ELSEVIER

Information Processing Letters 70 (1999) 127–139

Information Processing Letters

www.elsevier.com/locate/ipl

A versatile divide and conquer technique for optimal string alignment

David R. Powell *, Lloyd Allison, Trevor I. Dix

Department of Computer Science, Monash University, Clayton, VIC 3168, Australia

Received 1 December 1997; received in revised form 1 March 1999
Communicated by R.G. Dromey

Abstract

Common string alignment algorithms such as the basic dynamic programming algorithm (DPA) and the time efficient Ukkonen algorithm use quadratic space to determine an alignment between two strings. In this paper we present a technique that can be applied to these algorithms to obtain an alignment using only linear space, while having little or no effect on the time complexity. This new technique has several advantages over previous methods for determining alignments in linear space, such as: simplicity, the ability to use essentially the same technique when using different cost functions, and the practical advantage of easily being able to trade available memory for running time.

Keywords: Algorithms; Sequence alignment; Dynamic programming; Edit distance

1. Introduction

Alignment algorithms are used to align strings optimally under a given cost function. These algorithms have a wide variety of different applications in molecular biology, such as DNA sequence alignment, protein sequence alignment [14] and protein structure alignment [17]. Aligning a pair of strings involves matching characters from the two strings either with each other or a null character, '-', to indicate an insertion or deletion. Fig. 1 shows an example alignment of two strings.

The well-known dynamic programming algorithm (DPA) [10,15] can be used to find an optimal alignment for a number of different cost functions. The DPA finds an alignment of minimum cost (an optimal alignment) for a given cost function. Typical cost functions include, in order of increasing complexity: simple costs, each mutation cost is a constant; linear (or affine) gap costs [4], insertions/deletions are costed via a linear function; piecewise linear gap costs [4] and concave gap costs [11]. An alignment can be considered as a way to edit one string into the other, thus the *cost* of aligning strings is sometimes called the *edit cost* or *edit distance*.

```
ATACTAG-A
| ||| | |
A-ACTTGGA
```

Alternatively:

```
<A,A> <T,-> <A,A> <C,C>
<T,T> <A,T> <G,G> <-,G> <A,A>
```

Fig. 1. An example alignment of the strings 'ATACTAGA' and 'AACTTGGA'.

* Corresponding author. Email: powell@cs.monash.edu.au.

0020-0190/99/$ – see front matter
PII: S0020-0190(99)00053-8

Assuming the strings are of similar length, $\sim n$, then the time complexity of the DPA is $O(n^2)$. The space complexity is $O(n^2)$ if an alignment is required, or just $O(n)$ if only the edit cost is desired. In this paper we briefly discuss the basic DPA and Hirschberg's [8] divide and conquer algorithm that allows the DPA to compute an alignment in $O(n)$ space. We then present an alternative to Hirschberg's algorithm which has a number of advantages. This alternative has been briefly described by Hirschberg [9], who attributes it to Eppstein (unpublished), and used to compute the basic edit distance. Here we show how it can be applied to many other cost functions and can also be combined with Ukkonen's fast algorithm (see below) to give a flexible, fast and space efficient alignment algorithm.

Ukkonen [18] devised an algorithm that runs faster than the basic DPA. On average, this algorithm has time complexity $O(n + d^2)$, where d is the edit cost between the two strings. If an alignment is required, $O(d^2)$ space is needed, otherwise $O(d)$ space can be used to determine the edit cost. We briefly explain Ukkonen's algorithm, and present how our method can be applied to reduce the space complexity of Ukkonen's algorithm to $O(d)$ while still producing an alignment (discussed in detail for both simple and linear gap costs). Previously, retrieval of an alignment in linear space from Ukkonen's algorithm was done by using Hirschberg's [8] technique which is more complicated (see [9, p. 137]) and less versatile than that presented here.

A recent paper by Grice et al. [7] uses a similar approach as that presented here, but they have a different motivation. They wish to train a Hidden Markov Model by using all possible paths through the DPA matrix. While we are interested only in an optimal alignment. Grice et al. do suggest a method for the single best path, however, it is more complicated than that presented here, and we also apply our technique to the more advanced Ukkonen algorithm (Section 4.1).

2. The basic DPA

The basic DPA with simple costs to align two strings (*As* and *Bs*) uses an edit distance matrix where each entry $D[x, y]$ contains the edit cost for strings $As[1..x]$ and $Bs[1..y]$. The algorithm is given in Fig. 2. To obtain an alignment using this algorithm it is necessary to trace back through the matrix from $D[|As|, |Bs|]$ to $D[0, 0]$ following the choices that were made in the min() function.

```
D[0,0] = 0
D[i,0] = i, i=1..|As|
D[0,j] = j, j=1..|Bs|

for j = 1..|Bs|
  for i = 1..|As|
    D[i,j] = min(D[i,j-1] + insertCost,
                 D[i-1,j] + deleteCost,
                 D[i-1,j-1] +
                  (if As[i] = Bs[j] then
                     matchCost
                   else
                     mismatchCost))
```

Fig. 2. The DPA to determine minimum edit distance.

It is obvious that the DPA uses $O(n^2)$ time and $O(n^2)$ space for the D matrix. However, each column of the DPA is calculated from the previous column only (or row), thus if an alignment is not required it is possible to use only $O(n)$ space (i.e., store only the previous and current columns). The simple modification to achieve this is to modify all column indexing to first be computed modulo 2 (e.g., $D[i, j] \rightarrow D[i, j \bmod 2]$).

2.1. Alignment in $O(n)$ space

Hirschberg [8] described an algorithm to determine the longest common subsequence (LCS) of two strings in $O(n)$ space and $O(n^2)$ time. The LCS problem is closely related to the edit distance problem. Hirschberg's algorithm splits the *Bs* string in half, i.e., at the middle column of the D matrix, and finds where in the D matrix the optimal alignment crosses this middle column. The crossing point is found by running the DPA forward on the first half of *Bs*, and in reverse on the last half. Where the forward and reverse calculations meet an optimal crossing point is found on the middle column and the crossing point is therefore known to lie on the optimal alignment. Two smaller alignment problems remain (the two halves of *Bs* against the corresponding parts of *As*), these are then solved recursively.

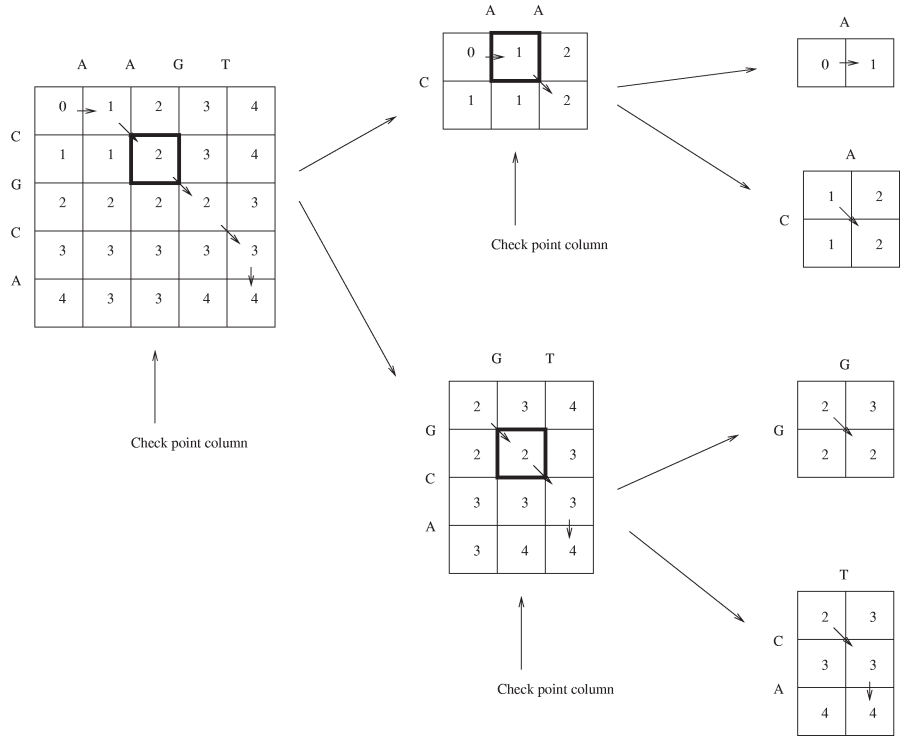

Fig. 3. An example of check pointing on the DPA for sequences CGCA and AAGT, using insertion, deletion and mismatch costs of 1 and a match cost of 0.

Each recursion step in Hirschberg's algorithm computes half as much area of the D matrix as the previous. Using $|D|$ to represent the size if the D matrix, the total computation is $|D| \times (1 + 1/2 + 1/4 + \cdots)$, the series converges to 2, thus since $|D|$ is $O(n^2)$, the time complexity is also $O(n^2)$. The space requirement is $O(n)$ because only a single column is needed at a time to allow the determination of the next column.

2.2. Check pointing

The algorithm described below has the same benefit as Hirschberg's (i.e., reduce the space requirement to linear in n without altering the time complexity from $O(n^2)$), but also has a number of other advantages. These include simpler implementation (especially for more complex cost functions), the ability to trade the constant in the space overhead for the constant in the running time, and the ability to be combined with the faster Ukkonen's algorithm (Section 4.1).

The idea of this algorithm is like Hirschberg's, that is, to split the D matrix continually in half. To achieve this it is necessary to find a cell on the middle column, column q say (i.e., $q = |Bs|/2$), that lies on the optimal alignment. We want to determine a row p such that the cell $D[p, q]$ lies on the optimal alignment. This is done by having every cell after column q carry the row index of the cell on column q it derived from. Once the D matrix is determined, the cell $D[|As|, |Bs|]$ will not only contain the edit cost, but also the desired row index p. Since this cell $D[p, q]$ is known to lie on the optimal alignment, the alignment problem can be divided into two sub problems. That is, the alignment from $D[0, 0]$ to $D[p, q]$ and from $D[p, q]$ to $D[|As|, |Bs|]$. So the algorithm then performs recursion on each of these regions. Only two columns of the D matrix are ever stored, the current column and the previous column. As with the cost only DPA (see last paragraph of Section 2), each column is dependent only on the previous column.

```
Procedure DPA_CP_Alignment(sRow, sCol, fRow, fCol)

  q = (sCol+fCol)/2                                          | Column to check point

  {Base conditions for D matrix and from matrix 'f'}
  D[sRow..fRow, sCol  mod 2] = sRow..fRow - sRow
  f[sRow..fRow, q mod 2] = sRow..fRow

  for j = sCol+1..fCol
    D[sRow, j mod 2] = j-sCol
    f[sRow, j mod 2] = sRow

    for i = sRow+1..fRow
      D[i, j mod 2] = min(D[i  ,(j-1) mod 2] + insertCost,   |
                          D[i-1, j    mod 2] + deleteCost,   |  Standard DPA calculation
                          D[i-1,(j-1) mod 2] +               |
                            (if As[i] = Bs[j] then           |
                               matchCost                     |
                             else                            |
                               mismatchCost))                |


      if (j>q) then                                          |  Carry CP information
        f[i, j mod 2] = f[(i,j) chosen in min() function]    |

    endfor
  endfor


  {Base Cases 1 or 2 cols}                                   |
  if (fCol-sCol = 1 or = 2) then                             |
    {Determine alignment directly from D matrix}             |  Base cases
    return                                                   |
  endif                                                      |


  p = f[fRow, fCol mod 2]                                    | Now have D[p,q]
  DPA_CP_Alignment(sRow, sCol, p, q)                         | Recursion
  DPA_CP_Alignment(p, q, fRow, fCol)                         |

end.
```

Fig. 4. DPA with check pointing to determine alignment.

Recursion ceases when the region to be determined consists of 1 or 2 columns (the base case). If there is only 1 column, then the corresponding alignment is a run of zero or more deletes. If there are 2 columns then the alignment corresponds to zero or more deletes followed by a mismatch or insertion followed by zero or more deletes. These are simply determined from the two columns available in D.

An example is given in Fig. 3 using a cost of 1 for an insert, delete or mismatch and a cost of 0 for a match. This example aligns the sequences $As = \mathsf{CGCA}$ and $Bs = \mathsf{AAGT}$ to determine an optimal alignment, there are in fact 4 optimal alignments for this example (the arrows indicate the optimal alignment that will be found). The first step computes the whole D matrix (though only 2 columns are stored at any given time). The cell shown in bold is found to lie on the optimal alignment (i.e., $D[p, q]$ from above). This cell defines the two regions for recursion. The process continues until the region consist of 1 or 2 columns from which the alignment is determined directly.

It is important to note that in the example, we have shown the values in the D matrix to be consistent at each level of recursion. To do this the edit costs in

column q must be stored so they can be used at the next level of recursion. We refer to this storing of the contents of cells on a column as *check pointing* a column. Throughout this paper we will refer to this method as a check point method.

For the simple cost alignment with the DPA it is not strictly necessary to check point any columns at all, and it is essentially this variant that Hirschberg [9] attributes to Eppstein. It is possible to restart the top left cell of each region with an edit cost of 0. The more complex algorithms such as linear gap costs, or Ukkonen's algorithm do require this check pointing because the matrix cells contain important *state* information that is required to restart the algorithm at the next level of recursion. Thus for the DPA with simple costs no check pointing is required, however we will still refer to our modification of the DPA as the 'DPA with check pointing' simply as a consistent basis for referring to our method of modifying alignment algorithms.

The DPA with check pointing is shown in Fig. 4. It is worth noting a large portion of this algorithm is the same as for the standard DPA (see Fig. 2). The later half of the algorithm is to determine the split of the D matrix and to perform the recursion. This general form of the check pointing algorithm is maintained when applied to different cost functions (and even when applied to the Ukkonen algorithm, see Fig. 8). This is a major advantage of this check point method, it is to a large extent independent of the underlying algorithm.

The check pointing method has also been applied to a variation of Ukkonen's algorithm for three strings [1]. The check pointing method reduces the space complexity of this algorithm from $O(d^3)$ to $O(d^2)$. Applying the check point method is similar to the two string algorithm and relatively straightforward to apply.

It is easily seen that this algorithm only requires $O(n)$ space. At any point through the DPA only two arrays of size $O(n)$ are used. When the DPA loop finishes and the alignment cell is determined, the checkpoint information is no longer needed and the space can be re-used in the next recursive step.

The proof that this algorithm has time complexity $O(n^2)$ follows the same reasoning as for Hirschberg's. The area computed of the D matrix is halved at each step, plus an extra column. So the work done by the DPA with check pointing is $|D| * (1 + 1/2 + 1/4 + \cdots) + \log_2 |D|$, where $|D|$ is the size of the D matrix. Since $|D|$ is about n^2 the time complexity is $O(n^2)$. Fig. 5 shows the experimental results confirm this analysis.

The test data for all sample runs in this paper were generated randomly with an alphabet size of 26. First string *As* was generated randomly, then string *Bs* was generated from *As* with a fixed mutation probabilities of 0.2, 0.1 and 0.1 for change, insertion and deletion, respectively (mismatches to the same character were allowed).

The extra column (and thus the log term in the time complexity), is due to the check point algorithm as presented here determining a *cell* of the D matrix that lies on the optimal alignment which is used in both recursive sub-parts. Compare this to Hirschberg's algorithm which determines a *step between* two cells on the optimal alignment and therefore is able to exactly halve the D matrix. It is straightforward to modify the check point algorithm to behave like Hirschberg's in this respect, but the added complication gives little practical advantage.

2.2.1. Trading space for time

A major advantage of our check pointing method over Hirschberg's, is that in practice the check pointing algorithm can be sped up by keeping more than one check point for each run through the DPA matrix. The work for the next step is then reduced by the number of checkpoints kept. That is, if 2 columns are check pointed (at $|Bs|/3$ and $2|Bs|/3$), the area of the DPA matrix that needs to be recomputed is divided by 3 at each step. The time/space complexities are unchanged, but the constant in the running time can be reduced by increasing the constant in the space required. This is a nice feature because available memory can be traded for running time. The running time constant relative to the basic DPA is $\sum_{i=0}(x+1)^{-i}$ where x is the number of checkpoints.

3. DPA with linear gap costs

The simple costs used in the basic DPA are not as biologically plausible as linear gap costs when aligning DNA or protein sequences. Here a run of insertions or deletions is treated as a single event and

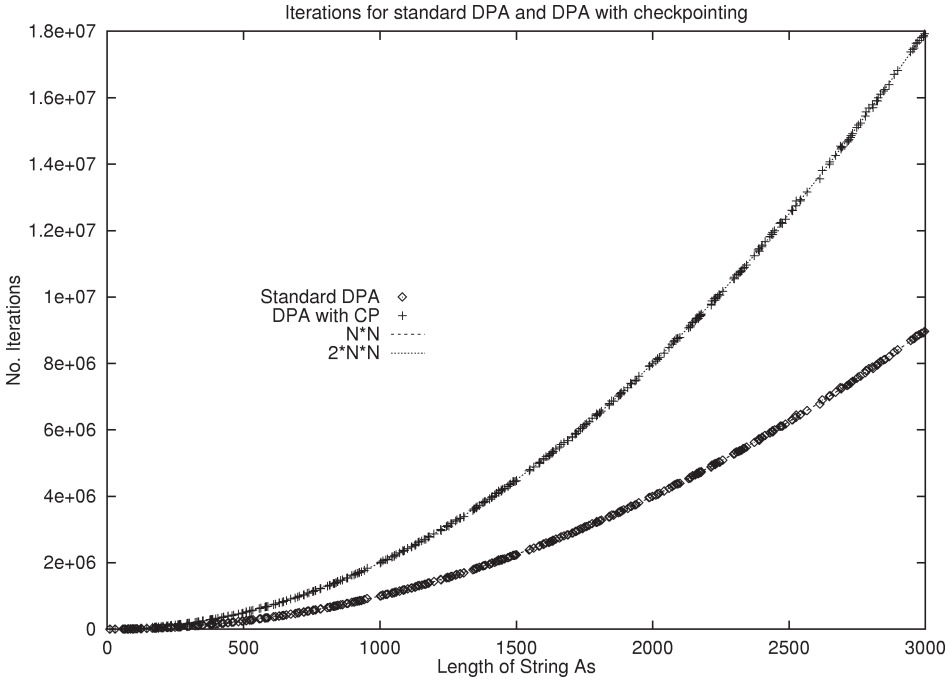

Fig. 5. Comparison of the number of iteration of the loop in the basic DPA against that of the DPA with check pointing. Both $N^2/2$ and n^2 are plotted. Note: the x-axis is the length of string *As* only, and while *Bs* will be of similar length it will differ slightly.

given a cost $a \cdot x + b$ where a and b are constants, and x the length of the run. Simple costs are a special case of linear costs with $a = 1$ and $b = 0$.

The basic DPA can be modified to compute an optimal alignment using a linear gap cost function (see [4]). The DPA for linear gap cost has in each cell of the DPA matrix 3 alignment costs—one each for the 3 possible *states* corresponding to the last step, mismatch, insert or delete. The linear gap cost DPA has time complexity $O(n^2)$ and space complexity $O(n^2)$, the same as for the basic DPA. Several methods have attempted to reduce the space complexity by constant factors see [16,19,3,5,6]. Myers and Miller [13] applied Hirschberg's technique to reduce the space complexity to $O(n)$, however, their method is more complicated than that presented here and our technique has the practical advantage of being able to trade space for time of execution.

The previously described check pointing method can just as easily be applied to the DPA with linear gap costs. There are two main differences, the first is that for it to be possible to restart the DPA on a region it is necessary to know the contents of the top left cell. This is necessary because of the 3-state nature of the alignment algorithm (it needs to be restarted with the correct state information). As the DPA is being run forward the check point column is saved (or check pointed), then when that DPA step has finished the contents of the cell on the optimal alignment, $D[p, q]$, is used to restart algorithm correctly.

The second difference is that it is necessary for each *state* to carry the column index of the cell it derived from on the check point column. That is, each cell of the D matrix will contain 3 edit costs and 3 column indices, one each for the three possible states (one state for each of the last possible operations, mismatch, insertion or deletion). The same time/space complexities apply as for the simple cost case (the column check pointing can be done in $O(1)$ time by using pointers to columns).

One advantage of the check pointing method described here over Hirschberg's method is that the check point algorithm for more complex cost functions (e.g., piecewise linear or concave costs) remains

essentially the same as for linear gap costs. That is, information needs to be carried forward about which check point cell lies on the optimal alignment, and a column of the D matrix needs to be saved so the algorithm can be restarted from the split cell. Whereas using Hirschberg's method it becomes much more difficult to find the split point.

4. Ukkonen's algorithm

Ukkonen [18] (and independently Myers [12]) presented an alignment algorithm that runs in $O(nd)$ time in the worst case and $O(n+d^2)$ on average (where n is the length of the strings, assumed to be of the same order, and d is the edit cost). This algorithm uses $O(d^2)$ space or if no alignment is required $O(d)$ space. A necessary condition for this algorithm is that all mutation costs are positive integers, and that a match costs 0. If however, the chosen costs do not meet these criteria it may be possible to choose costs that do meet the criteria and leave the optimal alignment unchanged (see [2]).

Ukkonen's algorithm speeds up the basic DPA by recognizing a number of facts about the DPA matrix: not all the entries of D are needed, the diagonals are monotonic non-decreasing, and only the end point of a run of matches is important. An alternative matrix U is used in Ukkonen's algorithm. Entry $U[ab, d]$ contains the maximum distance obtainable along string *As* for cost d on diagonal ab. A row of the U matrix corresponds to a diagonal of the D matrix, and a column of the U matrix to a "contour" of fixed cost in the D matrix. As an example assume DPA matrix cell $D[i, j]$ is on the optimal alignment, then in terms of the U matrix this cell will be on the diagonal $ab = i - j$, and thus $U[i - j, D[i, j]] = i$. Ukkonen's algorithm is given in Fig. 6.

The outer loop of Ukkonen's algorithm loops over each entry in the U matrix determining for diagonal ab and cost d how far along string *As* can be reached. This is determined by looking at U with a cost $d-1$ on the same diagonal, ab, and the two neighboring diagonals, $ab+1$ and $ab-1$. The inner loop then extends this distance while strings *As* and *Bs* match (corresponding to a run of matches down a diagonal of the D matrix). As with the basic DPA, the alignment is obtained by

```
{U[ab,d] = max a s.t. D[a,b] = d
             where ab = a-b
         = -infinity if no such a exists}

U[0,0]  = max a s.t. As[1..a] = Bs[1..a]
U[ab,d] = -infinity, if |ab|>d

{ Outer loop, iterated until
    U[|As|-|Bs|,d] = |As| }
  U[ab,d] = max(U[ab+1, d-insertCost],
                U[ab,   d-mismatchCost]+1,
                U[ab-1, d-deleteCost]+1)

{ Inner Loop, extends diagonal
    on a run of matches }
  while
    ( As[U[ab,d]+1] = Bs[U[ab,d]-ab+1] )
    U[ab,d] += 1
```

Fig. 6. Ukkonen's algorithm for simple mutation costs.

tracing back through the choices made in the max() function when calculating the U matrix.

An example of the U matrix is shown in Fig. 7 for the same example as shown for the DPA in Fig. 3. In this figure, the U matrix has for diagonal $ab = -1$ and cost 2 the contents 2 (i.e., $U[-1, 2] = 2$), this means that for a cost of 2 on diagonal $ab = -1$ the furthest that can be reached on *As* is the second character. In terms of the D matrix this corresponds to the cell with row 2 and column $2 - (-1) = 3$ which (from Fig. 3) contains the cost 2.

Thus in terms of the D matrix, Ukkonen's algorithm calculates the entries in a region around the final diagonal that has a width equal to the edit distance of the two strings. However "holes" are left in the D matrix for each run of matches. For a brief discussion of the complexity of this algorithm see Section 4.2.

Each column of the U matrix is completely defined within terms of the previous column (for simple costs), hence it is possible to calculate the edit cost by using only $O(d)$ space (the whole U matrix is required if the alignment is desired). This is simply done by changing all cost indexing of the U matrix to be computed modulo 2 (i.e., $U[ab, d] \rightarrow U[ab, d \bmod 2]$). This is identical to reducing DPA to $O(n)$ space when the alignment is not wanted (see start of Section 2).

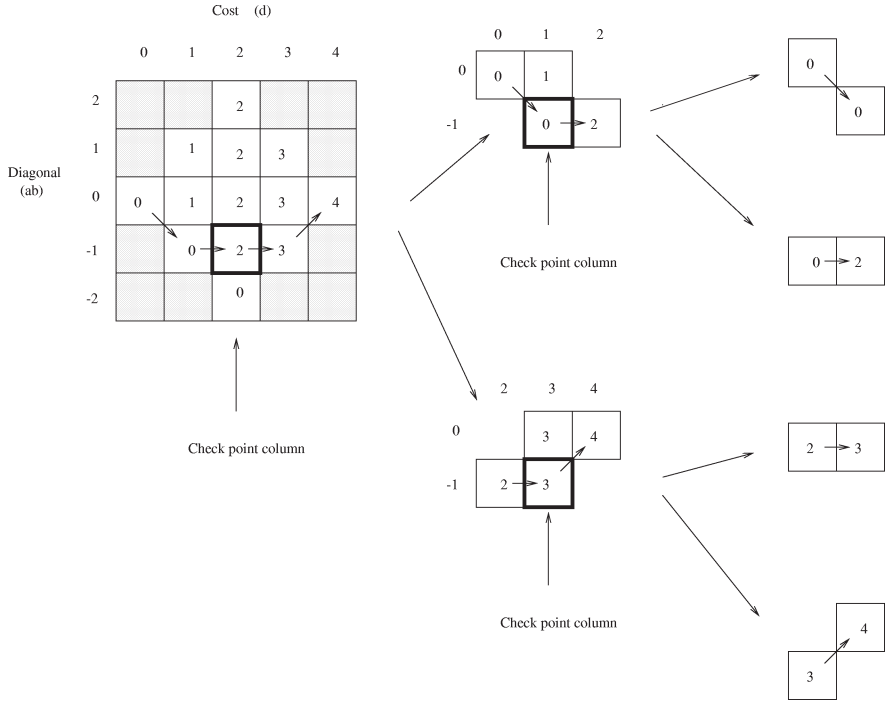

Fig. 7. An example of check pointing with Ukkonen's algorithm for sequences CGCA and AAGT, using insertion, deletion and mismatch costs of 1 and a match cost of 0.

4.1. Ukkonen's algorithm in linear space

Myers [12] applied Hirschberg's technique to his version of Ukkonen's alignment algorithm to reduce the space required to determine an optimal alignment from $O(d^2)$ to $O(d)$. This maintains the worst case time complexity of $O(nd)$, but does not keep the average complexity at $O(n + d^2)$; in fact Myers does not discuss the expected time complexity of his linear space algorithm. The reason is that the work is not distributed evenly over the Ukkonen matrix, making it impossible to split the work evenly into two halves. The technique we describe here to reduce the space complexity of Ukkonen's algorithm to linear, has the same advantages over Myers [12] as our check pointing DPA has over Hirschberg [8]. It is simpler, especially for more complex cost functions because it is not necessary to run the algorithm in reverse (as it is with Hirschberg's). The other advantage is that there is an easy practical trade off between the running time and the space overhead.

The check pointing technique used on the DPA (on the D matrix) can be adapted to be used on the U matrix with Ukkonen's algorithm. The outer loop of the Ukkonen algorithm works along the columns of the U matrix. The column with index $d/2$ is check pointed, and a cell on this column that lies on an optimal alignment is determined. This is done as with the DPA version by having each cell carry extra information forward about which cell on the check point column it derives from. Knowing a cell on the optimal alignment allows the U matrix to be split into two smaller regions. Recursion is then used on these two smaller regions until the complete alignment is determined.

As with the linear gap cost version of the DPA with check pointing it is necessary to store the check point column of the U matrix. This is because the contents of the split cell (i.e., the distance along the *As* string), is needed for recursion on the second half of the U matrix.

```
Procedure CP_UKK(sDiag, sCost, sDist, fDiag, fCost)
    if (sCost = fCost) then return                                  | First base case

    U[sDiag, sCost mod 2] = sDist                                   | Starting distance
    q = (fCost+sCost)/2                                             | Cost to check point at

    { General step, iterated from sCost until U[fDiag, fCost] }     |
        U[ab,d] = max(U[ab+1, (d-1) mod 2],                         |  Standard Ukkonen
                      U[ab,   (d-1) mod 2]+1,                       |
                      U[ab-1, (d-1) mod 2]+1)                       |
                                                                    |
        while (As[U[ab, d]+1] = Bs[U[ab, d] - ab + 1])              |
            U[ab,d] += 1                                            |


        if (d = q)                                                  |
            CP[ab] = U[ab, d mod 2]                                 | Set up check point
            f[ab, d mod 2] = ab                                     |
        elseif (d > q)                                              |
            f[ab, d mod 2] = f[(ab,d) chosen in max()]              | Carry check point information
        endif                                                       |
    {end general loop}

    if (sCost+1 >= fCost) then                                      | Second base case
        {Determine alignment directly from U}                       |
        return                                                      |
    endif                                                           |

    p = f[fDiag, fCost mod 2]                                       | Now have U[p,q]
    CP_UKK(sDiag,  sCost,  sDist,     p , q)                        | Recursion
    CP_UKK(p, q, f[p], fDiag,  fCost)                               |
end.

CP_UKK(0,0,0, |As| - |Bs|, edit_cost)
```

Fig. 8. Ukkonen's algorithm with check pointing using simple mutation costs.

An example of this is shown in Fig. 7. Note that this example is for the same sequences as used in Fig. 3. For a given cell, $U[ab, c]$, of the Ukkonen matrix, the corresponding cell of the DPA matrix is $D[U[ab, c], U[ab, c] - ab]$. The cell on the check point column known to lie on the optimal alignment is shown in bold at all steps.

The first time through the U matrix, the edit cost d, is unknown, so it is not possible to know which column is the middle column. Hence, the first time through, the column at cost $d/2$ cannot be check pointed. The simplest solution is to first run the cost only version of Ukkonen's algorithm to determine the edit cost, then run the check point algorithm to determine the alignment. While this does work, it is not optimal. A better solution, is to check point the column when half of the string *As* has been processed (i.e., at $n/2$). This serves as a fair approximation to $d/2$ assuming the mutations are evenly distributed along the strings. Note that this choice on the first pass of the algorithm does not change the time complexity, it simply improves the constant.

A complication with adding the check pointing technique to the Ukkonen algorithm arises when some mutation costs are > 1. This is because each column of the U matrix is no longer simply defined in terms of the previous column, but in terms of the previous x columns, where x is the maximum mutation cost. This

is overcome by check pointing x consecutive columns rather than a single column.

Fig. 8 shows Ukkonen's algorithm with check pointing added. This is for simple mutation costs where all mutations have cost 1. For more complex costs both the modulo size and the number of columns check pointed must be increased to the maximum mutation cost. It is worth comparing this algorithm to the DPA with check pointing (Fig. 4) since they both have a very similar structure. Similar check pointing steps are added around the standard algorithm in both cases. This similarity when the check pointing method is applied to different algorithms and even different cost functions is a major advantage of this technique.

4.2. *Complexity of Ukkonen's algorithm with and without check pointing*

The worst case complexity of Ukkonen's algorithm is $O(nd)$ and corresponds to the largest area of the D matrix equivalent to that calculated in the U matrix. This area in the D matrix is of length n, along the final diagonal, and of width d around that diagonal. The expected time complexity is $O(n + d^2)$ which is almost always achieved. A brief explanation of this expected time complexity follows (for a fuller explanation see [12]).

- The number of iterations of the outer loop is $d^2/2$.
- The inner loop is iterated $L = n - d$ times for the optimal alignment. If all matches off the optimal alignment are assumed to be coincidental, the expected length of such a coincidental match is $1/(\Sigma - 1)$ (where Σ is the alphabet size). There are $\sim d^2/2$ positions in the matrix where such matches can occur. So expected number of inner loop iterations

$$= L + \frac{d^2}{(2(\Sigma - 1))}.$$

If we now consider the check pointing algorithm, where the edit distance is already known so the U matrix can be split at $d/2$ we obtain the following:

- Outer loop iterations

$$= \underbrace{\frac{d^2+1}{2} + \frac{d^2+2}{4} + \frac{d^2+6}{8} + \cdots}_{\sim \log_2 d \text{ terms}}$$

$$\approx d^2 + \log_2 d \approx d^2.$$

- Inner loop iterations

$$\approx L(1 + \log_2 d) + \frac{d^2}{2(\Sigma - 1)}\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right).$$

Figs. 9 and 10 show plots for experimental data. These plots of the check pointing algorithm loop counts do not take into account first working out the edit distance.

The expected time complexity of Ukkonen's algorithm with check pointing to determine the alignment is thus $O(n \log_2 d + d^2)$; in practice the d^2 term dominates. The Ukkonen algorithm with check pointing implemented for this paper (not particularly optimized) was found to be typically 3.5 times as slow as the standard Ukkonen algorithm (see Fig. 11). Run times of the check point algorithm included running the cost only version of the Ukkonen algorithm first, followed by Ukkonen's algorithm with check pointing to determine the alignment. It is expected that if the first iteration through the U matrix had check pointed at $n/2$, as previously suggested, it would have been only 2.5 times as slow as the standard Ukkonen algorithm.

As with the DPA version of the check pointing algorithm, it is possible to trade running time for space overhead by keeping more than one check point. This has the effect of modifying time/space complexity constants (see Section 2.2.1).

5. Ukkonen's algorithm with linear gap costs

Ukkonen's algorithm has been applied to aligning strings using linear gap costs [20]. The check pointing method can also be applied to this algorithm so an alignment can be determined in linear space. Little change needs to be made to the check pointing algorithm of the previous section to produce alignments using linear gap costs. Linear gap costs are charged as $a + bx$ where x is the length of a gap. As mentioned in Section 4.1 the number of columns of the U matrix that must be check pointed is equal to the maximum mutation cost. Thus for linear gap costs the number of consecutive columns to be check pointed is $a + b$ (provided this is greater than the mismatch cost, which is typically the case).

As with the check pointing DPA for linear costs, each cell of the matrix contains 3 distances, one each

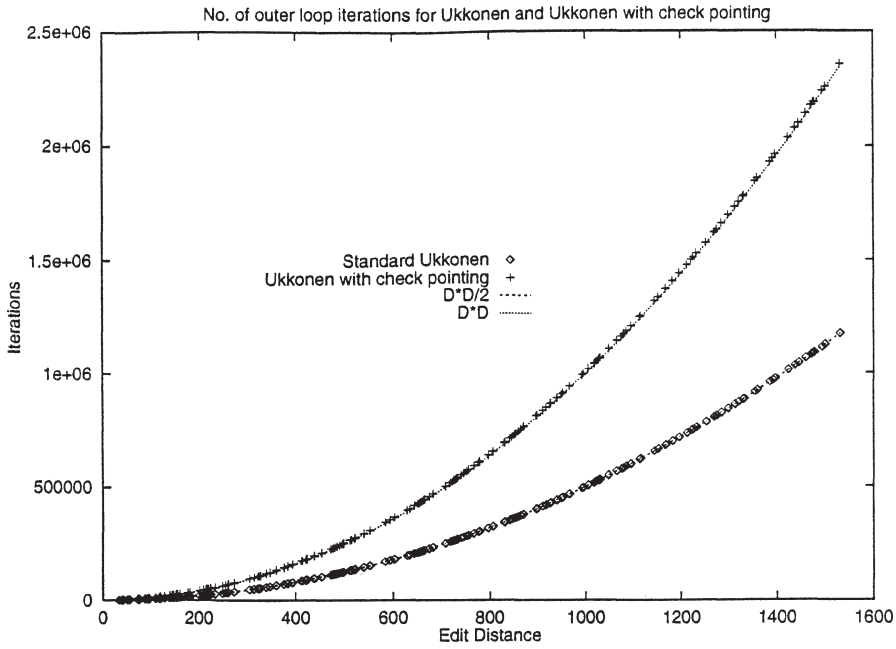

Fig. 9. Comparison of iterations of the outer loop in Ukkonen's algorithm with and without check pointing. String *A* was generated with length 5000, string *B* was mutated from this so will be of similar length, but slightly different.

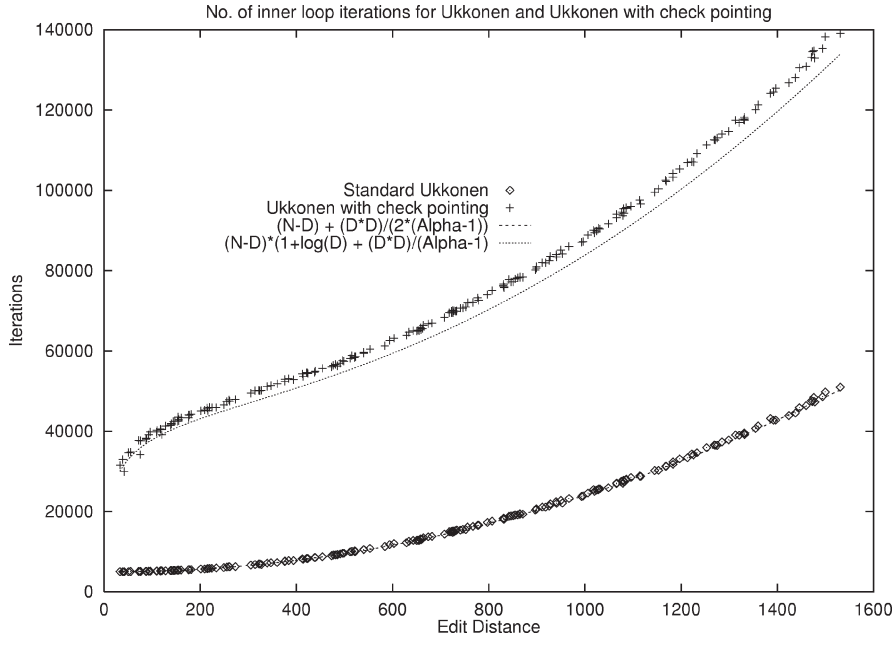

Fig. 10. Comparison of iterations of the inner loop in Ukkonen's algorithm with and without check pointing. String *A* was generated with length 5000, string *B* was mutated from this so will be of similar length, but slightly different.

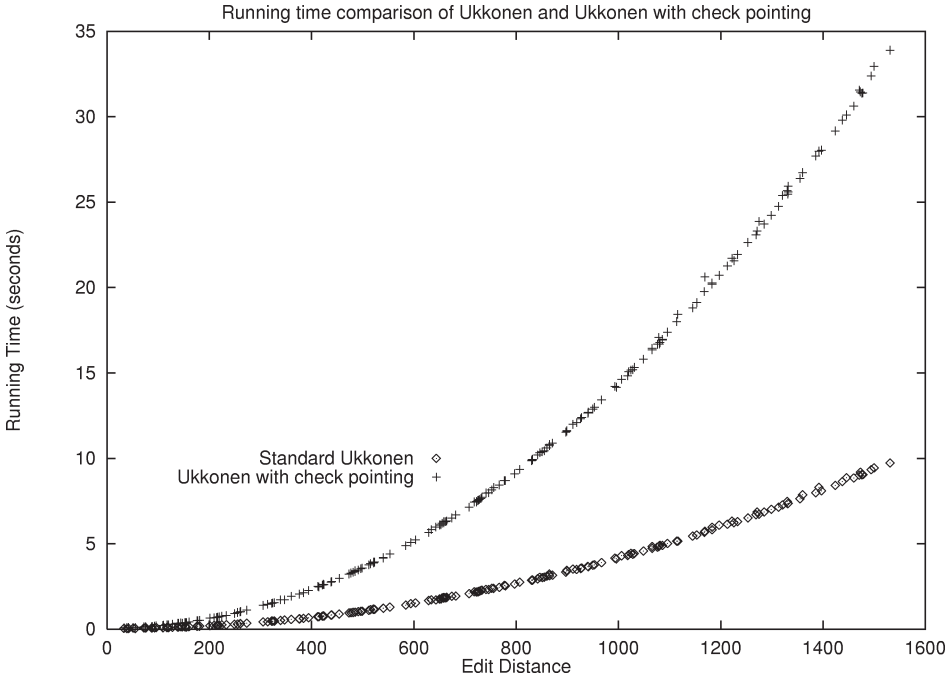

Fig. 11. Actual running time of standard Ukkonen algorithm against Ukkonen's algorithm with check pointing. String *A*s was generated with length 5000, string *B*s was mutated from this so will be of similar length, but slightly different. (Timings done on a Pentium 100 with 16 Mbytes RAM running Linux 2.0.0.)

for the three possible step directions into the cell. Thus the check pointed column must also store 3 distances in each cell. Also, since each cell of the U matrix contains 3 distances, it is necessary to carry forward for each one, the row index of the cell it derived from on the check point column. This is almost identical to the modifications necessary for the linear gap cost DPA in Section 3.

Apart from these differences the check pointing algorithm on the Ukkonen matrix with linear gap costs is the same as that for simple mutation costs.

This is a considerable advantage over the method employed by Myers [12], although Myers investigated only simple mutation costs. The check pointing method becomes no more complicated with more complex mutation costs.

6. Conclusion

We have presented a check pointing technique to modify alignment algorithms to produce an alignment in linear space with little or no effect on the time complexity. This method had been previously applied to the simple DPA with simple costs. We showed the advantages of this method over other methods. It is easier to implement especially for more complex cost functions, and the check point algorithm is very similar even when applied to different alignment algorithms. There is also the added practical bonus of being able to run the program faster by using more check points. The check pointing technique when applied to Ukkonen's algorithm produces an alignment algorithm that uses $O(d)$ space and runs in $O(n \log_2 d + d^2)$ time on the average.

References

[1] L. Allison, A fast algorithm for the optimal alignment of three strings, J. Theoret. Biol. 164 (1993) 261–269.
[2] L. Allison, Normalization of affine gap costs used in optimal sequence alignment, J. Theoret. Biol. 161 (1993) 263–269.
[3] S. Altschul, B. Erickson, Optimal sequence alignments using affine gap costs, Bull. Math. Biol. 48 (1986) 606–616.

[4] O. Gotoh, An improved algorithm for matching biological sequences, J. Molecular Biol. 162 (1982).
[5] O. Gotoh, Alignment of three biological sequences with an efficient traceback procedure, J. Theoret. Biol. 121 (1986) 327–337.
[6] O. Gotoh, Pattern matching of biological sequences with limited storage, CABIOS 3 (1987) 17–20.
[7] J. Grice, R. Hughey, D. Speck, Reduced space sequence alignment, CABIOS 13 (1) (1997) 45–53.
[8] D. Hirschberg, A linear space algorithm for computing maximal common subsequences, Comm. ACM 18 (6) (1975) 341–343.
[9] D. Hirschberg, Serial computations of Levenshtein distances, in: A. Apostolico, Z. Galil (Eds.), Pattern Matching Algorithms, Oxford University Press, 1997, pp. 123–141.
[10] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, Soviet Phys. Dokl. 10 (8) (1966) 707–710.
[11] W. Miller, E.W. Myers, Sequence comparison with concave weighting functions, Bull. Math. Biol. 50 (2) (1988) 97–120.
[12] E.W. Myers, An O(nd) difference algorithm and its variations, Algorithmica 1 (1986) 251–266.
[13] E.W. Myers, W. Miller, Optimal alignments in linear space, CABIOS 4 (1) (1988) 11–17.
[14] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Molecular Biol. 48 (1970) 443–453.
[15] P.H. Sellers, On the theory and computation of evolutionary distances, SIAM J. Appl. Math. 26 (4) (1974) 787–793.
[16] P. Taylor, A fast homology program for aligning biological sequences, Nucleic Acids Res. 12 (1984) 447–455.
[17] W.R. Taylor, C.A. Orengo, Protein structure alignment, J. Theoret. Biol. 208 (1989) 1–22.
[18] E. Ukkonen, On approximate string matching, Found. Comput. Theory 158 (1983) 487–495.
[19] K. Watanabe, Y. Urano, T. Tamaoki, Optimal alignments of biological sequences on a microcomputer, CABIOS 1 (1985) 83–87.
[20] C.N. Yee, L. Allison, Fast string alignment with linear indel costs, Technical Report 92/165, Monash University, Department of Computer Science, Clayton, Vic., Australia, 1992.