A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming

Gene Myers [*]

March 27, 1998

Abstract

The approximate string matching problem is to find all locations at which a query of length m matches a substring of a text of length n with k-or-fewer differences. Simple and practical bit-vector algorithms have been designed for this problem, most notably the one used in *agrep*. These algorithms compute a bit representation of the current state-set of the k-difference automaton for the query, and asymptotically run in $O(nmk/w)$ time where w is the word size of the machine (e.g. 32 or 64 in practice). Here we present an algorithm of comparable simplicity that requires only $O(nm/w)$ time by virtue of computing a bit representation of the *relocatable* dynamic programming matrix for the problem. Thus the algorithm's performance is independent of k, and it is found to be more efficient than the previous results for many useful choices of k and small m.

Moreover, because the algorithm is not dependent on k, it can be used to rapidly compute blocks of the dynamic programming matrix as in the 4-Russians algorithm of Wu, Manber, and Myers. This gives rise to an $O(kn/w)$ expected-time algorithm for the case where m may be arbitrarily large. In practice this new algorithm, which computes a region of the d.p. matrix in $1 \times w$ blocks using the basic algorithm as a subroutine, is significantly faster than our previous 4-Russians algorithm, which computes the same region in 1×5 blocks using table lookup. This performance improvement yields a code which is superior to *all* existing algorithms except for some filtration algorithms when k/m is sufficiently small.

1 Introduction

The problem of finding substrings of a text similar to a given query string is a central problem in information retrieval and computational biology, to name but a few applications. It has been intensively studied over the last twenty years. In its most common incarnation, the problem is to find substrings that match the query with k or fewer differences. The first algorithm addressing exactly this problem is attributable to Sellers [Sel80] although one might claim that it was effectively solved by work in the early 70's on string comparison. Sellers algorithm requires $O(mn)$ time where m is the length of the query and n is the length of the text. Subsequently this was refined to $O(kn)$ expected-time by Ukkonen [Ukk85], then to $O(kn)$ worst-case time, first with $O(n)$ space by Landau and Vishkin [LV88], and later with $O(m^2)$ space by Galil and Park [GP90].

[*] Dept. of Computer Science, University of Arizona Tucson, AZ 85721 (e-mail: gene@cs.arizona.edu). Partially supported by NLM grant LM-04960

Of these early algorithms, the $O(kn)$ expected-time algorithm was universally the best in practice. The algorithm achieves its efficiency by computing only the region or zone of the underlying dynamic programming matrix that has entries less than or equal to k. Further refining this basic design, Chang and Lampe [CL92] went on to devise a faster algorithm which is conjectured to run in $O(kn/\sqrt{\sigma})$ expected-time where σ is the size of the underlying alphabet. Next, Wu, Manber, and this author [WMM96] developed a practical realization of the 4-Russians approach [MP80] that when applied to Ukkonen's zone, gives an $O(kn/\log s)$ expected-time algorithm, given that $O(s)$ space can be dedicated to a universal lookup table. In practice, these two algorithms are always superior to Ukkonen's zone design, and each faster than the other in different regions of the (k, σ) input-parameter space.

At around the same time, another new thread of practice-oriented results exploited the hardware parallelism of bit-vector operations. Letting w be the number of bits in a machine word, this sequence of results began with an $O(n\lceil m/w\rceil)$ algorithm for the exact matching case by Baeza-Yates and Gonnet [BYG92], and culminated with an $O(kn\lceil m/w\rceil)$ algorithm for the k-differences problem by Wu and Manber [WM92]. These authors were interested specifically in text-retrieval applications where m is quite small, small enough that the expression between the ceiling braces is 1. Under such circumstances the algorithms run in $O(n)$ or $O(kn)$ time, respectively. More recently, Baeza-Yates and Navarro [BYN96] have realized an $O(n\lceil km/w\rceil)$ variation on the Wu/Manber algorithm, implying $O(n)$ performance when $mk = O(w)$.

The final recent thrust has been the development of *filter* algorithms that eliminate regions of the text that cannot match the query. The results here can broadly divided into on-line algorithms (e.g. [WM92, CL94]) and off-line algorithms (e.g. [Mye94]) that are permitted to preprocess a presumably static text before performing a number of queries over it. After filtering out all but a presumably small segment of the text, these methods then invoke one of the algorithms above to verify if a match is actually present in the portion that remains. The *filtration efficiency* (i.e. percentage of the text removed from consideration) of these methods increases as the *mismatch ratio* $e = k/m$ approaches 0, and at some point, dependent on σ and the algorithm, they provide the fastest results in practice. However, improvements in verification-capable algorithms are still very desirable, as such results improve the filter-based algorithms when there are a large number of matches, and also are needed for the many applications where e is such that filtration is ineffective.

In this paper, we present two verification-capable algorithms, inspired by the 4-Russians approach, but using bit-vector computation instead of table lookup. First, we develop an $O(n\lceil m/w\rceil)$ bit-vector algorithm for the approximate string matching problem. This is asymptotically superior to prior bit-vector results, and in practice will be shown to be superior to the other bit-vector algorithms for all but a few choices of m and k. In brief, the previous algorithms use bit-vectors to model and maintain the state set of a non-deterministic finite automaton with $(m+1)(k+1)$ states that (exactly) matches all strings that are k-differences or fewer from the query. Our method uses bit-vectors in a very different way, namely, to encode the list of m arithmetic differences between successive entries in a column of the dynamic programming matrix. Our second algorithm comes from the observation that our first result can be thought of as a subroutine for computing any $1 \times w$ block of a d.p. matrix in $O(1)$ time. We may thus embed it in the zone paradigm of the Ukkonen algorithm, exactly as we did with the 4-Russians technique [WMM96]. The result is an $O(kn/w)$ expected-time algorithm which we will show in practice outperforms both our previous work and that of Chang and Lampe [CL92] for *all* regions of the (k, σ) parameter space.

2 Preliminaries

Assume the query sequence is $P = p_1 p_2 \ldots p_m$, the text is $T = t_1 t_2 \ldots t_n$, and that we are given a positive threshold $k \geq 0$. Further let $\delta(A, B)$ be the unit cost edit distance between strings A and B. The approximate string matching problem is to find all positions j in T such that there is a suffix of $T[1..j]$ matching P with k-or-fewer differences, i.e., j such that $\min_g \delta(P, T[g..j]) \leq k$.

The classic approach to this problem [Sel80] is to compute an $(m+1) \times (n+1)$ *dynamic programming (d.p.) matrix* $C[0..m, 0..n]$ for which $C[i,j] = \min_g \delta(P[1..i], T[g..j])$ after an $O(mn)$ time computation using the well-known recurrence:

$$C[i,j] \quad = \quad \min\{C[i-1,j-1] + (\text{if } p_i = t_j \text{ then } 0 \text{ else } 1), C[i-1,j]+1, C[i,j-1]+1\} \qquad (1)$$

subject to the boundary condition that $C[0,j] = 0$ for all j. It then follows that the solution to the approximate string matching problem is all locations j such that $C[m,j] \leq k$.

A basic observation is that the computation above can be done in only $O(m)$ space because computing column $C_j = < C[i,j] >_{i=0}^{m}$ only requires knowing the values of the previous column C_{j-1}. This leads to the important conceptual realization that one may think of a column C_j as a state of an automaton, and the algorithm as advancing from state C_{j-1} to state C_j as its "scans" symbol t_j of the text. The automaton is started in the state $C_0 =< 0, 1, 2, \ldots, m >$ and any state whose last entry is k-or-fewer is considered to be a final state.

The automaton just introduced has at most 3^m states. This follows because the d.p. matrix C has the property that the difference between adjacent entries in any row or any column is either 1, 0, or -1. Formally, define the *horizontal delta* $\Delta h[i,j]$ at (i,j) as $C[i,j] - C[i,j-1]$ and the *vertical delta* $\Delta v[i,j]$ as $C[i,j] - C[i-1,j]$ for all $(i,j) \in [1,m] \times [1,n]$.

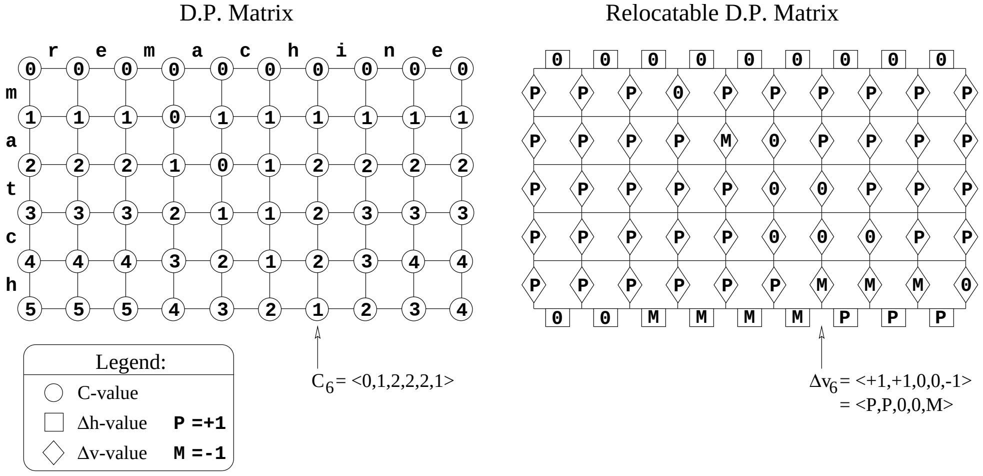
Lemma 1 [MP80, Ukk85]: For all i,j: $\Delta v[i,j], \Delta h[i,j] \in \{-1, 0, 1\}$.

It follows that to know a particular state C_j it suffices to know the *relocatable* column $\Delta v_j = < \Delta v[i,j] >_{i=1}^{m}$ because $C[0,j] = 0$ for all j.

We can thus replace the problem of computing C with the problem of computing the *relocatable d.p. matrix* Δv. One potential difficulty is that determining if Δv_j is final requires $O(m)$ time if one computes the sum $\Sigma_i \Delta v_j[i] = C[m,j]$ explicitly in order to do so. Our algorithm will compute a *block* of vertical deltas in $O(1)$ time, and thus cannot afford to compute this sum. Fortunately, one can simultaneously maintain the value of $Score_j = C[m,j]$ as one computes the $\Delta v_j' s$ using the fact that $Score_0 = m$ and $Score_j = Score_{j-1} + \Delta h[m,j]$. Note that the *horizontal* delta in the last row of the matrix is required, but the horizontal delta at the end of a block of vertical delta's will be seen to be a natural by-product of the block's computation. Figure 1 illustrates the basic dynamic programming matrix and its formulation in relocatable terms.

3 The Basic Algorithm

Representation. We seek to compute successive $\Delta v_j' s$ in $O(1)$ time using bit-vector operations under the assumption that $m \leq w$. We begin by choosing to represent the column Δv_j with two bit-vectors Pv_j and Mv_j, whose bits are set according to whether the corresponding delta in Δv_j is $+1$ or -1, respectively. Formally, $Pv_j(i) \equiv (\Delta v[i,j] = +1)$ and $Mv_j(i) \equiv (\Delta v[i,j] = -1)$, where the notation $W(i)$ denotes the i^{th} bit of the integer W. Note that $\Delta v[i,j] = 0$ exactly when *not* $(Pv_j(i)$ *or* $Mv_j(i))$ is true.



D.P. Matrix

		r	e	m	a	c	h	i	n	e
	0	0	0	0	0	0	0	0	0	0
m	1	1	1	0	1	1	1	1	1	1
a	2	2	2	1	0	1	2	2	2	2
t	3	3	3	2	1	1	2	3	3	3
c	4	4	4	3	2	1	2	3	4	4
h	5	5	5	4	3	2	1	2	3	4

Relocatable D.P. Matrix

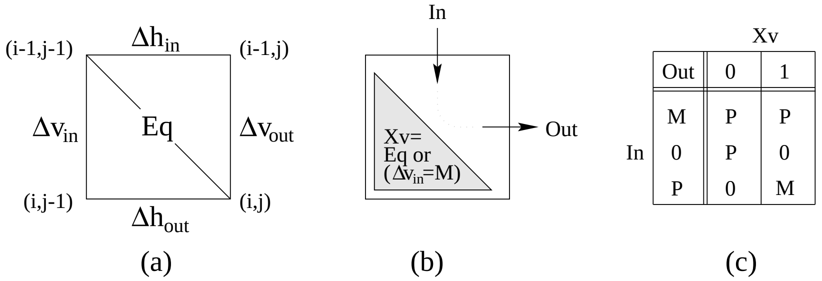
	0	0	0	0	0	0	0	0	0
P	P	P	0	P	P	P	P	P	P
P	P	P	P	M	0	P	P	P	P
P	P	P	P	P	0	0	P	P	P
P	P	P	P	P	0	0	0	P	P
P	P	P	P	P	P	M	M	M	0
	0	0	M	M	M	M	P	P	P

Figure 1: Dynamic Programming (D.P.) Matrices for P = *match* and T = *remachine*.

Cell Structure. Consider an individual *cell* of the d.p. matrix consisting of the square $(i-1, j-1)$, $(i-1, j)$, $(i, j-1)$, and (i, j). There are two vertical deltas, $\Delta v_{out} = \Delta v[i, j]$ and $\Delta h_{in} = \Delta v[i, j-1]$, and two horizontal deltas, $\Delta h_{out} = \Delta h[i, j]$ and $\Delta h_{in} = \Delta h[i-1, j]$, associated with the sides of this cell as shown in Figure 2(a). Further define $Eq = Eq[i, j]$ to be 1 if $p_i = t_j$ and 0 otherwise. Using the definition of the deltas and the basic recurrence for C-values it follows that:

$$\Delta v_{out} = \min\{-Eq, \Delta v_{in}, \Delta h_{in}\} + (1 - \Delta h_{in}) \quad (2a)$$
$$\Delta h_{out} = \min\{-Eq, \Delta v_{in}, \Delta h_{in}\} + (1 - \Delta v_{in}) \quad (2b)$$

It is thus the case that one may view Δv_{in}, Δh_{in}, and Eq as *inputs* to the cell at (i, j), and Δv_{out} and Δh_{out} as its *outputs*.



Out	0	1
M	P	P
In 0	P	0
P	0	M

Figure 2: D.P. Cell Structure and Input/Output Function.

Cell Logic. Observe that there are 3 choices for each of Δv_{in} and Δh_{in} and 2 possible values for Eq. Thus there are only 18 possible inputs for a given cell. The crucial idea that lead to this paper was the simple observation that in such a case one must be able to devise logical circuits or formulas capturing the functional dependence of the two outputs on the three inputs, and that these formulas apply universally to all cells.

As Figure 2(b) suggests, we find it conceptually easiest to think of Δv_{out} as a function of Δh_{in} modulated by an auxiliary boolean value $Xv \equiv (Eq \text{ } or \text{ } (\Delta v_{in} = -1))$ capturing the net effect of both Δv_{in} and Eq on Δv_{out}. With a brute force enumeration of the 18 possible inputs, one may verify the correctness of the table in Figure 2(c) which describes Δv_{out} as a function of Δh_{in} and Xv. In the table, the value -1 is denoted by M and $+1$ by P, in order to emphasize the logical, as opposed to the numerical, relationship between the input and output. Let Px_{io} and Mx_{io} be the bit values encoding Δx_{io}, i.e. $Px_{io} \equiv (\Delta x_{io} = +1)$ and $Mx_{io} \equiv (\Delta x_{io} = -1)$. By definition $Xv = Eq \textit{ or } Mv_{in}$ and from the the table one can verify that:

$$(Pv_{out}, Mv_{out}) = (Mh_{in} \textit{ or not } (Xv \textit{ or } Ph_{in}), Ph_{in} \textit{ and } Xv) \tag{3a}$$

By symmetry, given $Xh = (Eq \textit{ or } Mh_{in})$, it follows that:

$$(Ph_{out}, Mh_{out}) = (Mv_{in} \textit{ or not } (Xh \textit{ or } Pv_{in}), Pv_{in} \textit{ and } Xh) \tag{3b}$$

Alphabet Preprocessing. To evaluate cells according to the treatment above, one needs the boolean value $Eq[i,j]$ for each cell (i,j). In terms of bit-vectors, we will need an integer Eq_j for which $Eq_j(i) \equiv (p_i = t_j)$. Computing these integers during the scan would require $O(m)$ time and defeat our goal. Fortunately, in a preprocessing step, performed before the scan begins, we can compute a table of the vectors that result for each possible text character. Formally, if Σ is the alphabet over which P and T originate, then we build an array $Peq[\Sigma]$ for which: $Peq[s](i) \equiv (p_i = s)$. Constructing the table can easily be done in $O(|\Sigma|m)$ time and it occupies $O(|\Sigma|)$ space (continuing with the assumption that $m \leq w$). We are assuming, or course, that Σ is finite. At a small loss in efficiency our algorithm can be made to operate over infinite alphabets.

The Scanning Step. The central inductive step is to compute $Score_j$ and the bit-vector pair (Pv_j, Mv_j) encodingΔv_j, given the same information at column $j-1$ and the symbol t_j. In keeping with the automata conception, we refer to this step as *scanning* t_j and illustrate it in Figure 3 at the left. The basis of the induction is easy as we know at the start of the scan that $Pv_0(i) = 1$, $Mv_0(i) = 0$, and $Score_0 = m$. A scanning step is accomplished in two stages as illustrated in Figure 3:

1. First, the vertical delta's in column $j-1$ are used to compute the horizontal delta's at the bottom of their respective cells, using formula (3b).

2. Then, these horizontal delta's are used in the cell *below* to compute the vertical deltas in column j, using formula (3a).

In between the two stages, the *Score* in the last row is updated using the last horizontal delta now available from the first stage, and then the horizontal deltas are all *shifted* by one, pushing out the last horizontal delta and introducing a 0-delta for the first row. We like to think of each stage as a pivot, where the pivot of the first stage is at the lower left of each cell, and the pivot of the second stage is at the upper right. The delta's swing in the arc depicted and produce results modulated by the relevant X values.

The logical formulas (3) for a cell and the schematic of Figure 3, lead directly to the formulas below for accomplishing a scanning step. Note that the horizontal deltas of the first stage are recorded in a pair of bit-vectors, (Ph_j, Mh_j), that encodes horizontal deltas exactly as (Pv_j, Mv_j) encodes vertical deltas, i.e., $Ph_j(i) \equiv (\Delta h[i,j] = +1)$ and $Mh_j(i) \equiv (\Delta h[i,j] = -1)$.

$$\begin{array}{rcl} Ph_j(i) & = & Mv_{j-1}(i) \textit{ or not } (Xh_j(i) \textit{ or } Pv_{j-1}(i)) \\ Mh_j(i) & = & Pv_{j-1}(i) \textit{ and } Xh_j(i) \end{array} \tag{Stage 1}$$

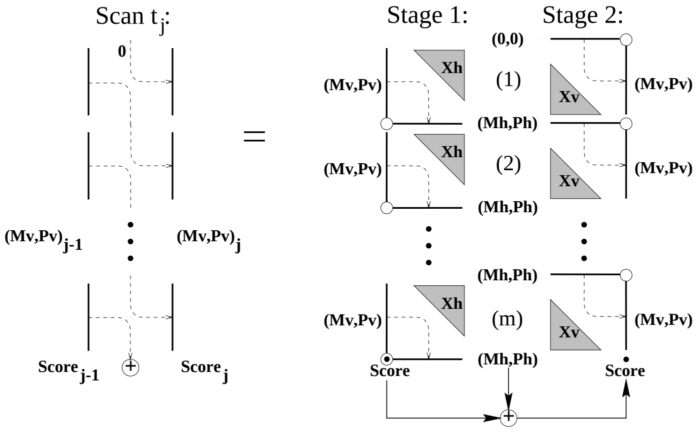


Figure 3: The Two Stages of a Scanning Step.

$$Score_j = Score_{j-1} + (1 \text{ if } Ph_j(m)) - (1 \text{ if } Mh_j(m)) \tag{4}$$

$$\begin{aligned} Ph_j(0) &= Mh_j(0) = 0^1 \\ Pv_j(i) &= Mh_j(i-1) \textit{ or not } (Xv_j(i) \textit{ or } Ph_j(i-1)) \\ Mv_j(i) &= Ph_j(i-1) \textit{ and } Xv_j(i) \end{aligned} \qquad \text{(Stage 2)}$$

At this point, it is important to understand that the formulas above specify the computation of bits in bit-vectors, all of whose bits can be computed in parallel with the appropriate machine operations. For example in *C*, we can express the computation of *all* of Ph_j as '`Ph = Mv | ~ (Xh | Pv)`'.

The X-Factors. The induction above is incomplete as we have yet to show how to compute Xv_j and Xh_j. By definition $Xv_j(i) = Peq[t_j](i) \textit{ or } Mv_{j-1}(i)$ and $Xh_j(i) = Peq[t_j](i) \textit{ or } Mh_j(i-1)$ where *Peq* is the precomputed table supplying *Eq* bits. The bitvector Xv_j can be directly computed at the start of the scan step as the vector Mv_{j-1} is input to the step. On the other hand, computing Xh_j requires the value of Mh_j which in turn requires the value of Xh_j! We thus have a cyclic dependency which must be unwound. Lemma 2 gives such a formulation of Xh_j which depends only on the values of Pv_{j-1} and $Peq[t_j]$.

Lemma 2: $Xh_j(i) = \exists k \leq i, Peq[t_j](k) \textit{ and } \forall x \in [k, i-1], Pv_{j-1}(x)^2$. (5)

Basically, Lemma 2 says that the i^{th} bit of *Xh* is set whenever there is a preceding *Eq* bit, say the k^{th} and a run of set *Pv* bits covering the interval $[k, i-1]$. In other words, one might think of the *Eq* bit as being "propagated" along a run of set *Pv* bits, setting positions in the *Xh* vector as it does so. This brings to mind the addition of integers, where carry propagation has a similar effect on the underlying bit encodings. Figure 4 illustrates the way we use addition to have the desired effects on bits which we summarize as Lemma 3 below, and which we prove precisely in the full paper.

Lemma 3: If $X = (((E\&P) + P)\hat{\ }P)|E$ then $X(i) = \exists k \leq i, E(k) \textit{ and } \forall x \in [k, i-1], P(x)$.

[1] In the more general case where the horizontal delta in the first row can be -1 or $+1$ as well as 0, these two bits must be set accordingly.

[2] In the more general case where the horizontal delta in the first row can be -1 or $+1$ as well as 0, $Peq[t_j](1)$ must be replaced with $Peq[t_j](1)$ *or* $Mh_j(0)$.

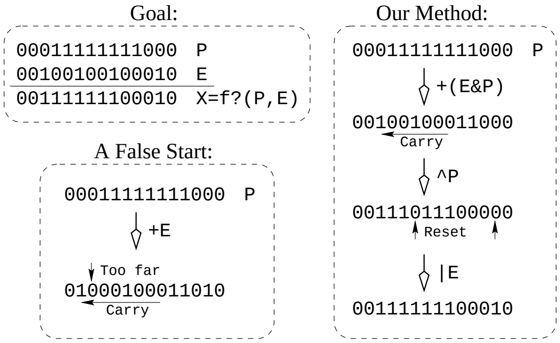


Figure 4: Illustration of Xv computation.

The Complete Algorithm. It now remains just to put all the pieces together. Figure 5 gives a complete specification in the style of a C program to give one a feel for the simplicity and efficiency of the result.

```
Precompute Peq[$\Sigma$]
Pv = $1^m$
Mv = 0
Score = $m$
for j = 1, 2, ...$n$ do
{ Eq = Peq[$t_j$]
  Xv = Eq | M
  Xh = (((Eq & Pv) + Pv) ^ Pv) | Eq

  Ph = Mv | ~ (Xh | Pv)
  Mh = Pv & Xh

  if Ph & $10^{m-1}$ then
     Score += 1
  else if Mh & $10^{m-1}$ then
     Score -= 1

  Ph <<= 1
  Pv = (Mh << 1) | ~ (Xv | Ph)
  Mv = Ph & Xv

  if Score $\leq k$ then
     print "Match at " · j
}
```

Figure 5: The Basic Algorithm.

The complexity of the algorithm is easily seen to be $O(m\sigma + n)$ where σ is the size of the alphabet Σ. Indeed only 17 bit operations are performed per character scanned. This is to be contrasted with the Wu/Manber bit-vector algorithm [WM92] which takes $O(m\sigma + kn)$ under the prevailing assumption that $m \leq w$. The Baeza-Yates/Navarro bit-vector algorithm [BYN96] has this same complexity under this assumption, but improves to $O(m\sigma + n)$ time when one assumes $m \leq 2\sqrt{w} - 2$ (e.g., $m \leq 9$ when $w = 32$ and $m \leq 14$ when $w = 64$).

Finally, consider the case where m is unrestricted. Such a situation can easily be accommodated by simply modeling an m-bit bit-vector with $\lceil m/w \rceil$ words. An operation on such bit-vectors takes $O(m/w)$ time. It then directly follows that the basic algorithm of this section runs in $O(m\sigma + nm/w)$ time and $O(\sigma m/w)$ space. This is to be contrasted with the previous bit-vector algorithms

[WM92, BYN96], *both* of which take $O(m\sigma + knm/w)$ time asymptotically. This leads us to say that our algorithm represents a true asymptotic improvement over previous *bit-vector* algorithms.

4 The Unrestricted Algorithm.

The Blocks Model. Just as we think of the computation of a single cell as realizing an input/output relationship on the four deltas at its borders, we may more generally think of the computation of a $u \times v$ rectangular subarray or *block* of cells as resulting in the output of deltas along its lower and right boundary, given deltas along its upper and left boundary as input. This is the basic observation behind Four Russians approaches to sequence comparison [MP80, WMM96], where the output resulting from every possible input combination is pretabulated and then used to effect the computation of blocks as they are encountered in a particular problem instance. We can similarly modify our basic algorithm to effect the $O(1)$ computation of $1 \times w$ blocks (via bitvector computation as opposed to table lookup), given that we are careful to observe that in this context the horizontal input delta may be -1 or $+1$, as well as 0.

There are several sequence comparison results that involve computing a region or *zone* of the underlying dynamic programming matrix, the first of which was [Ukk85]. Figure 6 depicts such a hypothetical zone and a tiling of it with $1 \times w$ blocks. Provided that one can still effectively delimit the zone while performing a block-based computation, using such a tiling gives a factor w speedup over the underlying zone algorithm. For example, we take Ukkonen's $O(kn)$ expected-time algorithm and improve it to $O(kn/w)$ below. Note that blocks are restricted to one of at most $b_{max} = \lceil m/w \rceil$ levels, so that only $O(\sigma m/w)$ *Eq*-vectors need be precomputed. Further note that any internal boundary of the tiling has a delta of 1.

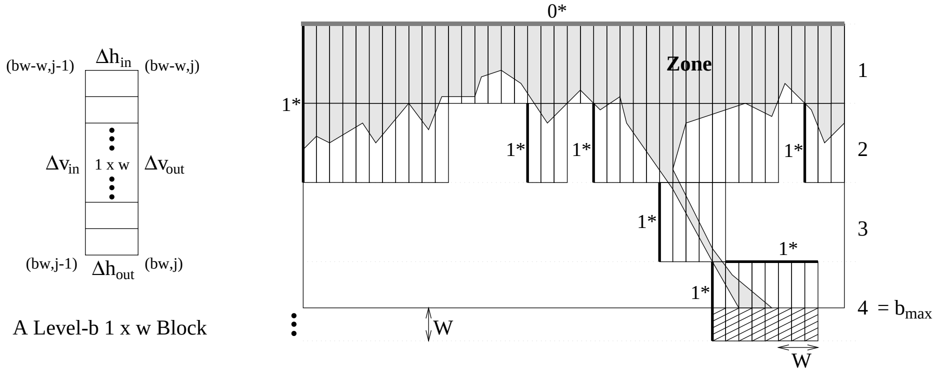


Figure 6: Block-Based Dynamic Programming.

A Block-Based Algorithm for Approximate String Matching. Ukkonen improved the expected time of the standard $O(mn)$ d.p. algorithm for approximate string matching, by computing only the zone of the d.p. matrix consisting of the prefix of each column ending with the last k in the column. That is, if $x_j = \max\{i : C(i,j) \leq k\}$ then the algorithm takes time proportional to the size of the zone $Z(k) = \cup_{j=0}^{n}\{(i,j) : i \in [0, x_j]\}$. It was shown in [CL92] that the expected size of $Z(k)$ is $O(kn)$. Computing just the zone is easily accomplished with the observation that $x_j = \max\{i : i \leq x_{j-1} + 1 \text{ and } C(i,j) \leq k\}$.

A block-based algorithm for this $O(kn)$ expected-time algorithm was devised and presented in an earlier paper of ours [WMM96] where the blocks were computed in $O(1)$ time using a 4-Russians lookup table. What we are proposing here, is to do exactly the same thing, except to use our bit-vector approach to compute $1 \times w$ blocks in $O(1)$ time. As we will see in the next section, this results in almost a factor of 4-5 improvement in performance, as the 4-Russians table lookups were limited to 1×5 blocks and the large tables involved result in much poorer cache coherence, compared to the bit-vector approach where all the storage required typically fits in the on-board CPU cache. We describe the small modifications necessary to tile $Z(k)$ in the full paper.

5 Some Empirical Results

We report on two sets of comparisons run on a Dec Alpha 4/233 for which $w = 64$. The first is a study of our basic bit-vector algorithm and the two previous bit-vector results [WM92, BYN96] for approximate string matching when $m \leq w$. The second set of experiments involves all verification-capable algorithms that work when k and m are unrestricted. Experiments to determine the range of k/m for which filter algorithms are superior have not been performed in this preliminary study.

The expected time complexity of each algorithm, A, is of the form $\Theta(f_A(m,k,\sigma)n)$ and our experiments are aimed at empirically measuring f_A. In the full paper we will detail the specific trials, noting here only that (a) their design guarantees a measurement error of at most 2.5%, and (b) the search texts were obtained by randomly selecting characters from an alphabet of size σ with equal probability.

Our first set of experiments compare the three bit-vector algorithms for the case where $m \leq 64$, and the results are shown in Figure 7. At left we show our best estimate for f_A for each algorithm. For our basic algorithm, f_A is a constant as is also true of the Baeza-Yates and Navarro algorithm save that it can only be applied to the region of the parameter space where $(m-k)(k+2) \leq 64$. Their algorithm can be extended to treat a greater range of k and m by linking automata together, but since such an extension will run at least twice as slowly as the case measured, it clearly will not be competitive with our basic algorithm in the remainder of the region. The Wu and Manber algorithm performs linearly in k and a least-squares regression line fits the results of 90 trials very well, save that the fit is off by roughly 9% for the first two values of k (see Figure 7). We hypothesize that this is due to the effect of branch-prediction in the instruction pipeline hardware. Figure 7 depicts the values of k and m for which each method is superior to the others. In the zone where the Baeza-Yates and Navarro algorithm requires no automata linking it is 12% faster than our basic algorithm, and for $k = 0$ the algorithm of Manber and Wu is 29% faster. For the remaining 1832 out of 2080 (88%) choices of m and k, our basic algorithm gives the best performance.

Our second set of experiments are aimed at comparing verification-capable algorithms that can accommodate unrestricted choices of k and m. In this case, we need only consider our block-based algorithm and the results of [CL92] and [WMM96], as all other competitors are already known to be dominated in practice by these two [WMM96]. These three algorithms are all zone-based and when m is suitably large, the zone never reaches the last row of the d.p. matrix, so that running time does not depend on m. We set $m = 400$ for all trials and ran 107 trials with $(k,\sigma) \in \{0,1,2,\ldots 6,8,10,\ldots 20,24,28\ldots 60\} \times \{2,4,8,16,32\} \cup \{64,68,72,\ldots 120\} \times \{32\}$. For each of the five choices of σ, Figure 8 has a graph of time as a function of k, one curve for each algorithm. From this figure it is immediately clear that our block-based algorithm is superior to the others for all choices of k and σ we tried. The Change and Lampe algorithm may eventually overtake ours but it did not do so with $\sigma = 95$, the number of printable ASCII characters.

Basic: .204n

Baeza-Yates/Navarro: .181n
(when (m-k)(k+2) ≤ 64)

Wu/Manber: (.159+.079k)n
k=0: .146n
k=1: .225n
k=2: .316n
k=3: .404n

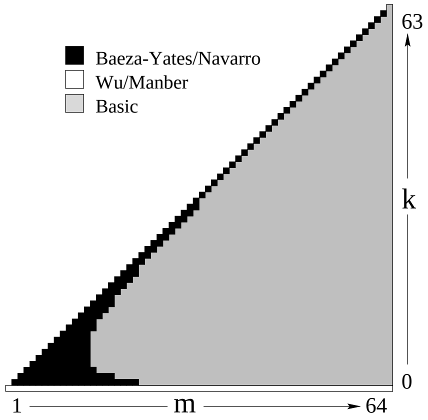


Figure 7: Performance Summary and Regions of Superiority for Bit-Vector Algorithms.

References

[BYG92] R.A. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35:74–82, 1992.

[BYN96] R.A. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. 7th Symp. on Combinatorial Pattern Matching. Springer LNCS 1075*, pages 1–23, 1996.

[CL92] W.I. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. 3rd Symp. on Combinatorial Pattern Matching. Springer LNCS 644*, pages 172–181, 1992.

[CL94] W.I. Chang and E.L. Lawler. Sublinear expected time approximate matching and biological applications. *Algorithmica*, 12:327–344, 1994.

[GP90] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19:989–999, 1990.

[LV88] G.M. Landau and U. Vishkin. Fast string matching with k differences. *J. of Computer and System Sciences*, 37:63–78, 1988.

[MP80] W.J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *J. of Computer and System Sciences*, 20:18–31, 1980.

[Mye94] E.W. Myers. A sublinear algorithm for approximate keywords searching. *Algorithmica*, 12:345–374, 1994.

[Sel80] P.H. Sellers. The theory and computations of evolutionary distances: Pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35:83–91, 1992.

[WMM96] S. Wu, U. Manber, and G. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15:50–67, 1996.

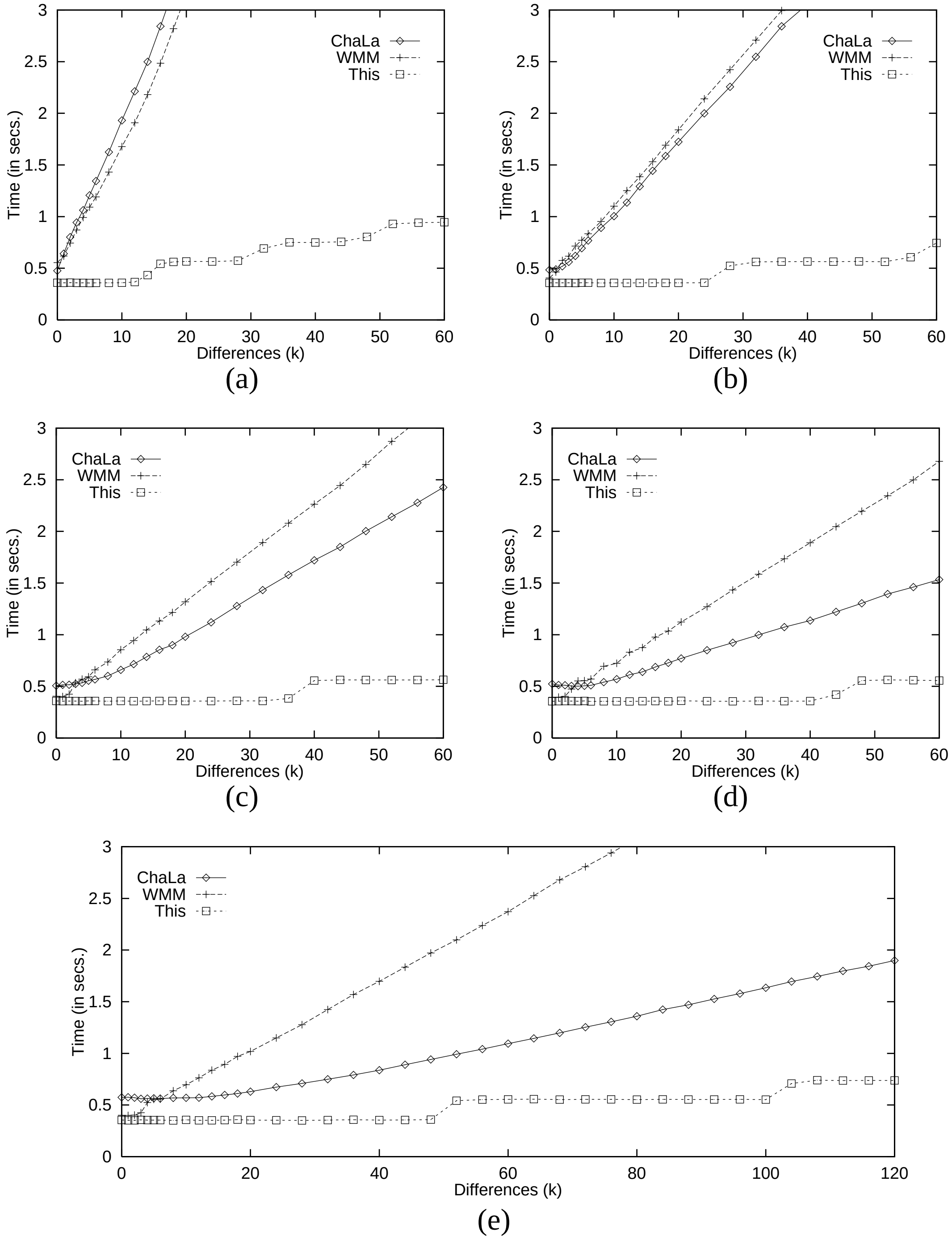


Figure 8: Timing curves for the $O(kn/w)$ block-based algorithm ("This") versus Chang/Lampe ("ChaLa") and Wu/Manber/Myers ("WMM"), with alphabet sizes (a) $\sigma = 2$, (b) $\sigma = 4$, (c) $\sigma = 8$, (d) $\sigma = 16$, and (e) $\sigma = 32$.