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

GENE MYERS

University of Arizona, Tucson, Arizona

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 either $O(nmk/w)$ or $O(nm \log \sigma/w)$ time where w is the word size of the machine (e.g., 32 or 64 in practice), and σ is the size of the pattern alphabet. 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 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 et al. [1996]. 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, that computes a region of the dynamic programming (d.p.) matrix w entries at a time using the basic algorithm as a subroutine, is significantly faster than our previous 4-Russians algorithm, that computes the same region 4 or 5 entries at a time using table lookup. This performance improvement yields a code that is either superior or competitive with *all* existing algorithms except for some filtration algorithms that are superior when k/m is sufficiently small.

Categories and Subject Descriptors: G.4 [**Mathematics of Computing**]: *Mathematical Software*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Algorithms, Designs

Additional Key Words and Phrases: Approximate string search, bit-parallelism, sequence comparison

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

This research was partially supported by NLM grant LM-04960.

Author's present address: Celera Genomics Corporation, 45 West Gude Drive, Rockville, MD 20850.

problem is attributable to Sellers [1980] although one might claim that it was effectively solved by earlier work on string comparison (e.g., Wagner and Fischer [1974]). 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 [1985], then to $O(kn)$ worst-case time, first with $O(n)$ space by Landau and Vishkin [1988], and later with $O(m^2)$ space by Galil and Park [1990].

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 [1992] 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 from which the strings are formed. Next, Wu et al. [1996] developed a particularly practical realization of the 4-Russians approach [Masek and Paterson 1980] that when applied to Ukkonen's zone, gives an algorithm that runs in $O(kn/\log s)$ expected time, given that $O(s)$ space can be dedicated to a universal lookup table. In practice, these two algorithms were 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 and an $O(n\lceil m \log k/w \rceil)$ algorithm for the k-mismatches problem by Baeza-Yates and Gonnet [1992], followed by an $O(nk\lceil m/w \rceil)$ algorithm for the k-differences problem by Wu and Manber [1992]. These authors were interested specifically in text-retrieval applications where m is quite small, small enough that the expressions between the ceiling braces is 1. Under such circumstances, the algorithms run in $O(n)$ or $O(kn)$ time, respectively. Two years later, Wright [1994] presented an $O(n \log_2 \sigma \lceil m/w \rceil)$ bit-vector style algorithm where σ is the size of the alphabet for the pattern. Most recently, Baeza-Yates and Navarro [1996] 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 and off-line algorithms[1] 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 decreases as the *mismatch ratio* $e = k/m$ is increased, and at some point, dependent on σ and the algorithm, they fail to eliminate enough of the text to be worthwhile. For sufficiently small e, the filter/verify paradigm gives the fastest results in practice.

[1]See, for example, Wu and Manber [1992], Ukkonen [1992], Chang and Lawler [1994], Pevener and Waterman [1995], and Sutinen and Tarhio [1996] for on-line algorithms and Ukkonen [1993], Myers [1994], and Cobbs [1995] for off-line algorithms.

However, improvements in verification-capable algorithms are still 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 many choices of m and k. In brief, the previous algorithms, except for that by Wright, use bit-vectors to model and maintain the state set of a nondeterministic 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 different way, namely, to encode the list of m (arithmetic) differences between successive entries in a column of the dynamic programming matrix. Wright's algorithm also takes this angle of attack, but proceeds arithmetically instead of logically, resulting in a less efficient encoding (three bits per entry versus one for our method) and further implying an additional factor of $\log_2\sigma$ in time. Our second algorithm comes from the observation that our first result can be thought of as a subroutine for simultaneously computing w entries 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. The result is an $O(kn/w)$ expected-time algorithm which we will show in practice outperforms both our previous work [Wu et al. 1996] and that of Chang and Lampe [1992] for *all* regions of the (k, σ) parameter space. It further outperforms a refinement of the algorithm of Baeza-Yates and Navarro [1996] except for a few values of k near 0, where it is slower by only a small percentage.

2. *Preliminaries*

We will assume that the query sequence is $P = p_1p_2 \cdots p_m$, that the text is $T = t_1t_2 \cdots t_n$, 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. Formally, the approximate string matching problem is to find all positions j in T such that there is a suffix of $T[1 \cdots j]$ matching P with k-or-fewer differences, that is, j such that $\min_g \delta(P, T[g \cdots j]) \leq k$.

The classic approach to this problem [Sellers 1980] is to compute an $(m + 1) \times (n + 1)$ *dynamic programming* (*d.p.*) *matrix* $C[0 \cdots m, 0 \cdots n]$ for which it will be true that $C[i, j] = \min_g \delta(P[1 \cdots i], T[g \cdots j])$ at the end of the computation. This can be done in $O(mn)$ time using the well-known recurrence:

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

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

Another basic observation is that the computation above can be done in only $O(m)$ space because computing column $C_j = \langle C[i, j] \rangle_{i=0}^{m}$ only requires knowing

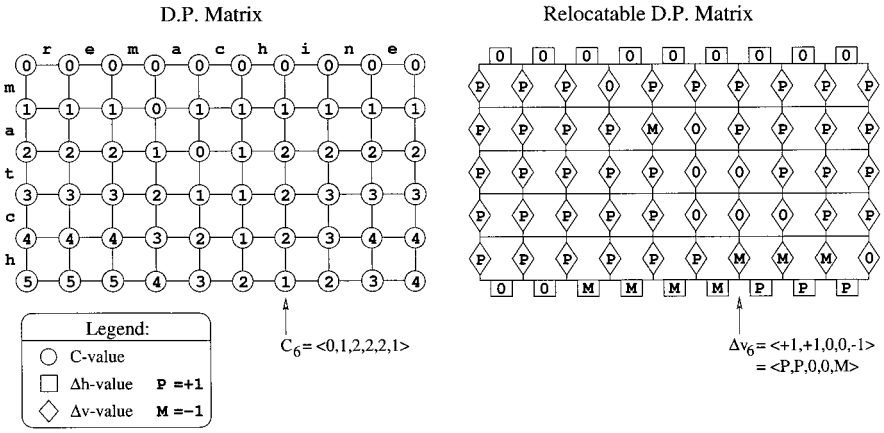

FIG. 1. Dynamic programming (d.p.) matrices for P = *match* and T = *remachine*.

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 it "scans" symbol t_j of the text. The automaton is started in the state $C_0 = \langle 0, 1, 2, \ldots, m \rangle$ and any state whose last entry is k-or-fewer is considered to be a final state.

Ukkonen [1986] showed that the automaton just introduced has a finite number of states, at most 3^m, in fact. This follows from the observation that 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. Interestingly, a more general version of the lemma below was first proven by Masek and Paterson [1980] in the context of the first 4-Russians algorithm for string comparison. 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 $(i, j) \in [1, m] \times [1, n]$. We have:

LEMMA 1. [MASEK AND PATERSON 1980; UKKONEN 1985]. 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. One now immediately sees that the automaton can have at most 3^m states as there are only 3 choices for each vertical delta.

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 as one must determine whether $\Sigma_i \Delta v_j[i] = C[m, j] \leq k$. While this does not effect the asymptotics of most algorithmic variations on the basic d.p. formulation, it is crucial to algorithms such as the one in this paper that compute a *block* of vertical deltas in $O(1)$ time, and thus cannot afford to compute the sum over these deltas without affecting both their asymptotic and practical efficiency. 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 as we will see later, the horizontal delta at the end of a block of vertical delta's is a natural by-product of the block's computation. Figure 1 illustrates the basic dynamic

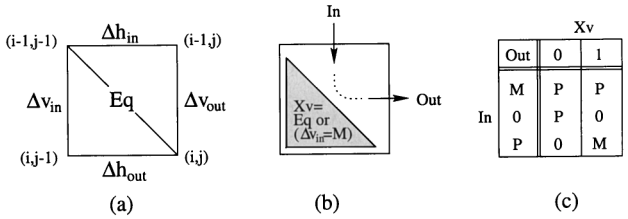

FIG. 2. D.P. cell structure and input/output function.

programming matrix and its formulation in relocatable terms.

3. *The Basic Algorithm*

We seek to compute successive $\Delta v_j's$ in $O(1)$ time using bit-vector operations. We assume, for the entirety of this section, that the size of a machine word is w and that $m \leq w$. We further assume that parallel bit operations, such as *or*, *and*, and *not*, and simple arithmetic operations, such as addition and subtractions, take the underlying RAM architecture constant time to perform on such words. On most current machines, w is typically 32 or 64.

3.1. REPRESENTATION. The first task is to choose a bit-vector representation for Δv_j. We do so 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,

$$Pv_j(i) \equiv (\Delta v[i, j] = +1)$$
$$Mv_j(i) \equiv (\Delta v[i, j] = -1) \tag{2}$$

where the notation $W(i)$ denotes the ith bit of the integer or word W, and where i is assumed to be in the range $[1, w]$. Note that the ith bits of the two vectors cannot be simultaneously set, and that we do not need a vector to encode the positions i that are zero, as we know they occur when *not* $(Pv_j(i)$ *or* $Mv_j(i))$ is true.

3.2. CELL STRUCTURE. The next task is to develop an understanding of how to compute the deltas in one column from those in the previous column. To start, 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 horizontal and two vertical deltas $-$ $\Delta v[i, j]$, $\Delta v[i, j - 1]$, $\Delta h[i, j]$, and $\Delta h[i - 1, j]$-associated with the sides of this cell as shown in Figure 2(a). Further, let $Eq[i, j]$ be a bit quantity which is 1 if $p_i = t_j$ and 0 otherwise. Using the definition of the deltas and the basic recurrence for C-values we arrive at the following equation for $\Delta v[i, j]$ in terms of $Eq[i, j]$, $\Delta v[i, j - 1]$, and $\Delta h[i - 1, j]$:

$$\begin{aligned}\Delta v[i, j] &= C[i, j] - C[i - 1, j] \\ &= \min\{C[i - 1, j - 1] + (\text{if } p_i = t_j \text{ then } 0 \text{ else } 1), \\ &\quad C[i - 1, j] + 1, C[i, j - 1] + 1\} - C[i - 1, j]\end{aligned}$$

$$= \min\begin{Bmatrix} C[i-1,j-1] + (1 - Eq[i,j]) \\ C[i-1,j-1] + \Delta v[i,j-1] + 1 \\ C[i-1,j-1] + \Delta h[i-1,j] + 1 \end{Bmatrix} - (C[i-1,j-1] + \Delta h[i-1,j])$$

$$= \min\{-Eq[i,j], \Delta v[i,j-1], \Delta h[i-1,j]\} + (1 - \Delta h[i-1,j]). \quad (3a)$$

Similarly:

$$\Delta h[i,j] = \min\{-Eq[i,j], \Delta v[i,j-1], \Delta h[i-1,j]\} + (1 - \Delta v[i,j-1]). \quad (3b)$$

It is thus the case that one may view $\Delta v_{in} = \Delta v[i, j-1]$, $\Delta h_{in} = \Delta h[i-1, j]$, and $Eq = Eq[i, j]$ as *inputs* to a cell, and $\Delta v_{out} = \Delta v[i, j]$ and $\Delta h_{out} = \Delta h[i, j]$ as its *outputs.*

3.3. CELL LOGIC. The next observation is that there are three choices for each of Δv_{in} and Δh_{in} and two possible values for Eq. Thus, there are only a finite number, 18, possible inputs for a given cell. This gave rise to the key idea that one could compute the numeric values in a column with Boolean logic, whereas all but one of the previous methods use a bit vector to implement a set over a finite number of elements. Wright [1994] also pursued the idea of computing the difference vectors but chose to think of them as (*mod* 4) numbers packed in a word with a padding bit separating each so that the numbers could be arithmetically operated upon in parallel. In contrast, we are viewing the computation purely in terms of Boolean logic. We experimented with different encodings and different formulations, but present here only our best design.

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 capturing the 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}, that is, $Px_{io} \equiv (\Delta x_{io} = +1)$ and $Mx_{io} \equiv (\Delta x_{io} = -1)$. From the table, one can verify the following logical formulas capturing the function:

$$\begin{aligned} Xv &= Eq \text{ or } Mv_{in} \\ Pv_{out} &= Mh_{in} \text{ or not } (Xv \text{ or } Ph_{in}) \\ Mv_{out} &= Ph_{in} \text{ and } Xv \end{aligned} \quad (4a)$$

Studying the relationship between Δh_{out} and Δv_{in} modulated by $Xh \equiv (Eq \text{ or } (\Delta h_{in} = -1))$, gives the following symmetric formulas for computing the bits of the encoding of Δh_{out}.

$$\begin{aligned} Xh &= Eq \text{ or } Mh_{in} \\ Ph_{out} &= Mv_{in} \text{ or not } (Xh \text{ or } Pv_{in}) \\ Mh_{out} &= Pv_{in} \text{ and } Xh \end{aligned} \quad (4b)$$

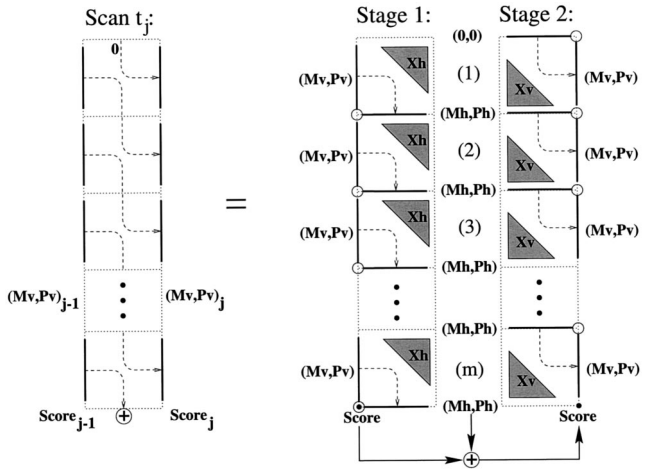

FIG. 3. The two stages of a scanning step.

3.4. 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:

$$Peq[s](i) \equiv (p_i = s). \tag{5}$$

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, as it invariably is in search applications over standard machine character sets (e.g., ASCII or the ISO standards). At a small loss in efficiency, our algorithm can be made to operate over infinite alphabets. We leave this as an exercise or refer the reader to Wu et al. [1996, page 57].

3.5. THE SCANNING STEP. The central inductive step is to compute $Score_j$ and the bit-vector pair (Pv_j, Mv_j) encoding Δv_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. The basis of the induction is easy as we know:

$$\begin{aligned} Pv_0(i) &= 1 \\ Mv_0(i) &= 0 \\ Score_0 &= m \end{aligned} \tag{6}$$

That is, at the start of the scan, the *Score* variable is m, the Mv bit-vector is all 0's, and the Pv bit-vector is all 1's.

The difficulty presented by the induction step is that given the vertical delta on its left side, the only applicable formulas, namely (4b), give the horizontal delta at the bottom of the cell, whereas the goal is to have the vertical delta on its right side. To achieve this requires 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 (4b).
(2) Then, these horizontal delta's are used in the cell *below* to compute the vertical deltas in column j, using formula (4a).

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. For the moment, we will assume Xh and Xv are known, deferring their computation til the next subsection.

The logical formulas (4) 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, that is, $Ph_j(i) \equiv (\Delta h[i,j] = +1)$ and $Mh_j(i) \equiv (\Delta h[i,j] = -1)$.

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

$$Score_j = Score_{j-1} + (1 \text{ if } Ph_j(m)) - (1 \text{ if } Mh_j(m)) \qquad (7)$$

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

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. In this paper, we use the C programming language to do so. In *C*, the operation | is bitwise-or, & is bitwise-and, ^ is bitwise-xor, ~ is prefix-unary bitwise-not, and ≪1 is suffix-unary shift-left-by-one. Thus, we can, for example, express the computation of *all* of Ph_j as `'Ph = Mv | ~ (Xh | Pv)'` and the computation of all of Mv_j as `'Mv = (Ph <<= 1) & Xv'`.

3.6. THE X-FACTORS. The induction above is incomplete, in that we did not show how to compute the bits of the bit-vectors Xv_j and Xh_j. We have immediately from their definition in (4) that:

$$\begin{aligned} Xv_j(i) &= Peq[t_j](i) \textit{ or } Mv_{j-1}(i) \\ Xh_j(i) &= Peq[t_j](i) \textit{ or } Mh_j(i-1), \end{aligned} \qquad (8)$$

where we are using the precomputed table *Peq* to lookup the necessary *Eq* bits. Computing Xv_j at the beginning of the scan step is not problematic, the vector Mv_{j-1} is input to the step. On the other hand, computing Xh_j requires the value

[2]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.

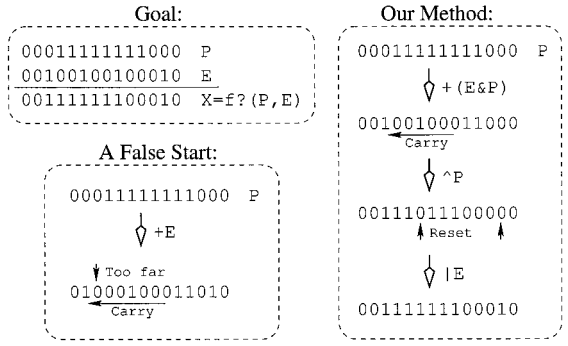

FIG. 4. Illustration of Xv computation.

of Mh_j which in turn requires the value of Xh_j! We thus have a cyclic dependency that 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. $Xh_j(i) = \exists k \leq i, Peq[t_j](k)$ *and* $\forall x \in [k, i-1], Pv_{j-1}(x)$.[3]

PROOF. Observe from formulas (4b) that for all k, $Mh_j(k)$ is true iff $Pv_{j-1}(k)$ and $Xh_j(k)$ are true. Combining this with Eq. (8), it follows that $Mh_j(k) \equiv ((Pv_{j-1}(k)$ *and* $Peq[t_j](k))$ *or* $((Pv_{j-1}(k)$ *and* $Mh_j(k-1))$. Repeatedly applying this we obtain the desired statement by induction:

$$
\begin{aligned}
Xh_j(i) &= Peq[t_j](i) \quad or \quad Mh_j(i-1) \\
&= Peq[t_j](i) \quad or \quad (Pv_{j-1}(i-1) \text{ and } Mh_j(i-2)) \\
&\qquad\quad or \quad (Pv_{j-1}(i-1) \text{ and } Mh_j(i-2)) \\
&= Peq[t_j](i) \quad or \quad Pv_{j-1}(i-1) \text{ and } Peq[t_j](i-1)) \\
&\qquad\quad or \quad (Pv_{j-1}(i-1) \text{ and } Pv_{j-1}(i-2) \text{ and } Peq[t_j](i-2)) \\
&\qquad\quad or \quad (Pv_{j-1}(i-1) \text{ and } Pv_{j-1}(i-2) \text{ and } Mh_j(i-3)) \\
&= \ldots \\
&= \exists k \leq i, Peq[t_j](k) \text{ and } \forall x \in [k, i-1], Pv_{j-1}(x) \qquad (\text{as } Mh_j(0) = 0).
\end{aligned}
$$

□

So the last remaining obstacle is to determine a way to compute the bit-vector Xh in a constant number of word-operations. Basically, Lemma 2 says that the ith bit of Xh is set whenever there is a preceding Eq bit, say the kth 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 idea. First, consider just the effect of adding P and E together, where P has the value of Pv_{j-1} and E that of $Peq[t_j]$. Each bit in E initiates a carry-propagation chain down a run of-set P-bits that turns these bits to 0's except where an E-bit is also set. In the figure, this possibility is labeled "A False Start" because we observe that the carry propagation can proceed beyond the

[3]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* $Mh_j(0)$.

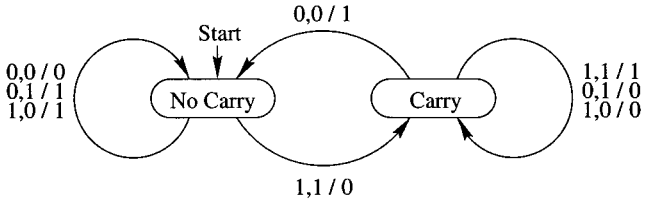

FIG. 5. The addition automation.

end of a run of set P-bits because of set E-bits. Therefore, one must first turn off all E-bits not covered by a run of set P-bits, that is, form $E\&P$, and then add this to P. One can then capture all the bits in P that got toggled during the carry propagation by taking the exclusive *or* of the result with P. Finally, one can *or* in the E-bits to capture those that were either not covered by a run of set P-bits, or that were not the initiators of a carry propagation chain. In summary we claim that:

$$Xh_j = (((Peq[t_j]\&Pv_{j-1}) + Pv_{j-1}) \hat{} Pv_{j-1}) | Peq[t_j] \tag{10}$$

and verify this formally in Lemma 3.

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

PROOF. Consider the transducer for addition shown in Figure 5 immediately above. A transition of the form $a, b/c$ is taken when the corresponding bits of the operands are a and b, and the bit c results. It follows that a 1 is output when in the *Carry*-state iff the bits of the operands are equal. The opposite is output if the transducer is in the *No Carry*-state. Furthermore, one is in the *Carry*-state when processing bit i iff there is a previous bit position k, for which the bits of both operands are set and where at least one of the operands bits is set in all positions between k and i. Putting this together leads to the following formal logical description of the effect of addition:

$$\begin{aligned}(Q + P)(i) &= (\exists k < i,\ Q(k)\ and\ P(k)\ and\ \forall x \in [k, i-1],\ (Q(x)\ or\ P(x))) \\ &\equiv (Q(i) \equiv P(i)).\end{aligned}$$

Replacing Q by $E\&P$ in this expression and then applying some simple logical inferences leads to the conclusion that, if $Y = (E\&P) + P$, then:

$$Y(i) = (\exists k < i,\ E(k)\ and\ \forall x \in [k, i-1],\ P(x)) \equiv (E(i)\ or\ not\ P(i)).$$

Next we use the inferences that $((A \equiv B)\ xor\ (P)$ iff $(A \equiv (B\ xor\ P))$ and that $((E\ or\ not\ (P)\ xor\ P)$ iff $not\ (E\ and\ P)$, to conclude that, if $Y = ((E\&P) + P) \hat{} P$, then:

$$Y(i) = (\exists k < i,\ E(k)\ and\ \forall x \in [k, i-1],\ P(x)) \equiv not\ (E(i))\ and\ P(i)).$$

The last step requires the inference $((A \equiv B)\ or\ E)\ and\ ((not\ B) \Rightarrow E)$ is equivalent to $(A\ or\ E)$. That is, if $Y = (((E\&P) + P) \hat{} P)|E$, then it follows

```
1.   Precompute Peq[σ]                                          (5)
2.   Pv = 1^m
3.   Mv = 0                                                     (6)
4.   Score = m

5.   for j = 1, 2, ...n do

6.    {  Eq = Peq[t_j]
7.       Xv = Eq | M                                           (8)
8.       Xh = (((Eq & Pv) + Pv) ^ Pv) | Eq                     (10)

9.       Ph = Mv | ~ (Xh | Pv)
10.      Mh = Pv & Xh                                          (4b)

11.      if Ph & 10^(m-1) then
12.         Score += 1
13.      else if Mh & 10^(m-1) then                            (7)
14.         Score -= 1

15.      Ph <<= 1
16.      Mh <<= 1
17.      Pv = Mh | ~ (Xv | Ph)                                 (4a)
18.      Mv = Ph & Xv

19.      if Score ≤ k then
20.         print "Match at " · j
      }
```

FIG. 6. The basic algorithm.

that:

$$Y(i) = (\exists k < i,\ E(k)\ \textit{and}\ \forall x \in [k, i-1],\ P(x))\ \textit{or}\ E(i),$$

which is just a slight restatement of the conclusion of the lemma. □

3.7. THE COMPLETE ALGORITHM. It now remains just to put all the pieces together. Figure 6 gives a complete specification in the style of a C program to give one a feel for the simplicity and efficiency of the result. At the right of each basic code block is the formula(s) justifying it.

The table `Peq` is built before the scan as specified in formula (5). Two bit-vectors, `Pv`, and `Mv`, and integer `Score` are maintained during the scan and at the completion of scanning the jth character contain the values of Pv_j, Mv_j, and $Score_j$, respectively. These variables are set according to formula (6) to correctly initiate the scan. To scan the symbol t_j, the algorithm uses five intermediate bit-vectors `Eq`, `Xv`, `Xh`, `Ph`, and `Mv` in the interior of the scan loop. First, `Xh` and `Xv` are computed to have the values of Xh_j and Xv_j according to formulas (8) and (10) using the variable `Eq` to factor the common subexpression $Peq[t_j]$. Then `Ph` and `Mh` are computed to hold the horizontal deltas for the jth column (formula (4b)), `Score` is updated to the value of $Score_j$ using formula (7), and `Pv` and `Mv` are updated to hold the vertical deltas in column j. Finally, the value of `Score` is checked to see if there is a match.

The correctness of the algorithm follows directly from the treatment leading to this point. Moreover, 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 [Wu and Manber 1992], which takes $O(m\sigma + kn)$

under the prevailing assumption that $m \leq w$. The Baeza-Yates/Navarro bit-vector algorithm [Baeza-Yates and Navarro 1996] 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 [Wu and Manber 1992; Baeza-Yates and Navarro 1996] *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. *Extensions*

4.1. Limited Regular Expressions. The first extension is that our bit-vector algorithm can accommodate limited regular expressions as introduced in Wu and Manber [1992] and Baeza-Yates and Gonnet [1992]. Conceptually, a limited regular expression is a sequence of sets of symbols, where a symbol of the text is considered to match position i of the pattern iff it is in the ith set. Typically, a subset of the *egrep*-syntax is used to specify the patterns (see Wu et al. [1996]), including the *wild-card*, '.', which matches any symbol. To effect the extension, all that is required is that the Peq table be set up to model the potential text symbol matches. That is, if p_i now denotes the set of symbols matching the ith position of the query, then one sets $Peq[t](i) \equiv (t \in p_i)$.

4.2. The Blocks Model. In order to efficiently extend our basic algorithm to the general case where there is no restriction on the length of the pattern, we must understand how to encapsulate its result into modules or *blocks* that can be pieced together to solve larger problems. Just as we thought 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 [Masek and Paterson 1980; Wu et al. 1996], 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.

Along these same lines, we can think of our basic algorithm as effecting the $O(1)$ computation of $1 \times m$ blocks under the special circumstances that the horizontal input delta is always 0. More generally, we can use our result to effect the computation of $1 \times w$ blocks where the horizontal input delta may also be -1 or $+1$. The diagram at the left of Figure 7 depicts such a block and further terms it a *level b* block because it extends from row $(b - 1)w$ to row bw. By restricting our attention to only blocks on $O(m/w)$ levels, we are still able to cover any desired region of a d.p. matrix, and only $O(\sigma m/w)$ Eq-vectors need be precomputed for them.

Consider an n-sequence by m-sequence comparison problem for which an algorithm computes a region or *zone* of the dynamic programming matrix. There

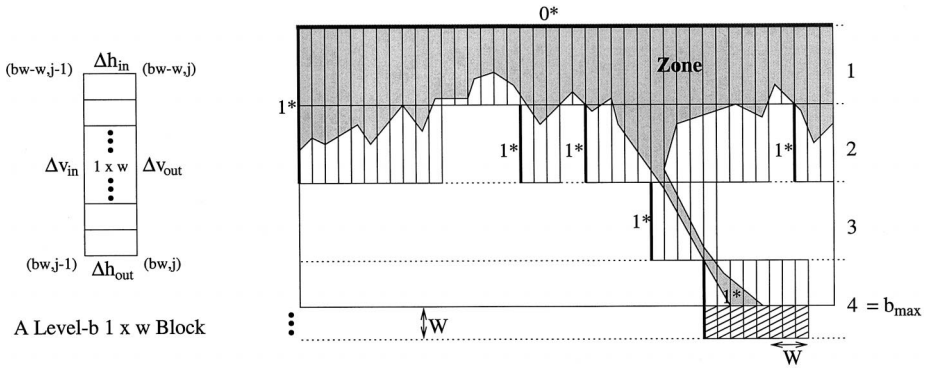

FIG. 7. Block-based dynamic programming.

are several such algorithms [Ukkonen 1985; Chao et al. 1992a; 1992b] and several others that compute small parts of such a matrix as a subprocedure.[4] Figure 7 at right shows a d.p. matrix and a hypothetical zone that might be computed by such an algorithm. The point of the figure is that we can take any such underlying computation and perform it in fewer steps by computing the region w cells at a time. Apart from the fact that any such tiling involves at most $b_{max} = \lceil m/w \rceil$ levels, there are several points to note:

(1) Almost invariably the computation can proceed in a column sweep so that only b_{max} vertical delta vectors need be maintained at any one time, that is, one can speak of *the* current vertical delta at level b.

(2) Blocks at the boundaries of the matrix have deltas of either 0 or 1 depending on the underlying computation. Figure 7 depicts 0-deltas on the upper boundary and 1-deltas on the left boundary of the matrix.

(3) Blocks that have no predecessor at the same level in the previous column can usually assume 1-deltas for their vertical inputs, as this conservatively models values greater than those in the zone.

(4) Blocks in the last level may extend beyond the last row by $W = w - m \pmod w$ cells. The easiest way to handle this is to pad the length m sequence with W extra wild-card symbols (see *Limited Regular Expressions* above). Under these circumstances, the value of the interior horizontal delta in row m, then appears at the output of the level-b_{max} block W columns later. This delay in output requires that one also pad the length n sequence with W wild-card symbols, and that one extend a tiling W columns beyond the end of the zone when in this last level. Figure 7 illustrates this.

For the sake of completeness and rigor, Figure 8 details a library that will maintain and manipulate blocks for a block-based approach as just described. The library assumes the client algorithm will proceed in a column sweep, and so has a statically allocated pair of bit-vectors arrays, `P`$[1 \ldots b_{max}]$ and `M`$[1 \ldots b_{max}]$, to encode the current vertical delta at each possible level. Moreover, Eq bits will be provided by the precomputed array $Peq[\sigma][1 \ldots b_{max}]$ for which

[4]See, for example, Ukkonen [1992], Myers [1994], Chang and Lawler [1994], and Sutinen and Tarhio [1996].

```
var int P[1..b_max], M[1..b_max], Peq[σ][1..b_max]

procedure Init_Block( b:1..b_max, vin:{0,1} )
 {  (P[b],M[b]) = (vin^w,0)  }

function Advance_Block( b:1..b_max, t:char, hin:{-1,0,1} ) : {-1,0,1}
 {  (Pv,Mv,Eq,hout) = (P[b],M[b],Peq[t][b],0)
    Lines 7-18 of Figure 6 with the following modifications:
        Replace Score with hout, and m with w.
        Add "if hin < 0 then Eq |= 1" between lines 7 and 8.
        Add "if hin < 0 then Mh |= 1 else if hin > 0 then Ph |= 1" between lines 16 and 17.
    (P[b],M[b]) = (Pv,Mv)
    return hout
 }
```

FIG. 8. The block-based library.

$Peq[s][b](i) \equiv (p_{(b-1)w+i} = s)$. The procedure `Init_Block` establishes all the vertical deltas of the level-b block to be `vin`. The function `Advance_Block`, takes the current level-b vertical delta vector, horizontal delta `hin`, and symbol `t`, as input. It computes the output of the block under such conditions, making the resulting vertical delta the current one, and returning the horizontal output delta, `hout`, as its functional result. The only fine point in the adaptation of the inner loop of the basic algorithm in Figure 6 to this purpose, is the observation that the horizontal input should be reflected into the vectors `Ph`, `Mh`, and `Eq` as detailed in footnotes 2 and 3 in the previous section on the basic algorithm.

In conclusion, one can improve the speed of any zone-based d.p. algorithm for approximate matching by a factor of w given that (1) the algorithm can be arranged to operate in a column (or row) sweep, (2) appropriate input delta's can be found for internal boundaries of the block tiling, and (3) one can effectively determine, from looking only at the level boundaries, whether the tiling contains the zone or not. We generally find these conditions to be true and we illustrate (3) for the $O(kn)$ expected-time algorithm of Ukkonen in the following discussion.

4.3. 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 $x_j = \max\{i\colon C(i, j) \leq k\}$ then the algorithm takes time proportional to the size of the zone $Z(k) = \cup_{j=0}^{n}\{(i, j)\colon i \in [0, x_j]\}$. It was shown in Chang and Lampe [1992] that the expected size of $Z(k)$ is $O(kn)$. Computing just the zone is easily accomplished with the observation that $x_j = \max\{i\colon i \leq x_{j-1} + 1$ and $C(i, j) \leq k\}$. Note that while some entries outside of the zone are computed in the event that $x_j \leq x_{j-1}$, the time for these can be charged to the corresponding entries in column $j - 1$. Another subtlety is that entries outside of the zone are not necessarily computed correctly but always have a value greater than k. However, values in the zone are correct, and so the algorithm correctly reports the positions of k-matches.

A block-based algorithm for this $O(kn)$ expected-time algorithm was devised and presented in an earlier paper of ours [Wu et al. 1996] 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.

In order for the block-based algorithm to tile $Z(k)$, it must know the value of $C(bw, j)$ at each active level-b boundary. This is accomplished by keeping an auxiliary array $Score[0 \ldots b_{max}]$ such that $Score[b]_j = C(bw, j)$ for any currently active blocks b in column j. Doing so is simply a matter of accumulating the horizontal delta returned by `Advance_Block` for block b, just as the basic algorithm accumulates the horizontal deltas in the last row. These *Score* values are then used to guide the tiling of the block-based algorithm that computes the blocks in levels 1 through y_j of column j, where y_j is computed from y_{j-1} as follows:

$$y_j = \begin{cases} y_{j-1} + 1 & \text{if } Score[b]_{j-1} = k \text{ and } y_{j-1} < b_{max} \text{ and} \\ & \quad (Peq[t_j][y_{j-1} + 1](1) \text{ or } Score[b]_j < k) \\ \max\{b : b \le y_{j-1} \text{ and} & \text{otherwise} \\ \quad C(bw, j) < k + w\} & \end{cases} \tag{11}$$

The induction is started by initializing blocks 1 through $y_0 = \lceil k/w \rceil$ in column 0. During the scan, we found it worthwhile to avoid computing the block at level $y_{j-1} + 1$ in column j, unless it is certain that $Z(k)$ properly intersects it. Furthermore, note that one can only ask to remove a block when the score at its level is greater or equal to $k + w$, rather than as soon as it does not intersect the zone because determining this would require an $O(w)$ test. While this leads to a slightly larger area than necessary being computed, it is still the case that the area covered by blocks is properly contained within the zone $Z(k + w)$ and so y_j is $O(k/w)$ in expectation by the theorem of Chang and Lampe [1992]. Thus, our block-based algorithm finds approximate k-matches in $O(kn/w)$ expected time. For completeness, we detail the algorithm in Figure 9 using subroutines of the block-based library in Figure 8.

5. *Some Empirical Results*

In this final section, we try to give some sense of the performance of our basic algorithm for the case where $m \le w$ and our block-based algorithm for the case of unrestricted m. All trials were run on a Dec Alpha 4/233 with 196Mb of memory and a 512Kb cache running the DEC UNIX 3.2 operating system. All algorithms considered were coded in ANSI-C and compiled under the GNU C compiler with the -O option on. Bit-vectors were modeled as 64-bit `unsigned long` integers. The Alpha architecture is optimized for this greater word length, and actually performs `unsigned long` loads, stores, and bit-ops *more efficiently* than it does so for `unsigned int`s.

We report on two sets of comparisons. The first is a study of our basic bit-vector algorithm, the bit-vector algorithm of Wu and Manber [1992] (and its specialization by Baeza-Yates and Gonnet [1992] for the case where $k = 0$), and a refinement of the bit-vector algorithm of Baeza-Yates and Navarro [1996]. The

```
1.  y = ⌈k/w⌉
2.  for b = 0, 2, ... y do
3.     Score[b] = bw
4.  for b = 1, 2, ... y do
5.     Init_Block(b)

6.  for j = 1, 2, ... n + W do

7.   { Carry = 0
8.     for b = 1, 2, ... y do
9.        Score[b] += (Carry = Advance_Block(b,t_j ,Carry))

10.    if Score[y] - Carry ≤ k and y < b_max and ((Peq[t_j][y+1] & 1) or Carry < 0) then
11.     { y += 1
12.       Init_Block(y)
13.       Score[y] = Score[y-1] + w - Carry + Advance_Block(y,t_j ,Carry)
        }
14.    else
15.       while Score[y] ≥ k + w do
16.          y -= 1

17.    if y = b_max and Score[y] ≤ k then
18.          print "Match at " · j-W
    }
```

FIG. 9. Block-based Ukkonen algorithm.

earlier result of Wright [1994] is not included as preliminary experiments indicated that it was noncompetitive even in the case of binary alphabets. In this first series of trials, we limit patterns to the case where m is not greater than 64, the number of bits in a bit-vector. The second set of experiments involves all verification-capable algorithms that work when k and m are unrestricted. In this case, we need only consider our block-based algorithm and the results of Chang and Lampe [1992] and Wu et al. [1996] and the refinement of Baeza-Yates and Navarro [1996], as all other competitors are already known to be dominated in practice by these algorithms [Wu et al. 1996]. Comparisons against the class of filter algorithms are not made in this preliminary study. It is true that they outperform all verification-capable algorithms for sufficiently small mismatch ratio k/m. Where, exactly, the line gets drawn is deferred to a broader study. Nonetheless, note that any filter algorithm needs a verification-capable algorithm as a subcomponent and hence benefits from a faster algorithm of this genre.

Our preliminary study thus involves seven algorithms, the two in this paper and those in Chang and Lampe [1992], Baeza-Yates and Gonnet [1992], Wu and Manber [1992], Wu et al. [1996], and Baeza-Yates and Navarro [1996]. We took the following steps to make the comparisons as fair as possible. First, the common components of reading and scanning the text and preprocessing the pattern were the same for all implementations. Thus, differences in times are all due to the difference in the body of the scan loop and not to issues such as how the I/O is performed. The code for the algorithm of Chang and Lampe [1992] was obtained from the authors and then further improved by us. The code for our earlier 4-Russians work was written by this author and was, of course, certainly his best effort. On the Alpha, this code ran best with a block size of 5, and this is the size used in the trials below. The code for the bit-vector methods of Baeza-Yates and Gonnet [1992], Wu and Manber [1992], and Baeza-Yates and Navarro [1996] was written by us exactly as detailed in the author's papers. It was then further optimized to remove all common subexpressions, minimize

loads and stores to memory, and move the evaluation of any constant terms in a loop to the start of the loop. We even went so far as to study the assembly code produced for the innermost loop of each algorithm to make sure that there were no obvious optimizations missed by the compiler. In one case [Baeza-Yates and Navarro 1996], we discovered that making three global variables local permitted the compiler to better use the available registers. Finally, over all cases, the inner-most loop averaged only 20 lines of code and so it was not difficult to give each implementation thorough consideration.

All the implementations, scanned data sets, and command-line scripts for all trials are available via anonymous ftp from the subdirectory `myers.grep` at `ftp.cs.arizona.edu`. We make this available not only for those interested in using our software, but also those who wish to perform further studies or to verify our methodology. There are two implementation details that deserve mention. The first is that our implementation of the Baeza-Yates and Navarro [1996] bit-vector algorithm does not include the filter that does not begin to run the underlying bit-vector algorithm until the character being scanned is one of the first $k + 1$ characters in the pattern. We do so, because this filter optimization is orthogonal and applicable to all of the algorithms being considered here. We want to get a sense of how the basic algorithms compare, such orthogonal optimizations can be layered on later. Secondly, we should mention that, in our implementation of our block-based algorithm, we (1) in-lined all the calls to `Init_Block` and `Advance_block`, (2) special-cased the `Advance_block` in line 13, and (3) otherwise optimized the code as noted above for our implementations of Wu and Manber [1992] and Baeza-Yates and Navarro [1996].

The expected-time complexity of each algorithm, A, is of the form

$$\Theta(f_A(m, k, \sigma)n),$$

where f_A is a function of the indicated parameters. Our experiments are aimed at empirically measuring f_A for each algorithm A. Each trial is designed to measure f_A for a given choice of the parameters. A trial consists of 10 runs of the program over a text of one million characters generated by random selection with equal probability from an alphabet of size σ, and a pattern that is a similarly generated sequence of m characters over the same alphabet as the text. The times measured for trials varied from 4 seconds to the 100's of seconds, and the system clock consistently measured run times to an accuracy of 0.1 seconds on our dedicated machine. Thus, the error in the time obtained from each trial is at most about 2.5%.

For the first set of experiments where $m \leq 64$, the algorithm of Baeza-Yates and Navarro [1996], as originally reported, only applies for $(m - k)(k + 2) \leq 64$ when a single word of bits is available. Partitioning the underlying automaton into blocks, each with few enough states to fit in a word, permits its application to larger m and k. Moreover, just as we compute only the blocks covering a zone of the d.p. matrix, one can compute just those blocks of the automaton with active states during the text scan to arrive at an $O(n\lceil k^2/\sqrt{\sigma w}\rceil)$ expected-time variation of the basic algorithm. In the case where $k \leq w - 2 = 62$, the automaton can also be partitioned in a way that leads to better code than when k is larger. The information for this refinement was communicated to us by the authors [Baeza-Yates and Navarro 1999], and we use it here in order to present

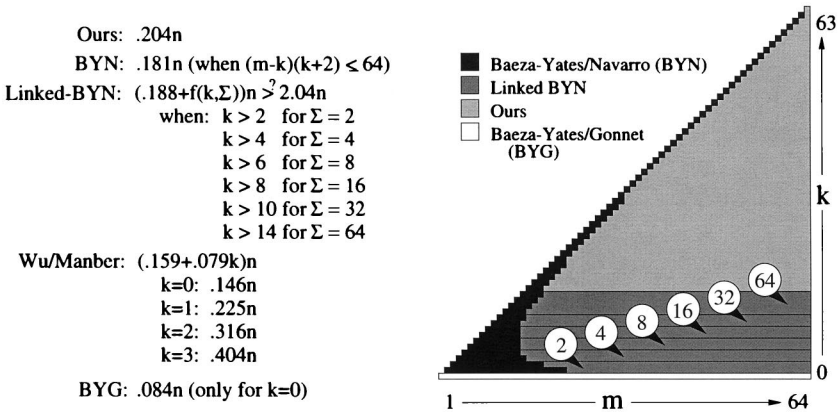

FIG. 10. Performance summary and legions of superiority for bit-vector algorithms.

their work in the best possible light. We refer to this variation as the "Linked-BYN" algorithm in the following discusion.

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 10. At left we show our best estimate for f_A for each algorithm. For our basic algorithm, f_A is a constant. We confirmed this by performing 90 trials with different values of k and m and observing that the time was constant save for a small variation due to semi-stochastic operating system fluctuations. The same is true of the basic Baeza-Yates and Navarro [1999] when $(m - k)(k + 2) \leq 64$. For the linked-variation, f_A depends on k and σ, so we ran experiments for every possible k and for σ equal to every power of 2. Finally, from theoretical considerations we know that the Wu and Manber algorithm should perform linearly in k. A least-squares regression line fits the results of 90 trials very well, but we note that the fit is off by roughly 9% for the first two values of k (see Figure 10). We hypothesize that this is due to the effect of branch-prediction in the instruction pipeline hardware. Figure 10 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 specialiation of the algorithm of Manber and Wu for this case by Baeza-Yates and Gonnet [1992] is 59% faster. In the remaining region, either our algorithm or the linked-BYN algorithm is superior depending on k and σ. Our algorithm is always superior in the light grey region, and superior in a medium grey region when σ is less than the alphabet size labeling the block.

Our second set of experiments are aimed at comparing verification-capable algorithms that can accommodate unrestricted choices of k and m. We argued previously that such a study need only involve our block-based algorithm and those of Chang and Lampe [1992], Wu et al. [1996], and the linked variation of Baeza-Yates and Navarro [1996]. All these are zone-based algorithms and when m is suitably large, the zone never reaches the last row of the d.p. matrix/automaton, so that the running time does not depend on m. Thus, we set $m = 400$ for all trials and ran 107 trials with $(k, \sigma) \in \{0, 1, 2, \ldots, 6, 8, 10, \ldots, 20, 24, 28, \ldots, 60\} \times \{2, 4, 8, 16, 32\} \cup \{64, 68, 72, \ldots, 120\} \times \{32\}$. The one exception is the linked-BYN algorithm which we did not run for $k > 62$ as for such k the approach requires additional partitioning which further

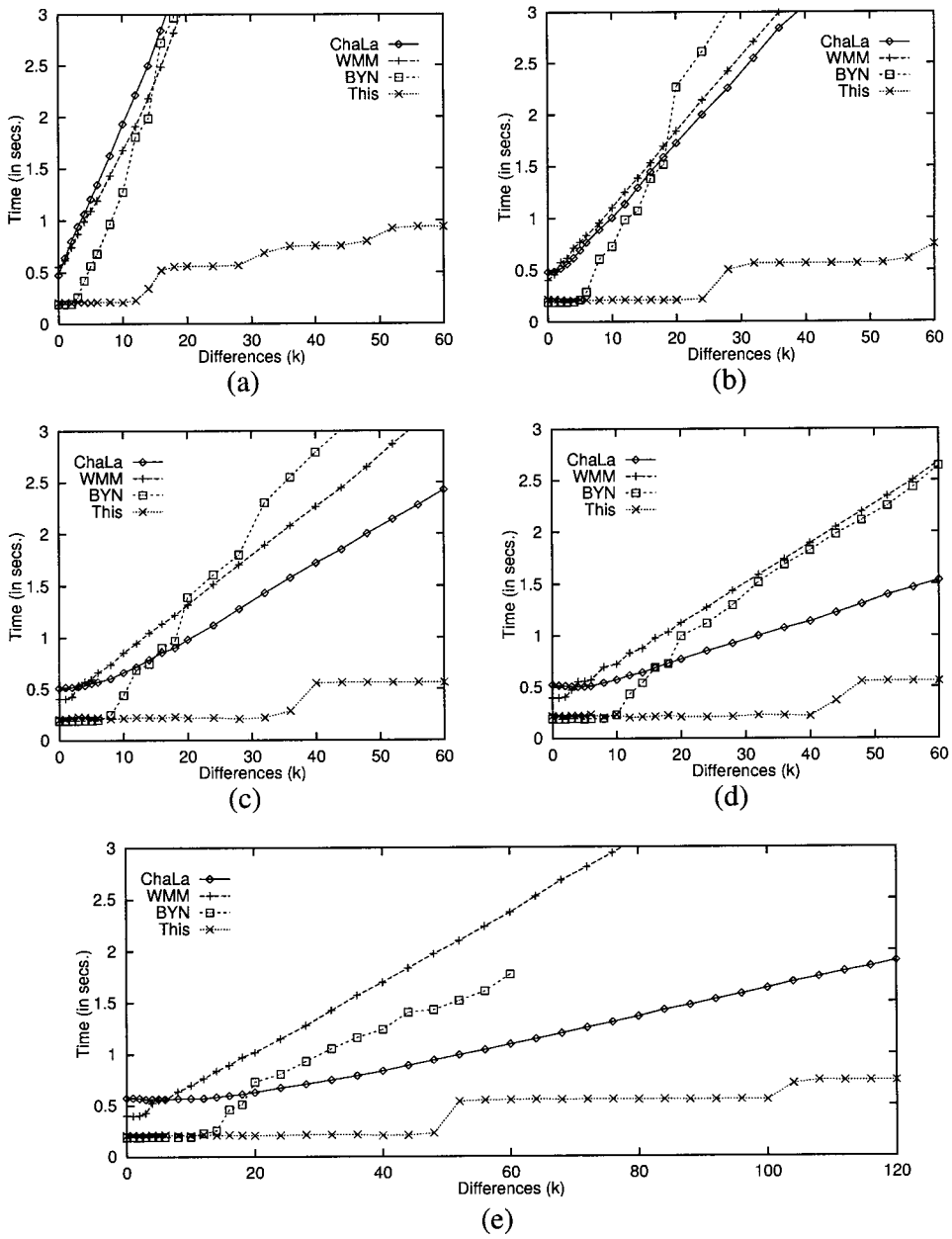

FIG. 11. Timing curves for the $O(kn/w)$ block-based algorithm ("This") versus Chang/Lampe ("ChaLa"), Wu/Manber/Myers ("WMM") with $\log s = 5$, and linked-Baeza-Yates/Navarro ("BYN"), with alphabet sizes (a) $\sigma = 2$, (b) $\sigma = 4$, (c) $\sigma = 8$, (d) $\sigma = 16$, and (e) $\sigma = 32$.

degrades its performance. For each of the five choices of σ, Figure 11 has a graph of time as a function of k, one curve for each algorithm. In the case of the 4-Russians algorithm [Wu et al. 1996], tables were built for evaluating the d.p. matrix in 1×5 blocks, the size at which the best performance is obtained as described in the original paper on this approach. From Figure 11, it is immediately clear that our block-based algorithm is superior to the others for all choices

of k and σ, except for a few values of k near 0 where the linked-BYN algorithm has a slight edge as expected from the previous experiment. The factor by which our algorithm is superior varies inversely with σ. We imagine that the Chang and Lampe may eventually overtake our algorithm for very large σ but we were only able to confirm that it does not do so for $\sigma = 95$, the number of printable ASCII characters and the largest setting possible for our software. Finally, note the discrete stair-step pattern of our algorithm due to discrete increases in the average number of blocks needed to cover each column of $Z(k)$.

While the study above is not exhaustive, it clearly shows that our bit-vector idea for approximate string matching leads to algorithms that are the best in practice for a wide range of operating conditions.

ACKNOWLEDGMENTS. The author is indebted to Marie-France Sagot for helpful conversations during the early development of this result, and to Will Evans, Toni Pitassi, and Maria Bonet for the suggestion that addition looked like it might provide the way to compute Xh quickly.

REFERENCES

BAEZA-YATES, R. A., AND GONNET, G. H. 1992. A new approach to text searching. *Commun. ACM 35*, 74–82.

BAEZA-YATES, R. A., AND NAVARRO, G. 1996. A faster algorithm for approximate string matching. In *Proceedings of the 7th Symposium on Combinatorial Pattern Matching.* Lecture Notes in Computer Science, Vol. 1075. Springer-Verlag, New York, pp. 1–23.

BAEZA-YATES, R. A. AND NAVARRO, G. 1999. Analysis for algorithm engineering: Improving an algorithm for approximate pattern matching. Unpublished manuscript.

CHAO, K. M., HARDISON, R. C., AND MILLER, W. 1992. Recent developments in linear-space alignment methods: A survey. *J. Comput. Biol. 1,* 271–291.

CHANG, W. I., AND LAMPE, J. 1992. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proceedings of the 3rd Symposium on Combinatorial Pattern Matching.* Lecture Notes in Computer Science, vol. 644. Springer-Verlag, New York, pp. 172–181.

CHANG, W. I. AND LAWLER, E. L. 1994. Sublinear expected time approximate matching and biological applications. *Algorithmica 12*, 327–344.

CHAO, K. M., HARDISON, R. C., AND MILLER, W. 1992. Recent developments in linear-space alignment methods: A survey. *J. Comput. Biol. 1*, 271–291.

CHAO, K. M., PEARSON, W. R., AND MILLER, W. 1992. Aligning two sequences within a specified diagonal band. *Comput. Appl. BioSciences 8*, 481–487.

COBBS, A. 1995. Fast approximate matching using suffix trees. In *Proceedings of the 6th Symposium on Combinatorial Pattern Matching.* Lecture Notes in Computer Science, vol. 937. Springer-Verlag, New York, pp. 41–54.

GALIL, Z., AND PARK, K. 1990. An improved algorithm for approximate string matching. *SIAM J. Comput. 19*, 989–999.

LANDAU, G. M., AND VISHKIN, U. 1988. Fast string matching with k differences. *J. Comput. Syst. Sci. 37*, 63–78.

MASEK, W. J., AND PATERSON, M. S. 1980. A faster algorithm for computing string edit distances. *J. Comput. Syst. Sci. 20*, 18–31.

MYERS, E. W. 1994. A sublinear algorithm for approximate keywords searching. *Algorithmica 12*, 345–374.

PEVZNER, P., AND WATERMAN, M. S. 1995. Multiple filtration and approximate pattern matching. *Algorithmica 13*, 135–154.

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

SUTINEN, E., AND TARHIO, J. 1996. Filtration with q-samples in approximate string matching. In *Proceedings of the 7th Symposium on Combinatorial Pattern Matching.* Lecture Notes in Computer Science, vol. 1075. Springer-Verlag, New York, pp. 50–63.

UKKONEN, E. 1985. Finding approximate patterns in strings. *J. Algorithms 6*, 132–137.

UKKONEN, E. 1992. Approximate string-matching with q-grams and maximal matches. *Theoret. Comput. Sci. 92*, 191–211.

UKKONEN, E. 1993. Approximate string matching over suffix trees. In *Proceedings of the 4th Symposium on Combinatorial Pattern Matching.* Lecture Notes in Computer Science, vol. 684. Springer-Verlag, New York, pp. 228–242.

WAGNER, R. A., AND FISCHER, M. J. 1974. The string to string correction problem. *J. ACM 21*, 168–173.

WU, S., AND MANBER, U. 1992. Fast text searching allowing errors. *Commun. ACM 35*, 10, 83–91.

WU, S., MANBER, U., AND MYERS, G. 1996. A subquadratic algorithm for approximate limited expression matching. *Algorithmica 15*, 50–67.

WRIGHT, A. H. 1994. Approximate string matching using within-word parallelism. *Soft. Pract. Exper. 24*, 337–362.

RECEIVED SEPTEMBER 1997; REVISED OCTOBER 1998; ACCEPTED NOVEMBER 1998