     Report an occurrence at pos + 1
33.         pos ← pos + last
```

Figure 11: The **ABNDM** algorithm using bit-parallel cutoff. The same comments of Figure 9 apply. For efficiency, the witnesses are not physically shifted, but instead we shift $D0$, VN and VP by δ.

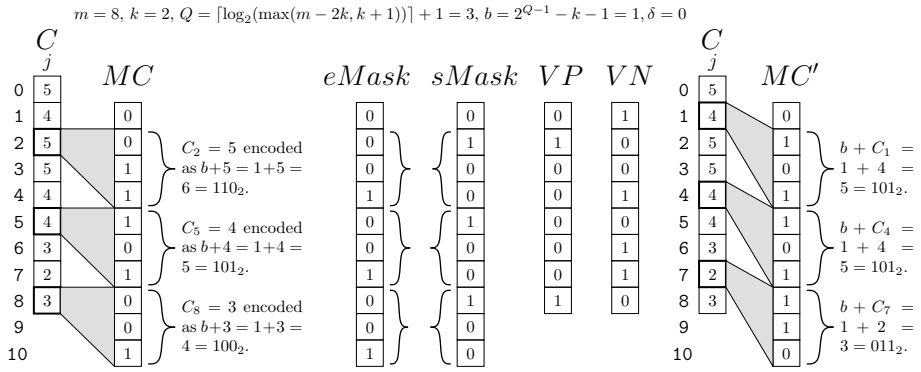

Figure 12: The left side shows a situation where the witnesses are in their original position ($\delta = 0$), and the equality $eMask \ \& \ MC = eMask$ indicates that all witnesses have exceeded $k = 2$. Now we let the witnesses float upwards by incrementing δ as long as $\delta < Q$ and no witness $\leq k$ has been found. When δ is incremented, the witnesses in MC get their above-neighbor values. We show the new situation on the right. Then the new witness values are evaluated by again checking whether $eMask \ \& \ MC' = eMask$. In this example only one increment of δ was needed, as the last witness found the value $C_7 = 2 \leq k$.

We also tried a different version of this algorithm, in which the witnesses are not shifted. Instead, they are updated in a similar fashion to the algorithm of Figure 9, and when all witnesses 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, $O(Q(k + \log_\sigma m)) = O(\log m(k + \log_\sigma m))$ per window, as at each processed character it is possible to make $O(Q)$ shifts. But in practice it turned out to be very similar to our original cutoff algorithm.

6 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 [13], 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 maximum optimization switch. 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 [24], using a generic implementation for any k.

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

ABNDM/BPM(fixed): ABDNM implemented using BPM and fixed-position witnesses, without the interleaving mentioned at the end (Section 5.2). The implementation differed slightly from Figure 9 due to optimizations.

ABNDM/BPM(floating): ABDNM implemented using BPM and cutoff, with floating-position witnesses (Section 5.3). The implementation differed slightly from Figure 11 due to optimizations.

BPM: The sequential BPM algorithm [12]. The implementation was by us and used the slightly different (but practically equivalent in terms of performance) formulation from [9].

BPP: A combined heuristic using pattern partitioning, superimposition and hierarchical verification, together with a diagonally bit-parallelized NFA [3, 15]. The implementation was by 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 [14]. The implementation was by the original authors.

Figure 13 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[3], and $\sigma = 52$ to show that our algorithms are useful even on large alphabets.

First of all it can be seen that ABNDM/BPM(floating) is always faster than ABNDM/BPM(fixed) 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 interesting when searching natural language text, in particular when searching for phrases.

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).

[3]On biased texts, most sequential string matching algorithms behave as on random texts of size σ, where $1/\sigma$ is the probability that two characters randomly chosen from the text match. On English texts this probability is usually between 1/12 and 1/15.

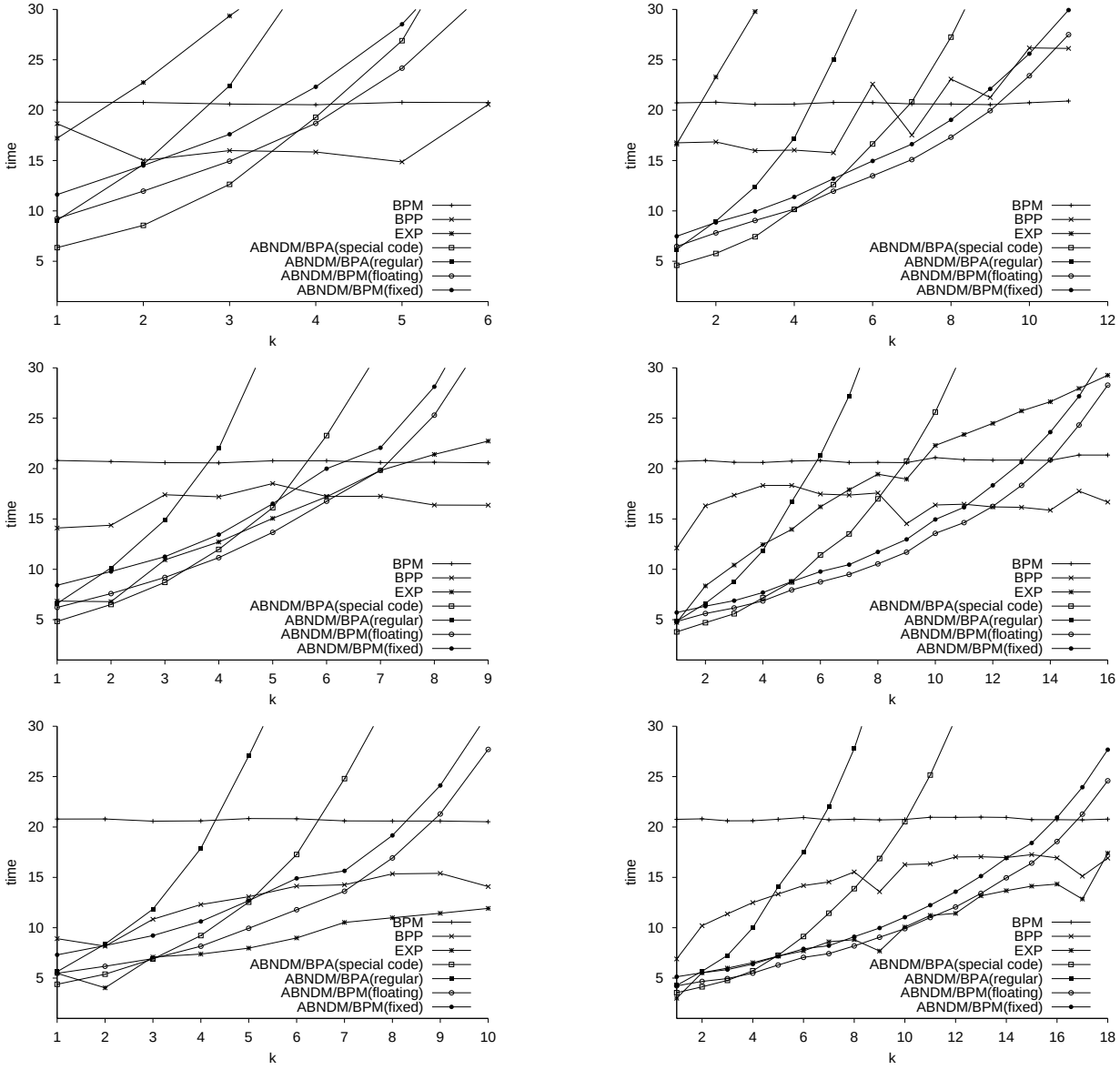

Figure 13: Comparison between algorithms, showing total elapsed time as a function of the number of differences permitted, k. From top to bottom row we show $\sigma = 4$, 13 and 52. On the left we show $m = 30$ and on the right $m = 55$.

7 Using Bit-Parallel Cutoff in Row-wise BPM

In this section we demonstrate that the idea of using witnesses can be applied to other scenarios. We consider a recent work [7], where the basic BPM algorithm is modified so that the dynamic programming matrix is filled row-wise rather than column-wise. This means that the text $T_{1...n}$ is cut into consecutive chunks of w characters, and for each chunk $T_{bw+1...bw+w}$ we compute the m rows of the corresponding part of the dynamic programming matrix, so that each row of each chunk is computed in $O(1)$ time using a variant of BPM.

Figure 14 illustrates the idea. The shaded area represents the real area of the dynamic programming matrix C that must be filled. Classical BPM fills it column-wise. If m mod w is not small, an important amount of work is wasted. This corresponds to work that is anyway carried out inside the bit masks. It is represented by the non-shaded area that is covered by the vertical rectangles. If the same matrix is filled row-wise, we need much less rectangles (bit-parallel steps) to cover the same matrix.

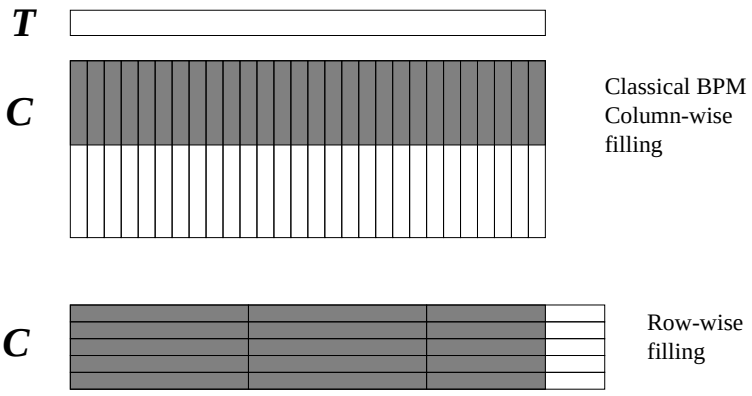

Figure 14: Column-wise versus row-wise bit-parallel filling of dynamic programming matrix C.

Modifying BPM to work by rows instead of by columns is rather easy because the rules to compute C are symmetric, so the formula for the transposed matrix is exactly the same. The only difference is that now the first row must start with all zeros, while the first value of row i is i. The changes are simple and have already been done in this paper for other purposes (the first to recognize any suffix of P, the second to compute edit distance using BPM). The really challenging part is how to preprocess the characters of the current text chunk efficiently, because the B table of a text chunk will be used just for m bit-parallel steps, unlike the pattern preprocessing of column-wise filling, that is used for all the n steps. The details are given in [7]. In particular, it is shown there how to build the B table efficiently in the case of searching DNA.

To take much more advantage of row-wise tiling, the cutoff technique is used in [7], so that only the necessary rows of each chunk are filled. For this sake, it is necessary to determine whether all the current row values exceed k, so that no more rows need to be evaluated in the current chunk. The approach of [7] is to use precomputed tables $Sum(q)$ and $Min(q)$ that are two-dimensional and of size $2^q \times 2^q$, where the parameter q is chosen so that the word size w is a multiple of q. Let I be a length-q vector in which a set bit denotes an increment by one at that position, and in similar fashion let D be a length-q vector in which a set bit denotes a decrement by one. The value

$Sum(q)_{I,D}$ gives the combined increment of I and D, that is, $Sum(q)_{I,D} = \sum_{i=1\ldots q}(I[i]-D[i])$. The value $Min(q)_{I,D}$ gives the minimum combined increment between equally long prefixes of I and D, that is, $Min(q)_{I,D} = \min(\sum_{i=1\ldots h}(I[i] - D[i]) \mid 1 \leq h \leq q)$. In the case of using BPM in row-wise manner, the roles of the vertical and the horizontal difference bit masks are reversed. Consider a situation where the length-w vertical bit masks VP and VN encode the horizontal differences $\Delta h_{i,j+1}, \Delta h_{i,j+2}, \ldots, \Delta h_{i,j+w}$ and the cell value $M_{i,j}$ of the dynamic programming matrix is known. Let the superscript h denote the hth length-q segment of a length-w bit mask. Now for example $VP = VP^1 \ldots VP^{w/q}$, and VP^h contains the bits $VP_{hq-q+1} \ldots VP_{hq}$. The definition of $Sum(q)$ means that $M_{i,j+q} = M_{i,j}+Sum(q)_{VP^1,VN^1}$. From the definition of $Min(q)$ we have that $M_{i,j+x} \leq k$ for some $1 \leq x \leq q$ if and only if $M_{i,j} + Min(q)_{VP^1,VN^1} \leq k$. Repeating the preceding w/q times is enough to check whether the whole region $M_{i,j+1} \ldots M_{i,j+w}$ contains a cell value not greater than k: First check the segment $M_{i,j+1} \ldots M_{i,j+q}$ by using the value $M_{i,j} + Min(q)_{VP^1,VN^1}$. Then compute the value $M_{i,j+q} = M_{i,j} + Sum(q)_{VP^1,VN^1}$. After checking the hth segment, the $(h+1)$th segment $M_{i,hq+1} \ldots M_{i,j+(h+1)q}$ can be checked by using the value $M_{i,j+hq} + Min(q)_{VP^{h+1},VN^{h+1}}$, and one can also compute the value $M_{i,j+(h+1)q} = M_{i,j+hq} + Sum(q)_{VP^{h+1},VN^{h+1}}$ for subsequent use in the checking process.

Consider filling the chunk of rows that corresponds to $T_{j+1\ldots j+w}$. Our proposal is to use bit-parallel witnesses (Section 5.3) in implementing the cut-off in row-wise BPM. This is quite straightforward as the case of filling a single chunk of rows in row-wise BPM is very similar to the case of backward scanning in ABNDM. The only differences are that the roles of the text chunk and the pattern are reversed, the boundary values $\Delta h_{0,j}$ depend on the previous chunk of rows, the "window-length" is m instead of $m-k$, and the witness size Q is determined so that the maximum value a witness needs to be able to hold is $\min(m, k+w)$ instead of $m-k$. The last part comes from the fact that the minimum value within a row that needs to be computed is at most $k+1$, and thus the maximum value within a length-w chunk is at most $k+1+(w-1) = k+w$.

We have tested modifying row-wise BPM to use the variant of our bit-parallel cutoff that was fastest in the previous section. The modification was built on the original code from [7] that builds the B table efficiently in the case of DNA searching. A prerequisite for this is that the DNA text has to be packed in a special way. The packed DNA takes 2 bits per character (see [7] for details of the packing scheme). We used the roughly 10MB genome of baker's yeast as the text, and the patterns were selected from the text in random fashion. The tested pattern lengths were $m =$ 16, 32 and 64, and for each combination of m and k we measured the average over searching 100 different patterns. The computer used in this test was a Sparc Ultra 2 without other significant processes running during the tests. The computer was setup in 64-bit mode and thus the chunks were of length $w = 64$.

We compared our modified version against the original, and for each version we measured both the elapsed run time and the average number of rows filled within the chunks. As mentioned in [7], the original row-wise BPM used a cut-off that requires the cell values to be larger than $k+1$ in order for them to become irrelevant. This way was claimed to be more convenient and also faster in practice. Our modified version uses the strict limit k.

The test included also row-wise BPM without cutoff and the regular BPM. The latter was an optimized version taking 75% of the time needed by the original version [12]. The former also used our bit-parallel witnesses to check fast whether row m in the current chunk contains a cell with

a value less than k, that is, whether we need to process the lowest row in the chunk cell-by-cell in order to report the pattern occurrences that end inside it. Note that this type of occurrence checking is extra work in comparison to the regular BPM. To take into account the possible gain from less I/O cost when the text is packed, we modified the regular BPM to search in a packed DNA where each character is encoded by two successive bits. Even though the size of the packed text is the same, this simple way of packing is not the same that the row-wise methods use.

Fig. 15 shows the results. It can be seen that row-wise BPM with our bit-parallel cutoff is considerably faster than the original row-wise method. This is true even if we compare our method with $k+1$ against the original with k to have a comparable number of filled rows due to the difference in the cutoff strategies. The plots also show that, when $m = 64$, the run time graphs of the two algorithms meet when $k = 21$. In that situation our row-wise BPM computes on average roughly 52 rows per chunk, and the time for checking/reporting occurrences is not a large factor. The original row-wise method starts being worse than plain BPM already from $k = 8$. With lower values of m the meeting point is before $k = 21$. This is because in those cases the row-wise methods begin to suffer from the cost of reporting occurrences at lower k values. This effect is also evident in how the graphs for row-wise BPM without cutoff have a distinctive step. The comparison among our two row-wise variants shows that the burden of using the cutoff is reasonably small: the version without cutoff is never much faster even when the cutoff method has to compute all or almost all of the m rows in each chunk.

Note that it is not possible to directly compare these results against those of Section 6, because here we use packed text and there we use standard text encoding. Packed text is necessary for the success of the algorithms of this section, while it is very cumbersome to handle by ABNDM. Yet, regular BPM can be used as a comparison ground between both algorithms. It can be seen that, on DNA text, this method is beaten by ABNDM variants only on $m = 64$ for rather low k values (which, however, are rather common in some applications).

8 Conclusions

The most successful approaches to approximate string matching are bit-parallelism and filtering. A promising algorithm combining both is ABNDM [16]. However, the original ABNDM uses a slow $O(k\lceil m/w\rceil n)$ time bit-parallel algorithm (BPA [24]) for its internal working because of its straightforward flexibility. In this paper we have shown how to extend BPM [12] to replace BPA. Since BPM is $O(\lceil m/w\rceil n)$ 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 than necessary, and a more complicated (and more costly per processed character) that reads exactly the required characters.

The main challenge faced was that we needed to act upon absolute values of the matrix cells, while BPM stores the information differentially. Our solution relies on a new concept called a *witness*. A witness is a matrix cell whose absolute value is known. Together with the differential

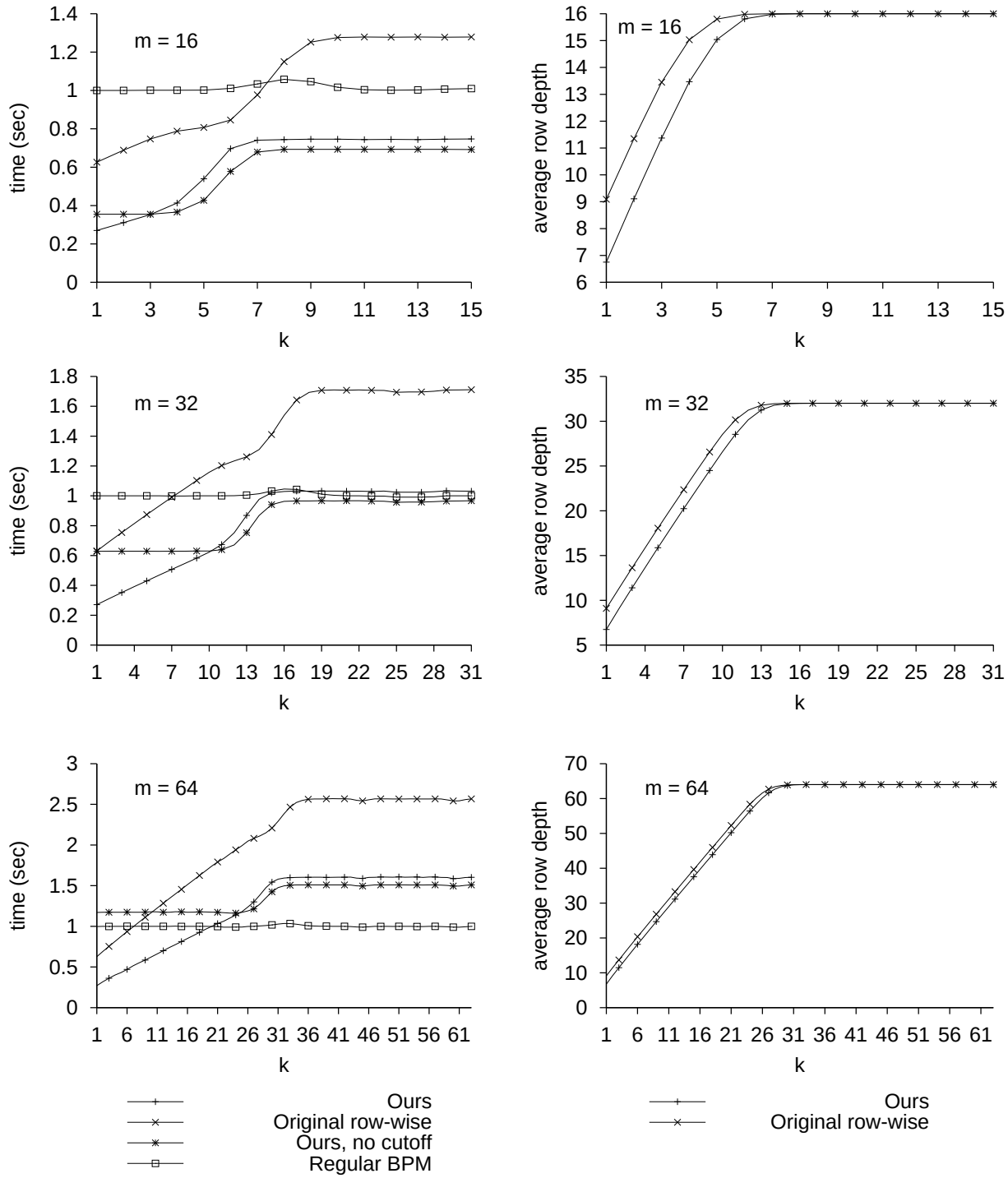

Figure 15: The three rows show the test results with the pattern lengths 16, 32 and 64. The left column shows the average time for searching for a pattern from baker's yeast, and the right column shows the corresponding average number of rows filled during the computation.

values, we update one or more witness values in parallel. Those witnesses are used to deduce, bound or compute all the other matrix values.

We present an improved average analysis of ABNDM that shows that it inspects the optimal number of characters, $O((k + \log_\sigma m)n/m)$. While the original ABNDM [16] is far from this optimality when its overall complexity is considered (not only inspected characters), our new ABNDM versions are much closer to the optimum, reaching average complexity $O((k + \log m)n/m)$. Indeed, this is optimal if we regard σ as a constant.

The experimental results show that our new algorithm beats the original 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 witnesses fit in a single computer word. This includes several interesting cases in searching DNA, natural language text, protein sequences, etc.

To demonstrate that the concept of witness can be applied to other scenarios, we apply it to a recent work that improves upon BPM by filling the matrix row-wise instead of column-wise [7]. A key part of the improved algorithm is the ability to stop when all the matrix cells exceed some value. We show that the use of witnesses provides a much faster solution than the original in [7].

Finally, we notice that the witness concept helps to solve the main problem that arises when trying to compute local score matrices [23] in a bit-parallel fashion. The formula to compute score permits increments and decrements in the score, but it is never let to run below zero. A reasonable simplification, useful for bit-parallel computation, is as follows

$$\begin{aligned} M_{i,0} &\leftarrow 0, \qquad\qquad M_{0,j} \leftarrow 0, \\ M_{i,j} &\leftarrow \text{if } (x_i = y_j) \text{ then } M_{i-1,j-1} + 1 \text{ else } \max(0, M_{i-1,j} - 1, M_{i,j-1} - 1, M_{i-1,j-1} - 1). \end{aligned}$$

An important obstacle preventing the bit-parallel computation of M is that we have to know when a cell value has become negative in order to make it zero. Therefore, we need to know the absolute cell values, a scenario where witnesses are the ideal solution. We are currently pursuing this idea.

Acknowledgements

We thank the comments of the reviewers, which helped making the paper more readable.

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 Annual Symposium on 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 Annual Symposium on Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 259–273, 1994.

[6] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[7] K. Fredriksson. Row-wise tiling for the Myers' bit-parallel dynamic programming algorithm. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*, LNCS 2857, pages 66–79, 2003.

[8] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM Journal on Computing*, 19(6):989–999, 1990.

[9] H. Hyyrö. Explaining and extending the bit-parallel algorithm of Myers. Technical Report A-2001-10, University of Tampere, Finland, 2001.

[10] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, LNCS 2373, pages 203–224, 2002.

[11] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989.

[12] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.

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

[14] G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, 72:65–70, 1999.

[15] G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate string matching. *Algorithmica*, 30(4):473–502, 2001.

[16] 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.

[17] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[18] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.

[19] 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.

[20] J. Tarhio and E. Ukkonen. Approximate Boyer-Moore string matching. *SIAM Journal on Computing*, 22(2):243–260, 1993.

[21] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[22] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[23] M. Waterman. *Introduction to Computational Biology*. Chapman and Hall, 1995.

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

[25] S. Wu, U. Manber, and G. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.