Approximate String Matching using Within-Word Parallelism

Alden H. Wright
Computer Science
University of Montana
Missoula, MT 59812

May 26, 1999

Software Practice and Experience, 24(4), 337-362 (April 1994)

SUMMARY

Given a text string, a pattern string, and an integer k, the problem of approximate string matching with k differences is to find all substrings of the text string whose edit distance from the pattern string is less than k. The edit distance between two string is defined as the minimum number of differences, where a difference can be a substitution, insertion, or deletion of a single character. An implementation of the dynamic programming algorithm for this problem is given that packs several characters and mod-4 integers into a computer word. Thus, it is a parallelization of the conventional implementation that runs on ordinary processors. Since a small alphabet means that characters have short binary codes, the degree of parallelism is greatest for small alphabets and for processors with long words. For an alphabet of size 8 or smaller and a 64 bit processor, a 21-fold parallelism over the conventional algorithm can be obtained. Empirical comparisons to the basic dynamic programming algorithm, to a version of Ukkonen's algorithm, and to the algorithm of Galil and Park, are given.

KEY WORDS Approximate string matching Approximate string searching Edit distance Within-word parallelism Parallel algorithm Parallel computation.

INTRODUCTION

This paper shows how the basic dynamic programming problem for the approximate string matching problem can be "parallelized" by using "pieces" of computer words. This technique is most useful when the alphabet is small and when the word size of the processor is large. For example, when the word size is 64 bits and the alphabet size is 8 (using three bits per character), the degree of parallelism is 21. Empirical results indicate an approximate speedup of 4.6 over the basic dynamic programming algorithm for alphabet size 8 on a 32 bit processor, and an approximate speedup of 2.2 for alphabet size 64 on a 32 bit processor. The speedups on a 64 bit processor should double these.

The *edit distance* between strings $X = X[1..m] = X[1]X[2] \ldots X[m]$ and $Y = Y[1..n] = Y[1]Y[2] \ldots Y[n]$ over an alphabet is the number of *differences* between them such that a correction of the differences converts one string the other. Differences can be substitutions, insertions, or deletions. For example, the edit distance between the strings *abcefgh* and *agcdfh* is 3, as suggested by the following alignment:

$$\begin{array}{ccccccc} a & b & c & - & e & f & g \\ | & | & | & | & | & | & | \\ a & m & c & d & e & - & g \end{array}$$

There are three differences: m is substituted for b, d is inserted, and f is deleted.

Let P be a pattern string of length m, let T be a text string of length n where n is typically much larger than m, and let k be a fixed integer less than or equal to m. The *approximate string matching problem with k differences* is to find all substrings of T which have edit distance less than k from P. The *approximate string matching problem with k mismatches* is to find all substrings of T which can be transformed into P by k substitutions. This paper considers the former problem, which will be referred to as the *k-approximate string matching problem*.

The basic dynamic programming algorithm to find edit distance between two strings and the sequence of edit operations to transform one string was first introduced by Needleman and Wunsch

[Needleman70] and was discovered independently by many other authors. The book [Sankoff83] discusses this algorithm in detail and gives historical background and many applications. The basic dynamic programming algorithm is also discussed in the next section. Ukkonen [Ukkonen83], [Ukkonen85b] gives an alternative algorithm, one version of which is tested empirically in this paper.

Ukkonen [Ukkonen85a] gave an algorithm for the k differences problem which runs in $O(m|\Sigma|G+n)$ time where $|\Sigma|$ is the size of the alphabet and $G = \min(3^m, 2^k|\Sigma|m^{k+1})$. The exponential preprocessing cost of this algorithm makes it impractical for all but very short patterns. Sellers [Sellers80] showed how to adapt the edit distance dynamic programming algorithm to the k differences approximate string matching problem. Landau and Vishkin [Landau89] showed how a version of Ukkonen's edit-distance algorithm can be adapted to the approximate string matching problem with k differences. Landau and Vishkin [Landau89] gave an algorithm which runs in $O(k^2n + m\log(\max(m,|\Sigma|)))$ time. This algorithm uses a suffix tree, which makes it impractical in practice. A more practical version runs in $O(k^2n + m^2)$ time. In [Landau89], Landau and Vishkin gave an algorithm which is $O(kn)$ for fixed size alphabet. This algorithm also uses a suffix tree, which limits its practicality. Galil and Park give both an adaptation of Ukkonen's edit-distance algorithm and a new algorithm for the k differences problem. The practical version of their new algorithm runs in $O(kn + m^2)$ time, and does not use a suffix tree. Our empirical results show that although the adaptation of Ukkonen's algorithm is $O(mn)$ in worst case, it is considerably faster in average practical complexity than the Galil and Park algorithm. Landau and Vishkin [Landau89] also gave a parallel CRCW PRAM algorithm that runs in time $O(\log m + k)$ using n processors.

THE BASIC DYNAMIC PROGRAMMING ALGORITHM

There are closely related dynamic programming algorithms for computing the edit distance between two strings and for solving the approximate string matching problem. To compute the edit distance between string $X = X[1..m]$ and string $Y = Y[1..n]$, an integer matrix $D[0..m,0..n]$ is defined as follows:

$$D[i,j] = edit\_distance(X[1..i], Y[1..j]).$$

Since the distance from the empty string to a string a length m is m, $D[m,0] = m$. Similarly, $D[0,n] = n$. The D matrix can be computed using:

$$D[i,j] = \min \begin{cases} D[i-1, j] \\ D[i-1, j-1] + c_{ij} \\ D[i, j-1] \end{cases} \tag{1}$$

where $c_{ij} = 0$ if $X[i] = Y[j]$ and $c_{ij} = 1$ otherwise. The first component of the minimum corresponds to an insertion operation, the second to a substitution operation, and the third to a deletion operation. The edit distance between X and Y is given as $D[m,n]$. The algorithm consists of computing the D matrix according to the above formulas. The matrix can be computed by rows, by columns, or by secondary diagonals. (A secondary diagonal of D consists of those elements $D[i,j]$ such that $i+j$ is a constant.) Clearly, the complexity of this algorithm is $\Theta(mn)$.

The edit distance dynamic programming algorithm can be adapted to the approximate string matching problem [Sellers80], [Erickson83], [Landau89]. The matrix element $D[i,j]$ is defined to be the minimum edit distance from $P[1..i]$ to any substring of T that ends at position j. Since the distance from the empty string $P[1..0]$ to the empty substring of T that ends at position j is 0, $D[0, j] = 0$ for all j. As with the edit distance problem, $D[i,0] = i$. Equation (1) (with X replaced by P and Y replaced by T) holds for this D matrix as well, and D can be computed similarly.

VARIATIONS ON THE DYNAMIC PROGRAMMING ALGORITHM

In this section, we introduce a number of variations on the basic dynamic programming algorithm for approximate string matching with k differences that lead up to the new implementation.

The dynamic programming algorithm to compute the D matrix for the k-approximate string matching problem by secondary diagonals is given as Algorithm 1:

procedure approx_edit_dis_matrix
 input: Pattern string: $A[1..m]$

Text string: $B[1..n]$

output: Result vector: $D[m, 1..n]$, where $D[m, j] = \min\{\text{edit\_dis}(P, T[k..j]) : k \in \{1,2,\ldots j+1\}\}$

begin

for $j \leftarrow 0$ **to** n **do** $D[0, j] \leftarrow 0$ **end for**

for $i \leftarrow 1$ **to** m **do** $D[i, j] \leftarrow i$ **end for**

for $j \leftarrow 1$ **to** $n + m$ **do**

for $k \leftarrow 1$ **to** m **do**

$i \leftarrow j - k + 1$

if $1 \le i$ and $j \le n$ **then**

if $P[i] = T[j]$ **then** $c_{ij} \leftarrow 1$ **else** $c_{ij} \leftarrow 0$ **end if**

$D[i, j] \leftarrow \min(c_{ij}, D[i-1, j], D[i, j-1])$

end if

end for

end for

end procedure

Algorithm 1.

The result of Algorithm 1 is the sequence of numbers in row m of the D matrix. Thus, $D[m, j]$ is the minimum edit distance from P to any substring of T which ends at position j of T. The substring of T that realizes this minimum distance, and the sequence of edit operations that transforms P into this substring of T can be found by working backward in the D matrix starting with $D[m, j]$.

Next, we show that the computations in the dynamic programming algorithm can be done using mod 4 arithmetic. The basic idea of using mod 4 arithmetic is due to Lipton and Lopresti [Lipton85].

LEMMA 1 [Ukkonen85b]. For every $D[i, j]$, $1 \le i \le m$ and $1 \le j \le n$, $D[i, j] = D[i-1, j-1]$ or $D[i, j] = D[i-1, j-1] + 1$.

LEMMA 2. ([Lipton85] contains a similar lemma.) For every $1 \le i \le m$ and $1 \le j \le n$, $D[i, j]$ and $D[i-1, j]$ differ by at most one, and $D[i, j]$ and $D[i, j-1]$ differ by at most one.

Proof. Since $D[i, j]$ represents $edit\_distance(X[1..i], Y[1..j])$, there is a sequence of $D[i, j]$ edit operations which transforms $X[1..i]$ into $Y[1..j]$. One additional insertion operation can be added to the beginning of the sequence so that it will transform $X[1..i-1]$ into $Y[1..j]$. This shows that $D[i-1,j] \le D[i,j]+1$. Similarly, $D[i,j] \le D[i-1,j]+1$, which proves the first half of the lemma. The proof of the second half of the lemma is similar.◊

From these two lemmas, formula (1) can be replaced by:

$$D[i,j] = \begin{cases} D[i-1,j-1] & \text{if} \begin{cases} c_{ij} = 0 \\ \text{or } D[i-1,j] = D[i,j]-1 \\ \text{or } D[i,j-1] = D[i,j]-1 \end{cases} \\ D[i-1,j-1]+1 & \textit{otherwise} \end{cases} \tag{2}$$

In addition, Algorithm 1 can be modified by replacing the two lines:

if $P[i] = T[j]$ **then** $c_{ij} \leftarrow 1$ **else** $c_{ij} \leftarrow 0$ **end if**
$D[i,j] \leftarrow \min(c_{ij}, D[i-1,j], D[i,j-1])$

with the following:

if $p_i = t_j$ or $D[i-1,j] = D[i,j]-1$ or $D[i,j-1] = D[i,j]-1$ **then**
  $D[i,j] \leftarrow D[i-1,j-1]$
else
  $D[i,j] \leftarrow D[i-1,j-1]+1$
endif

Clearly, the D matrix can be computed using mod 4 arithmetic by employing the modified Algorithm 1. To recover the edit distance, the bottom row entries (or the rightmost column entries) can be computed using full integer arithmetic.

Algorithm 2 conserves space by saving three consecutive diagonals as vectors. The vectors *previous_diag, current_diag,* and *next_diag* are vectors of mod 4 integers.

procedure approx_edit_dis_diagonal
  input: Pattern string: $P[1..m]$
    Text string: $T[1..n]$
  output: $result[1..n]$, where $result[j] = \min\{ \text{edit\_dis}(P, T[k..j]) : k \in \{1,2,... j+1\} \}$
begin

```
    mod_4_convert[−3] ←  1;          { used to convert a difference from mod 4 to an integer }
    mod_4_convert[−1] ← −1;
    mod_4_convert[ 0] ←  0;
    mod_4_convert[ 1] ←  1;
    mod_4_convert[ 3] ← −1;
    current_diag[0] ← 0;
    current_diag[1] ← 1;
    previous_diag[0] ← 0;
    result[0] ← 0;
    for t ← 2 to m + n do
        for i ← 1 to min(t - 1, m) do
            if  P[i] = T[t-i]
                or current_diag[i] = mod( previous_diag[i-1] - 1, 4 )
                or current_diag[i-1] = mod( previous_diag[i-1] - 1, 4 )
            then
                next_diag[i] ← previous_diag[i-1];
            else
                next_diag[i] ← mod( previous_diag[i-1] + 1, 4 );
        end for
        if t ≤ m then
            next_diag[t] ← mod( t, 4 );
        else
            result[t - m] ← result[t - m - 1] + mod_4_convert[ next_diag[m] - current_diag[m] ];
        end if
        for i ← 1 to t + 1 do
            previous_diag[i] ← current_diag[i];
            current_diag[i] ← next_diag[i];
        end for
    end for
end
```

Algorithm 2.

The new algorithm (Algorithms 3 and 4) described by this paper is a "parallelization" of Algorithm 2.

DESCRIPTION OF A SPECIAL CASE OF THE NEW IMPLEMENTATION: SINGLE WORD PATTERN

This section describes the special case of the new implementation where the pattern can be packed into a single word. This special case include most of the complications of the general case.

The new implementation is a parallelization of Algorithm 2. The parallelization is done by packing several mod 4 integers and characters into a computer word.

A mod 4 integer is represented in 3 consecutive bits of a word – 2 bits to represent the integer, and 1 bit for "carry". The carry bit allows mod 4 addition and subtraction to be done without having results from one integer interfere with an adjacent integer. Thus, a 32-bit word can store a vector of 10 mod 4 integers, and a 64-bit word can store a vector of 21 mod 4 integers.

However, the degree of parallelism is also limited by the number of bits needed to store characters, which depends on the alphabet size. As one would expect, it takes $\lceil \log_2 |\Sigma| \rceil$ bits to represent a character where the alphabet size is $|\Sigma|$.

To simplify the exposition, Algorithm 3 is the special case where all of the P string can be packed into a single word using at least 3 bits per character. Let BITS_PER_CHAR denote the number of bits used to store one character, and let CHARS_PER_WORD be the number of characters in a word. We assume that the characters of P are packed into a single computer word, with the first character of P going into the low end of the word, and we assume that the characters of T are packed into a sequence of words, with the high end of T going into the high end of the first word.

The operations NOT, AND, OR, and XOR are defined to be bitwise operations on words, and we use the following notation, taken from the C language, for bitwise shifts on words:

$w << k$	Word w shifted to the left by k bits.
$w >> k$	Word w shifted to the right by k bits.

Note that $w << k$ is equivalent to considering w to be an unsigned integer, and multiplying w by 2^k, where any overflow is discarded. Also, $w >> k$ is equivalent to considering w to be an unsigned integer, and dividing by 2^k.

procedure approx_edit_dis_packed
  input: Packed pattern string: *A_packed*[0] { Packed low-to-high }
    Packed text string: *B_packed*[1..*n'*] { Packed high-to-low in *B_packed*[1], . . . , }
      where $n' = \lceil n/\text{CHARS\_PER\_WORD} \rceil$.
  output: *result*[1..*n*], where *result*[*j*] = min{ edit_dis(*A_packed*, *B_packed*[*k*..*j*]) : $k \in$ {1,2,... *j*+1} }
begin

```
{ Initialization }
B_packed[0] = 0;
j ← 0;                                  { index for B }
r ← 0;                                  { size of shift of B words }
s ← BITS_PER_CHAR*CHARS_PER_WORD; { size of shift of B }
curr_diag ← 1;                          { current diagonal }
prev_diag ← 0;                          { previous diagonal }
prev_diag_shift;                        { previous diagonal shifted }
curr_result ← m;
result_mod_4 ← m  mod  4;
unit ← 1;                               { used in the computation of  init }
init ← 1;                               { used to initialize the "left border" of the matrix }
mod_4_convert[−3] ←   1;                { used to convert a difference from mod 4 to an integer }
mod_4_convert[−1] ← −1;
mod_4_convert[  0] ←   0;
mod_4_convert[  1] ←   1;
mod_4_convert[  3] ← −1;
result_shift ← BITS_PER_CHAR*(m-1);
{ The following are masks }
curr_mask ← 0;                          { mask for currently used part of word }
result_mask ← 3 << result_shift;        { mask to select result  }
m1 ←   11 . . . 11  octal;              { single word mask with 1's in one's bits }
m3 ←   33 . . . 33  octal;              { single word mask with 1's in one's and two's bits }
m4 ←   44 . . . 44  octal;              { single word mask with 1's in four's bits }
```

```
{ Main loop }
for t ← 2 to  m + n  do
    prev_diag_shift ← prev_diag << BITS_PER_CHAR;
    r  ← r + BITS_PER_CHAR;
    s  ← s - BITS_PER_CHAR;
    if s = 0  then
        r ← 0;
        s ← BITS_PER_CHAR*CHARS_PER_WORD;
        j ← j + 1;
        { first part of test for equality (uses XOR) }
        c ←  NOT (A_packed[0]  XOR  B_packed[j]);
    else
        { first part of test for equality }
        c ←  NOT (A_packed[0]  XOR  (B_packed[j] << r  OR  B_packed[j+1] >> s));
    end if
    { The following five lines can be converted to a single assignment if BITS_PER_CHAR is known
}
    e ←  c;
    for  k ←  2  to  BITS_PER_CHAR  do
        c ←  c  >> 1;
        e ←  e  AND  c;
    end for
    p1 ← (curr_diag  OR  m4) - prev_diag_shift;
    p2 ← ((curr_diag  OR  m4) - prev_diag) << BITS_PER_CHAR;
    p3 ← NOT  (((p1  OR  p2) >> 1)  OR  e)  AND  m1;
    prev_diag ← curr_diag;
    curr_diag ← (( prev_diag_shift + p3 )  AND  curr_mask)  OR  init;
    if  t  ≤ m  then
        unit ← unit << BITS_PER_CHAR;
        init ← ((init << BITS_PER_CHAR) + unit)  AND  m3;
        curr_mask ← (curr_mask << BITS_PER_CHAR)  OR  3;
        curr_diag ← curr_diag OR init;
    else
        prev_result_mod_4 ← result_mod_4;
        result_mod_4 ← curr_diag >> result_shift;
        curr_result ←
```

curr_result + *mod_4_convert*[*result_mod_*4 - *prev_result_mod_*4];
result[*t* - *m*] ← *curr_result*;
end if
end for
end

Algorithm 3

Explanation of Algorithm 3. In Algorithm 3, the variables *curr_diag* and *prev_diag* contain packed representations of the secondary diagonals of the dynamic programming matrix. These secondary diagonals do not contain row 0 of the matrix since this row consists entirely of zeros. Initially, *curr_diag* represents the secondary diagonal 1 (whose entries satisfy $i + j = 1$). Iteration t of the main loop computes secondary diagonal t as *next_diag*. The variable *prev_diag_shift* contains *prev_diag* shifted one piece to the right.

In the following, we assume, for explanatory purposes only, that characters and integers are stored in 3 bit pieces. This allows us to use octal notation for words, since octal digits correspond naturally to the pieces of a word. We will also assume five pieces per word (i. e., a word size of 15 or 16) even though this is unrealistically small in practice. Thus, we assume that the constant BITS_PER_CHAR is set to 3, and the constant CHARS_PER_WORD is set to 5. We will refer to the low order bit of a 3-bit piece as the one's bit, the middle bit as the two's bit, and the high order bit as the four's bit.

The algorithm uses a number of masks. First, the masks $m1$, $m3$, $m4$ are constants as follows:

$m1 = 11111$

$m3 = 33333$

$m4 = 44444$

The mask *current_mask* is used to mask the part of the word currently being used for the pattern. At the beginning of the first major loop, *current_mask* is set to 00003, and then on each iteration (up to the m^{th}) another 3 is added on the left. The mask *result_mask* is used to mask the result, i. e. the part of the diagonal that corresponds to row m of the D matrix. The integer *result_shift* is used to shift this result to the low end of the word.

The words *unit* and *init* are used in the first major loop to initialize the left end of *curr_diag* to correspond to column 0 of the D matrix. The word *unit* contains a 1 in the position corresponding to column 0, and on the t^{th} iteration of the first major loop, *init* contains t mod 4 in the position corresponding to column 0. Thus, on succeeding iterations (up to the m^{th}), *init* has values 20, 300, 4000, 00000, etc.

The variable $p1$ corresponds to the results of the test:

$$current\_diag[i] = \text{mod}(\ previous\_diag[i\text{-}1] - 1, 4\ ) \qquad (3)$$

in Algorithm 2. The two's bit of the piece of $p1$ corresponding to i is set to 1 if the result of the test is true, and to 0 if the result of the test is false. The computation of $p1$ adds a carry bit to each piece of *curr_diag* by OR'ing with the mask $m4$, and then subtracts *prev_diag_shift*. Note that *prev_diag_shift* is used instead of *prev_diag* because of the index i - 1 in formula (3). Lemma 2 implies that the integers $current\_diag[i]$ and $previous\_diag[i\text{-}1]$ differ by at most 1. Thus, if equality (3) holds, the result of the mod 4 subtraction (*curr_diag* OR $m4$) - *prev_diag_shift* will be 3 for the corresponding piece, and if equality (4) does not hold (*curr_diag* OR $m4$) - *prev_diag_shift* will be 0 or 1 for the corresponding piece. Thus, the two's bit for the piece is set if and only if (4) holds. Similarly, the variable $p2$ corresponds to the results of the Algorithm 2 test:

$$current\_diag[i\text{-}1] = \text{mod}(\ previous\_diag[i\text{-}1] - 1, 4\ ). \qquad (4)$$

In order to do the Algorithm 2 comparisons $P[i] = T[t\text{-}i]$ for $i =$ 1, 2, . . . min(t, m), the characters $T[t - 1]$ to $T[t - \min(t - 1, m)]$ are loaded into a single packed word c so that they can be compared with the pattern. The index j keeps track of which word or words are used to construct c in Algorithm 3. The integer variables r and s give the size of the necessary shifts of two consecutive packed words *B_packed*[j] and *B_packed*[j+1]. The actual comparison is done with the XOR operation; the result of the comparison is negated and stored in c. The next loop, the **for** k loop, combines the bit results of the comparison into a single bit, namely the one's bit of the variable e. If the constant BITS_PER_CHAR is known at compile time, then this loop can be replaced by a single statement. For example, if bits per char is 3, the statement is:

$$e \leftarrow ((c >> 2) \text{ AND } (c >> 1) \text{ AND } c)$$

Example of Algorithm 3. An example is given to illustrate Algorithm 3 with alphabet $\Sigma = \{a, b, c, d\}$ as Table 1. The pattern string is "dbcd" with $m = 4$, and the text string is "cdbcdbcbc" with $n = 9$. Table 1 is the dynamic programming matrix for this example:

		c	d	b	c	d	b	c	b	c
col. index	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0
d	1	1	0	1	1	0	1	1	1	1
b	2	2	1	0	1	1	0	1	1	2
c	3	2	2	1	0	1	1	0	1	1
d	4	3	2	2	1	0	1	1	1	2

Table 1.

The packed pattern string in octal is: $A[0] = 3213$, and the packed text string is: $B[0] = 0$, $B[1] = 23123$, $B[2] = 12120$. After the initialization at the beginning of the body of the first main loop for $t = 1$, we have $unit = 10$, $current\_mask = 3$, $init = 20$, $prev\_diag = prev\_diag\_shift = 0$, $curr\_diag = 1$.

Table 2 contains a trace of some of the important variables of the algorithm. The variables e, $p1$, and $p2$ are displayed as masked by $m1$ or $m2$ to show the meaningful bits of each piece. Note that since the pattern length is 4, the leftmost piece of words is not meaningful.

t	*r*	*s*	*j*	*c*	*e* AND *m*1	*p*1 AND *m*2	*p*2 AND *m*2	*p*3	*curr_diag*
2	3	12	0	37135	10000	00000	00000	01111	0021
3	6	9	0	37131	10001	00200	00000	01010	0320
4	9	6	0	37173	10100	02020	00000	00001	0211
5	12	3	0	37557	10010	02200	00000	00001	3201
6	0	15	1	33711	01001	20200	00020	00000	2110
7	3	12	1	31373	00100	22020	00000	00001	2011
8	6	9	1	15557	00010	02000	20200	00001	1101
9	9	6	1	13711	01000	20200	20020	00001	0111
10	12	3	1	31377	00110	20000	22000	00001	1011
11	0	15	2	35511	00000	02000	00200	10011	1121
12	3	12	2	13375	00100	00000	20000	01011	1121
13	6	9	2	35535	00001	00200	20200	01010	2220

Table 2.

DESCRIPTION THE GENERAL CASE OF THE NEW IMPLEMENTATION: MULTIPLE WORD PATTERN

In this section we describe the general case of the new implementation of the dynamic programming algorithm for approximate string matching. The pattern string is packed into multiple words. Each word is packed low-to-high, and there is a one-piece overlap from one word to the next. For example, if the pattern string is "bcdefghabcdefg", then

A_packed[0] = 54321   *A_packed*[1] = 10765   *A_packed*[2] = 54321   *A_packed*[3] = 00065

This overlap is necessary since the computation of each piece of *curr_diag* needs the previous piece. Thus, the low-order piece of a word of *curr_diag*[*k*] for $k > 0$ is not computed correctly. Instead, it is computed as the high-order piece of *curr_diag*[*k*–1], and copied into *curr_diag*[*k*].

The main loop (with index *t*) is over the secondary diagonals of the dynamic programming matrix. The inner loop (with index *k*) is over the full words of the *A_packed*, in other words, the words of the *A_packed* that contain CHARS_PER_WORD packed characters. The variable *i* represents the number of full words of the pattern contained in the current diagonal. It increases as the length of the diagonal increases.

The part of the main loop after the **for** k loop handles the end of the diagonal that corresponds to a partially full word of *A_packed*. For $t \le m$ the algorithm builds the leftmost piece of *curr_diag* to correspond to the leftmost column of the D matrix of the dynamic programming algorithm. For $t > m$ the algorithm computes the result vector in the same way as was described in the previous section.

procedure approx_edit_dis_packed
input: Packed pattern string: A_packed[0..*m'*] { Packed low-to-high }
Packed text string: B_packed[1..*n'*] { Packed high-to-low in *B_packed*[1], *B_packed*[2], . . . , }
where $m' = \lceil (m-1)/(\text{CHARS\_PER\_WORD}-1) \rceil$ and $n' = \lceil (n-1)/\text{CHARS\_PER\_WORD}+1 \rceil$,
output: *result*[1..*n*], where *result*[*j*] = min{ edit_dis(*A_packed*, *B_packed*[*k*..*j*]) : $k \in \{1,2,\ldots j+1\}$ }
begin
{ Initialization }
B_packed[0] ← 0;
result_mod_4 ← *m* mod 4;
unit ← 1; { used in the computation of *init* }
init ← 1; { used to initialize the "left border" of the matrix }
mod_4_convert[−3] ← 1; { used to convert a difference from mod 4 to an integer }
mod_4_convert[−1] ← −1;
mod_4_convert[0] ← 0;
mod_4_convert[1] ← 1;
mod_4_convert[3] ← −1;
if *m* = 1 **then** *result_shift* ← 0;
else *result_shift* ← BITS_PER_CHAR*((*m* - 2) **mod** (CHARS_PER_WORD - 1) + 1);
end if
{ initialize the masks }
curr_mask ← 3; { mask for currently used part of word }
*m*1 ← 11 . . . 11 octal; { single word mask with 1's in one's bits }
*m*3 ← 33 . . . 33 octal; { single word mask with 1's in one's and two's bits }
*m*4 ← 44 . . . 44 octal; { single word mask with 1's in four's bits }
m3_high ← 33 . . . 30 octal; { *m3_high* = *m*3 AND (77 . . . 70 octal) }

{ Initialize diagonals and indices of B }
for *k* ← 0 **to** (*m* - 1)/(CHARS_PER_WORD - 1) + 1 **do**
curr_diag[*k*] ← 0;

```
        prev_diag[k] ← 0;
        j[k] ← 0;
        r[k] ← 0;
        s[k] ← BITS_PER_CHAR*CHARS_PER_WORD;
    end for
    curr_diag[0] ← 1;

    { Main loop }
    i ← 0;
    for t ← 2  to  m + n  do
        for k ← 0 to  i - 1  do
            prev_diag_shift ← prev_diag[k] << BITS_PER_CHAR;
            p ← (((curr_diag[k] OR m4) - prev_diag[k]) << BITS_PER_CHAR)
                    OR ((curr_diag[k] OR m4) - prev_diag_shift);
            r[k] ← r[k] + BITS_PER_CHAR;
            s[k] ← s[k] - BITS_PER_CHAR;
            if  s[k] = 0  then
                r[k] ← 0;
                s[k] ← BITS_PER_CHAR*CHARS_PER_WORD;
                j[k] ← j[k] + 1;
                c ← NOT (A_packed[k] XOR B_packed[j[k]]);
            else
                c ← NOT (A_packed[k]
                    XOR ((B_packed[j[k]] << r[k]) OR (B_packed[j[k]+1] >> s[k] )));
            end if
        { The following five lines can be converted to a single assignment if BITS_PER_CHAR is known}
            e ← c;
            for  kk ← 2  to  BITS_PER_CHAR  do
                c ← c >> 1;
                e ←  e AND c;
            end for
            p ← NOT ((p >> 1) OR e ) AND m1;
            prev_diag[k] ← curr_diag[k];
            curr_diag[k] ←  prev_diag_shift + p;

            if  k > 0  then
```

```
                curr_diag[k] ← (curr_diag[k] AND m3_high)
                    OR (curr_diag[k-1] >> (BITS_PER_CHAR*(CHARS_PER_WORD - 1)));
            else
                curr_diag[k] ← curr_diag[k] AND m3;
            end if
        end for
        prev_diag_shift ← prev_diag[i] << BITS_PER_CHAR;
        p ← (((curr_diag[i] OR m4) - prev_diag[i]) << BITS_PER_CHAR)
                    OR ((curr_diag[i] OR m4) - prev_diag_shift);
        r[i] ← r[i] + BITS_PER_CHAR;
        s[i] ← s[i] - BITS_PER_CHAR;
        if  s[i] = 0  then
            r[i] ← 0;
            s[i] ← BITS_PER_CHAR*CHARS_PER_WORD;
            j[i] ← j[i] + 1;
            c ← NOT (A_packed[i] XOR B_packed[j[i]]);
        else
            c ← NOT (A_packed[i] XOR (B_packed[j[i]] << r[i] OR B_packed[j[i]+1] >> s[i] ));
        end if
    { The following five lines can be converted to a single assignment if BITS_PER_CHAR is known }
        e ← c;
        for kk ← 2  to BITS_PER_CHAR  do
            c ← c >> 1;
            e ← e AND c;
        end for
        p ← NOT ((p >> 1) OR e ) AND m1;
        prev_diag[i] ← curr_diag[i];
        curr_diag[i] ← (( prev_diag_shift + p ) AND curr_mask);
        if  i > 0  then
            curr_diag[i] ← (curr_diag[i] AND m3_high)
                OR  (curr_diag[i-1] >> (BITS_PER_CHAR*(CHARS_PER_WORD - 1)));
        if  t ≤ m  then
            unit ← unit << BITS_PER_CHAR;
            init ← ((init << BITS_PER_CHAR) + unit) AND m3;
            curr_diag[i] |= init;
            if (t mod (CHARS_PER_WORD - 1) = 1) and (t < m)  then
                i ← i + 1;
```

```
            init ← init >> (BITS_PER_CHAR*(CHARS_PER_WORD - 1));
            unit ← 1;
            curr_diag[i] ← init;
            curr_mask ← 3;
        else
            curr_mask ← (curr_mask << BITS_PER_CHAR) OR 3;
        end if
    else
        previous_result_mod_4 ← result_mod_4;
        result_mod_4 ← curr_diag[i] >> result_shift;
        current_result ← current_result + mod_4_convert[result_mod_4 - previous_result_mod_4 + 3];
        result[ t - m ] ← current_result;
    end if
  end for
end
```

Algorithm 4.

LIMITATIONS OF THE NEW IMPLEMENTATION

The biggest limitation of the new implementation is that the costs of substitutions, insertions, and deletions are fixed at 1. Ukkonen's algorithm and the Galil-Park algorithm share this limitation. This does not allow for what the molecular biologist call *gap-penalties*, which are essentially different costs on insertions/deletions than on substitutions. This is one reason that we have concentrated on the approximate string matching problem rather than the edit-distance or alignment problem. The new implementation could be used as a fast method to find regions of interest in a long string or text database, and then a more flexible method could be used to compute edit distances or alignments.

The second obvious limitation is that the code for this implementation is much more complex than for either the conventional dynamic programming algorithm or for Ukkonen's algorithm.

EMPIRICAL RESULTS

In this section we compare the new implementation (Algorithm 4, called the packed algorithm) with the dynamic programming algorithm (Algorithm 2 without mod 4), the variation of Ukkonen's algorithm presented as Algorithm MN2 by Galil and Park in [Galil90], and the $O(kn+m^2)$ algorithm of Galil and Park [Galil90]. Algorithm MN2 (Ukkonen's algorithm) was implemented using three successive diagonals of the C matrix. The times for all algorithms include time for allocation of space for local arrays. The times for the packed algorithm include time for packing the pattern and text strings.

Both Ukkonen's algorithm and the Galil-Park algorithm require k as an input parameter, where k is the minimum acceptable edit distance. We specify k by giving a *match_criterion* which is a fraction (or percentage) of the pattern length m. Thus, $k = \lfloor match\_criterion \cdot m \rfloor$.

All algorithms were implemented in the C language using $O(m)$ space. Testing was done on an IBM RS6000 Model 340 (32-bit processor) using IBM's C compiler. All tests used randomly generated pattern and text strings, where characters were chosen from an alphabet of size 8.

A computer with a 64 bit word, such as a computer that uses the MIPS R4000 or the DEC Alpha processor, should improve the performance of the packed algorithm by a factor of 2 relative to the other algorithms.

Figure 1 compares the algorithms for different pattern lengths m. The packed algorithm was implemented for alphabet sizes 8 and 64. The text length was 300,000.

Figure 1 shows that all algorithms are almost linear in the pattern length m. The packed algorithm with a maximum alphabet size of 8 (ten 3-bit pieces per word) was approximately 4.6 times faster than the dynamic programming algorithm and $2\frac{1}{2}$ times faster than the Ukkonen's algorithm. The packed algorithm with a maximum alphabet size of 64 (five 6-bit pieces per word) was 2.2 times faster than the dynamic programming algorithm and just slightly faster than Ukkonen's algorithm. Ukkonen's algorithm was about $2\frac{1}{2}$ times faster than the Galil-Park algorithm.

Figure 2 shows the dependence of CPU time on the match criterion. Clearly, only the Ukkonen and Galil-Park algorithms depend on the match criterion, and in both of these the dependence is approximately linear. These results show that the packed algorithm for a maximum alphabet size of 8 will be better than Ukkonen's algorithm for a match criterion greater than about 11%, and the packed algorithm with a maximum alphabet size of 64 will be better than Ukkonen's algorithm for a match criterion greater than about 29%.

Figure 3 shows the dependence of the packed algorithm on maximum alphabet size (bits per character) for different pattern lengths. The pattern length was 100, and the text length was 100,000.

Figure 4 shows the dependence of CPU time on the bits per character. The pattern length was 100, and the text length was 100,000. Thus the CPU time increases approximately linearly with the bits per character.

CONCLUSIONS

A "byte-parallel" implementation of the dynamic programming algorithm for the *k* approximate string matching problem provides an substantial speedup over a conventional implementation, especially for small alphabets and long-word processors.

REFERENCES

[Erickson83] B. W. Erickson, and P. H. Sellers, 'Recognition of patterns in genetic sequences', in [Sankoff83], 55-91 (1983).

[Galil90] Zvi Galil and Kunsoo Park, 'An improved algorithm for approximate string matching', *SIAM J. Computing*, **19**, 989-999 (1990).

[Landau88] Gad M. Landau and Uzi Vishkin, 'Fast string matching with k differences', *J. Comput. System Sci.*, **37**, 63-78 (1988).

[Landau89] Gad M. Landau and Uzi Vishkin, 'Fast parallel and serial approximate string matching', *Journal of Algorithms*, **10**, 157-169 (1989).

[Lipton85] R. J. Lipton and D. Lopresti, ’A systolic array for rapid string comparison’, *Proceedings of* 1985 *Chapel Hill Conference on VLSI*, 363-376 (1985).

[Needleman70] S. B. Needleman and C. D. Wuncsh, ’A general method applicable to the search for similarities in the amino acid sequences of two proteins’, *J. Mol. Bio.*, **48**, 444-453 (1970).

[Sankoff83] D. Sankoff and J. B. Kruskal (Eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, (1983).

[Sellers80] P. H. Sellers, ’The theory and computation of evolutionary distances; pattern recognition,’ *J. Algorithms*}, **1**, 359-373, (1980).

[Ukkonen83] Esko Ukkonen, ’On approximate string matching’, *Proceedings Int. Conf. Found. Comput. Theory*, *Lecture Notes in Computer Science*, **158**, 487-495 (1983).

[Ukkonen85a] Esko Ukkonen, ’Finding approximate pattern in strings’, *J. Algorithms*, **6**, 132-137 (1985).

[Ukkonen85b] Esko Ukkonen, ’Algorithms for approximate string matching’, *Information and Control*, **64**, 100-118 (1985).

Wu and Manber [Wu92] give a bit-parallel algorithm for the k-differences problem which is based on the work of Baeza-Yates and Gonnet [Baeza-Yates92]. This algorithm has a preprocessing phase which requires $O\left(m|\Sigma| + k\left\lceil \frac{m}{w} \right\rceil\right)$ steps, where w is the word-length of the computer. Then the search phase runs in $O\left(nk\left\lceil \frac{m}{w} \right\rceil\right)$ steps. While this algorithm breaks no new ground in asymptotic complexity, it is very fast in practice for short patterns (especially when the pattern is shorter than the word length). It also can be extended to support patterns that include regular expressions.

The Wu-Manber algorithm was only implemented for patterns that could be stored in a single word. Since a 32-bit processor was used, this limited m to 32.

[Baeza-Yates92] R. Baeza-Yates, and G. H. Gonnet, 'A new approach to text searching', *Comm. Assoc. Computing. Mach.*, **35**, 75-82, (1992).

[Wu92] S. Wu, and U. Manber, 'Fast text searching allowing errors', *Comm. Assoc. Computing. Mach.*, 83--91, **35**, (1992), .