Algorithms for Sequence Alignment

David Richard Powell

B.Sc (Hons)

School of Computer Science and Software Engineering

Monash University

Australia.

Submitted for the degree of Doctor of Philosophy

August 2001

Abstract

Sequence alignment is an important tool for describing relationships between sequences. Many sequence alignment algorithms exist, differing in efficiency, and in their models of the sequences and of the relationship between sequences. The focus of this thesis is on algorithms for the optimal alignment of two or three sequences of biological data, particularly DNA sequences. The algorithms are discussed with particular emphasis on space and time complexity.

A divide-and-conquer method is presented for use with a number of different alignment algorithms. This method may be used to reduce the space complexity of an alignment algorithm with little or no effect to the time complexity. The advantages of this divide-and-conquer method include its simplicity and the ease with which it can be applied to many different alignment algorithms. These advantages are demonstrated by using the divide-and-conquer method in conjunction with several known alignment algorithms.

An efficient alignment algorithm is presented for the important problem of optimally aligning three sequences using a linear function for costing gaps in the alignment. For sequences of length n, and a minimum edit cost of d, this new algorithm has a time complexity of $O(d^3 + n)$. The algorithm is further developed by using the aforementioned divide-and-conquer method to improve its space complexity. This combination results in a time and space efficient algorithm, while also illustrating the usefulness of the divide-and-conquer method.

It is important when aligning sequences to correctly account for any non-randomness that is significant in the sequences. For example, if certain statistical patterns appear throughout sequences from a certain family, it is important to make use of this information when aligning sequences from this family. Common, unsurprising, patterns provide less evidence for the relatedness of sequences than more surprising regions provide. A new algorithm is presented to align optimally two non-random sequences. For a particular sequence model, this new algorithm apportions weight to every part of the alignment dependent on the importance of that part as determined by the sequence model. This algorithm is then developed further so that it can be used to infer whether two non-random sequences are related.

Declaration

This thesis contains no material that has been accepted for the award of any other degree in any university or other institution. To the best of my knowledge, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

David R. Powell

August 2001

Acknowledgements

I would like to thank my supervisors, Trevor Dix and Lloyd Allison, both of whom gave important feedback after assiduously proofreading this thesis. I have always been amazed at their willingness to give their time to assist me. They have given useful advice throughout my candidature on topics both related and unrelated to research. The many interesting discussions we had were both productive and enjoyable. They have shown me more patience than I deserved.

I would also like to thank the members of the School of Computer Science and Software Engineering for providing an enjoyable atmosphere in which work. My time at Monash would not have been the same without this group of friendly and talented people.

It has been a pleasure to share an office with Torsten Seemann. I deeply indebted to him for the help he has given me by always being willing to listen. My thanks also go to him for his effort in proofreading this manuscript.

Finally, I would like thank my family. Without their continual love and support this thesis would never have been written.

Contents

Listings

List of Figures

List of Tables

Chapter 1

Introduction

Sequence alignment is important in many different areas as a method to infer how two or more sequences are related. Areas as diverse as computer vision, speech recognition, evolutionary biology and DNA sequencing have seen the use of sequence alignment algorithms. This thesis focuses on the application to biological data, in particular nucleotide sequences. However, it is possible to apply the algorithms presented to different areas with little or no change. The main focus of research in this thesis is the computational aspect of the algorithms. A method to reduce the space complexity of standard alignment algorithms is developed. A fast algorithm for alignment of three sequences using linear gap costs is developed. This algorithm is then extended to have a lower space complexity while still finding the same optimal alignment. Also developed is an algorithm for aligning sequences that have low complexity regions by correctly accounting for the information contributed by each part of the sequences.

Much DNA sequence alignment is done using a dynamic programming algorithm (DPA) first presented by Levenshtein [33] and independently, but later, by Sellers [52]. This DPA was first applied to biological sequences by Needleman and Wunsch [40] and since has been used extensively. Gotoh [20] presented a modified version of the DPA which uses a better model of mutation for biological data by giving a gap in the alignment a cost based on a linear function of the gap length. For sequences of length n all these algorithms require $O(n^2)$ time and space.

Hirschberg [27] improved the standard DPA using a divide-and-conquer method which requires only $O(n)$ space while still running in $O(n^2)$ time. Myers and Miller [38] applied Hirschberg's method to Gotoh's linear gap cost algorithm to achieve the same improved space complexity.

Ukkonen [62, 63] and independently Myers [37] discovered redundancies in the standard

DPA which they exploited to speed it up. There is a limitation to Ukkonen's algorithm: matches must cost 0, and other mutation costs must be small positive integers. For sequences of length $O(n)$ with an alignment cost of d, Ukkonen's algorithm requires $O(nd)$ time in the worst case and $O(d^2 + n)$ on average. The space complexity is $O(d^2)$. This algorithm runs much faster than the standard DPA especially, for similar sequences.

An alternative divide-and-conquer method has been briefly described by Hirschberg [28], who attributed it to Eppstein (unpublished). Hirschberg describes how to apply this method to the standard DPA. Chapter 2 explains this divide-and-conquer method and how it can be applied simply to the standard DPA. In Chapter 2, this divide-and-conquer method is then developed further so it can be applied to Gotoh's linear gap cost algorithm. A major contribution of Chapter 2 is the inclusion of the divide-and-conquer method into both Ukkonen's algorithm and Ukkonen's algorithm with linear gap costs to reduce the required space to $O(d)$. The superiority of this divide-and-conquer method over Hirschberg's [27] method lies in its simplicity, especially when applied to the more complex algorithms and cost functions.

Three sequence alignment can be useful in many situations. For example, alignments of three sequences provide more information than alignments of two sequences. Also, three sequence alignment is useful for inference of evolutionary trees from sequence data. The three sequence version of the standard DPA is a straightforward extension of the two sequence version. Allison [2] published a three sequence alignment algorithm extending Ukkonen's two sequence algorithm. Allison's algorithm requires $O(n + d^3)$ time on average and $O(d^3)$ space. Gotoh [21] presented a three sequence linear gap cost alignment algorithm which requires $O(n^3)$ time and space. In Chapter 3, an improved version of Gotoh's three sequence, linear gap cost algorithm is developed. Further, an algorithm for the same problem, three sequences and linear gap costs, is developed based on Ukkonen's speed-up. This algorithm requires $O(n + d^3)$ time on average and $O(d^3)$ space.

Chapter 4 shows how the divide-and-conquer method of Chapter 2 can be applied to the new three sequence linear gap cost, Ukkonen based algorithm of Chapter 3. The time complexity is argued to be $O(n + d^3)$ on average and the space used $O(d^2)$. The complications of memory allocation for this complex algorithm are also discussed.

The alignment of two sequences may be viewed as aligning two mutated copies of an unknown parent sequence. With this view in mind, all the alignment algorithms discussed so far consider all possible parent sequences, of the same length, equally likely *a priori*. If, however, something is known about the parent sequence, such that it fits a non-uniform model, the choice of optimal alignment may be different. Chapter 5 presents an algorithm for aligning two sequences which are *noisy* copies of a non-random parent sequence. It is shown how to infer the parent sequence from the sequence model and the two known child sequences. The significance of the alignment is judged by comparing it to a null model. It is also shown that the alignments produced by this algorithm are *better* than the alignments produced by the standard DPA for sequences that fit the defined sequence model.

The focus of this thesis is on the computational aspect of alignment algorithms especially for DNA sequences. The new divide-and-conquer method developed in this thesis is quite general and could be applied to the alignment problem for many different types of sequence data. The fast, three-sequence alignment algorithm with linear gap costs is particularly useful for small alphabet sequences such as DNA. The new low information alignment algorithm was inspired by the problem of aligning DNA sequences that have repetitive or low information regions.

The remainder of this chapter is gives an introduction to previously known alignment algorithms that form the background of this thesis. A taxonomy is provided at the end of the chapter which classifies both the known and the new alignment algorithms by their characteristics. This helps to illustrate how the new algorithms and methods developed here relate to other work in the area.

1.1 Sequence Alignment

Alignment of sequences can be an important tool to measure the similarity of two or more sequences. An alignment can also be used to express how sequences are related. Sequence alignment has been used in many diverse areas such as stereo image matching [13], speech recognition [16], spelling correction, gas chromatography [47] and hand writing recognition [18, 10]. Sequence alignment has become an important tool for processing many different types of biological data, such as DNA sequence assembly [54, 55, 31], protein sequence alignment [40], construction of evolutionary trees [29] and comparison of protein secondary structure [59, 60, 30].

Aligning a pair of sequences involves matching characters from the two sequences either with each other or the null character '-', to indicate an insertion or deletion. Insertions and deletions are sometimes referred to as *indels*, or as *gaps*. The following is an example of aligning the sequences `ATACTAGA` and `AACTTGGA`.

```
A T A C T A G - A
|   | | |   |   |
A - A C T T G G A
```

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

It is important to come up with a measure of the goodness, or *scoring function*, for an alignment. Sometimes a *cost function* is used instead of a scoring function. A high *cost* implies a poor alignment, while a high *score* implies a good alignment. Scoring functions, and cost functions may be considered equivalent since one may be treated as the negative of the other. A cost can be given to the whole alignment by treating the alignment a position at a time and assigning a cost to each position based on the characters aligned. The cost of an alignment is called the *edit cost*. A cost function that gives a fixed cost to each type of point mutation, insert, delete or mismatch, shall be referred to as *simple* costs. Levenshtein [33] presented an instance of simple costs that gives a cost of 0 to a match and a cost of 1 to each of the possible point mutations; insertion, deletion or mismatch. These costs shall be referred to as *Levenshtein costs* and the resulting cost of the alignment as the *Levenshtein distance*. Therefore the Levenshtein distance of the above alignment would be 3.

A problem closely related to the alignment problem is the Longest Common Subsequence (LCS) problem. An LCS is a subsequence of maximal length common to two or more sequences. In the above example, the LCS would be `AACTGA` having a length of 6. The length of the LCS and the Levenshtein distance of two sequences of lengths m and n are related in a simple manner. The Levenshtein distance is the number of point mutations in an alignment, and the length of the LCS is the number of matches in the alignment. So, if we take $m + n$, the total number of characters in the two sequence, and subtract $2|lcs|$, the number of these characters that are matches, the remainder is the sum of the number of insertion, the number of deletions and twice the number of mismatches in the alignment. Thus, using mis to denote the number of mismatches, the following relation holds:

$$\text{Levenshtein distance} = m + n - 2|lcs| - mis$$

Other cost functions are often employed in sequence alignment. Linear, also called affine, gap costs are often used when aligning biological sequences. Linear gap costs give a cost to a run of k insertions or deletions of $a + b \times k$ for some fixed values of a and b. Having $a > b$ has the effect of favoring fewer long gaps over many short gaps. Section 3.3 contains further discussion on linear gap costs. Some of the other cost functions used in sequence alignment are piecewise linear gap costs [20] and concave gap costs [34].

In some situations it is desirable to assign a cost in the alignment based on the actual characters being aligned. For example, if alignment were being used to find possible corrections to mistyped words in a word processor, it would be beneficial to have the cost of aligning a 'U' with an 'I' lower than a 'U' with a 'B', since 'U' is close to 'I' on the 'QWERTY' keyboard. In protein alignment algorithms amino acid substitution matrices are often used to give the cost of aligning any two amino acids. Originally the PAM [14, 15] matrices were used, but the more recent BLOSUM [24] matrices are now recommended.

1.2 Finding an Optimal Alignment

Once it is decided how to assign a cost to an alignment, the next problem is how to find an alignment with minimal cost, also known as an *optimal alignment*. The algorithm to find an optimal alignment depends upon the chosen cost function. In the next two sections, algorithms to find an optimal alignment under simple costs and then under linear gap costs shall be discussed. In the sections following these, optimal alignment algorithms that have better space and/or time complexity shall be shown.

1.2.1 The Basic DPA

The best known algorithm for finding an optimal alignment is a dynamic programming algorithm. Hereafter this algorithm for simple costs shall be referred to as the *standard* or *basic* DPA, or the *DPA2s* algorithm denoting the DPA on two sequences with simple costs.

Listing 1.1 shows the basic DPA for aligning two sequences As and Bs. The D matrix is the basis for the calculation. The matrix entry $D[i, j]$ contains the edit cost for aligning sequences $As[1..i]$ and $Bs[1..j]$. This algorithm allows for any simple costs subject to the following constraints:

1. $\forall x, y \quad d(x, y) \geq 0$
2. $\forall x, y \quad d(x, y) = d(y, x)$
3. $\forall x, y, z \quad d(x, y) \leq d(x, z) + d(z, y)$

Setting the costs as follows: matchCost=0, changeCost=insertCost=deleteCost=1, meets the above requirements and corresponds to calculating the Levenshtein distance. The algorithm shown in Listing 1.1 calculates the edit cost.

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

for i = 1..|As|
  for j = 1..|Bs|
    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
                                  changeCost))
  endfor
endfor
editCost = D[|As|,|Bs|]
```

Listing 1.1: The basic DPA or DPA2s algorithm.

Figure 1.1 shows an example of the DPA matrix after aligning the sequence `ACCGGTCGGC` and `TGGTCGCCC` with Levenshtein costs. The arrows indicate one (of the nine possible) optimal alignments which is also given below. The edit cost of this alignment is 5 as seen in the bottom right cell of the DPA matrix. The shaded cells will be discussed in Section 1.3.

```
A  C  C  G  G  T  C  G  G  C  –
         |  |  |  |  |     |
T  –  –  G  G  T  C  G  C  C  C
```

The algorithm as given in Listing 1.1 computes the DPA matrix, but does not return an actual alignment only the edit cost of an optimal alignment. If an optimal alignment is desired then the choices that were made in the $min()$ function must be traced back. In Figure 1.1 this corresponds to following the arrows backwards from the bottom right cell to the top left cell of the matrix. This part of the DPA computation is called the *back-trace* step. For sequences of length n, the calculation of the edit cost uses $O(n^2)$ time and space and the back-trace $O(n)$.

It is easily seen that the basic DPA calculates the rectangular D matrix, one row (or column) at a time. At each point of the calculation only the previous row of the matrix is required to calculate the next row. Thus, if only the edit cost is desired, and not the actual alignment, then the matrix can be reduced to just two rows. All row accesses are modified to be computed modulo 2, that is, $D[i, j] \to D[i \bmod 2, j]$. This reduces the space complexity of the algorithm to $O(n)$ at the cost of being unable to recover the alignment. A method for recovering the alignment while still using $O(n)$ space shall be discussed in Section 1.2.3.

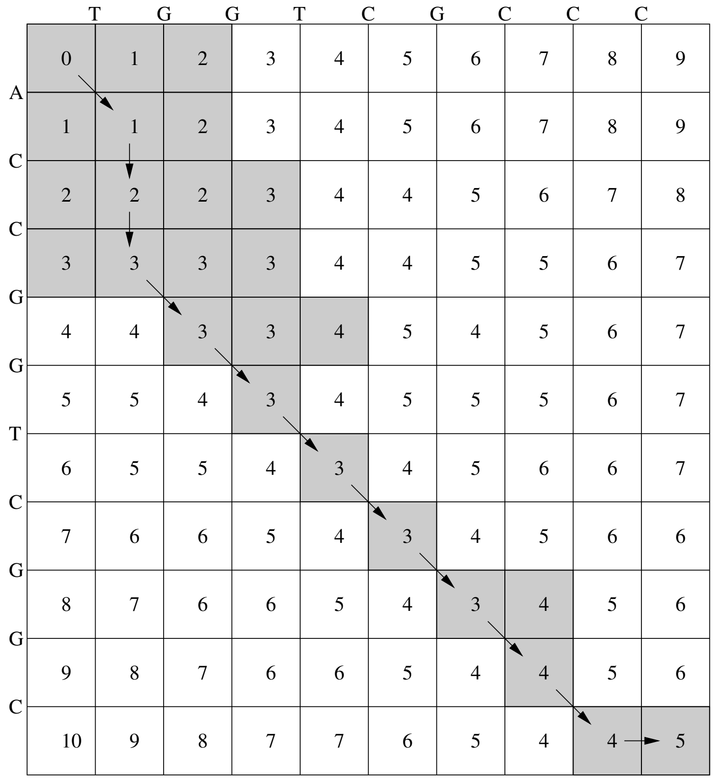

Figure 1.1: An example of the DPA matrix after aligning the sequences ACCGGTCGGC and TGGTCGCCC.

1.2.2 The DPA for Linear Gap Costs

In many situations, the use of simple costs produces undesirable alignments. Often it is more desirable to have a small number of long gaps than a large number of small gaps. To this end, simple costs can be generalised to *linear gap costs*. Linear gap costs give a cost to run of indels, a gap, based on a linear function. So, the cost of a gap of length k is $a + b \times k$, where a and b are positive numbers. Generally, the value for a, which is the cost to start a gap, is higher than the value for b, which is the cost to continue a gap. Under linear gap costs, matches and changes are treated as they were under simple costs. Linear gap costs reduce to simple costs if a is set to zero. As an example, consider the sequences `ACGGCTGGAAGTTAC` and `ACGGTAAC`. An optimal alignment of these sequences is shown below for Levenshtein costs, and then for linear gap costs.

A	C	G	G	C	T	G	G	A	A	G	T	T	A	C
|	|	|	|		|			|	|					|
A	C	G	G	–	T	–	–	A	A	–	–	–	–	C

An optimal alignment using Levenshtein costs.

A	C	G	G	C	T	G	G	A	A	G	T	T	A	C
|	|	|	|								|		|	|
A	C	G	G	–	–	–	–	–	–	–	T	A	A	C

An optimal alignment using linear gap costs.

Under Levenshtein costs, there are three optimal alignments which have the minimum edit cost of 7. The costs used for the linear gap costs alignment were: 0 for match, 1 for a change, 1 for the parameter a, and 3 for the parameter b. Using these values, the linear gap cost alignment is the only optimal alignment and has an edit cost of 11. There are many applications for which the optimal alignment using linear gap costs is the better alignment. These applications tend to model the insertion or deletion of a number of characters at a time as a single mutation event. Whereas, simple costs model every insertion, deletion or change as a separate mutation event.

The basic DPA can be modified to compute an optimal alignment using a linear gap cost function [20]. When using linear gap costs for calculating the cost of an alignment, an alignment pair at a time, it is important whether the previous alignment pair was an indel or not. Thus the DPA for finding an optimal alignment with linear gap costs is similar to the basic DPA but with information about the last operation carried into each cell of the DPA matrix.

Listing 1.2 shows the DPA for linear gap costs, where the cost of a gap of length k is $startGap + contGap \times k$. This algorithm shall be referred to as the DPA2l algorithm, short for the DPA style on two sequences using linear gap costs. The main difference from the basic DPA is that each cell of the matrix contains three edit costs, one for each of the last possible operations: insertion, deletion or match/mismatch. Like the basic DPA the alignment is obtained by tracing back through the matrix. The basic DPA can be seen as a special case of this algorithm. By setting startGap=0 in the linear gap cost DPA, a simple cost alignment algorithm is obtained.

The time and space complexity is $O(n^2)$, the same as for the basic DPA. If only the edit cost is required, the space complexity can be reduced to $O(n)$ in the same way as the basic DPA.

1.2.3 Alignment in $O(n)$ Space

Hirschberg [27] described an algorithm to calculate 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 alignment problem, so for consistency, Hirschberg's algorithm will be described in terms of the alignment problem.

Hirschberg's algorithm is based on the standard DPA where only the previous row of the D matrix is stored, as described at the end of Section 1.2.1. A divide-and-conquer method

```
D[0,0].diag = 0
D[i,0].vert = startGap + contGap*i,  i=1..|As|
D[0,j].horz = startGap + contGap*j,  j=1..|Bs|

{ All other entries on first row and first column set to infinity.}

for i = 1..|As|
  for j = 1..|Bs|
    D[i,j].diag = min(D[i-1,j-1].diag,
                      D[i-1,j-1].vert,
                      D[i-1,j-1].horz)
                  + (if As[i] = Bs[j] then
                       matchCost
                     else
                       changeCost)

    D[i,j].vert = min(D[i-1,j].diag + startGap + contGap,
                      D[i-1,j].vert +            contGap,
                      D[i-1,j].horz + startGap + contGap)

    D[i,j].horz = min(D[i-1,j].diag + startGap + contGap,
                      D[i-1,j].vert + startGap + contGap,
                      D[i-1,j].horz +            contGap)
  endfor
endfor

editCost = min(D[|As|,|Bs|].diag,
               D[|As|,|Bs|].vert,
               D[|As|,|Bs|].horz)
```

Listing 1.2: The DPA2l algorithm.

is used to recover the alignment. The idea is to split the DPA matrix on the middle row $p = |As|/2$, and find the crossing point of the optimal alignment on this row. This is done by aligning $As[1..p]$ with Bs and aligning the *reverse* of $As[p+1..|As|]$ with the *reverse* of Bs. An example of this is shown in Figure 1.2 for the sequences `ACCGGTCGGC` and `TGGTCGCCC` with Levenshtein costs. The final row of the forward alignment, and the final row of the reverse alignment are added together column by column to produce the middle row shown in the diagram. The minimum of this new row is taken, which in the example contains the value 5, which is also the edit cost of the two sequences. This column of this minimum value defines a split point for the alignment, so in this example the remaining alignment sub-problems are for `ACCGG` with `TGG` and `TCGGC` with `TCGCCC`. The algorithm the uses recursion on these two sub-problems to find the complete alignment. Note that this process always splits the first sequence in half (if it is of even length). For this example the optimal alignment found is shown below.

```
A  C  C  G  G  T  C  G  G  C  –
         |  |  |  |  |     |
T  –  –  G  G  T  C  G  C  C  C
```

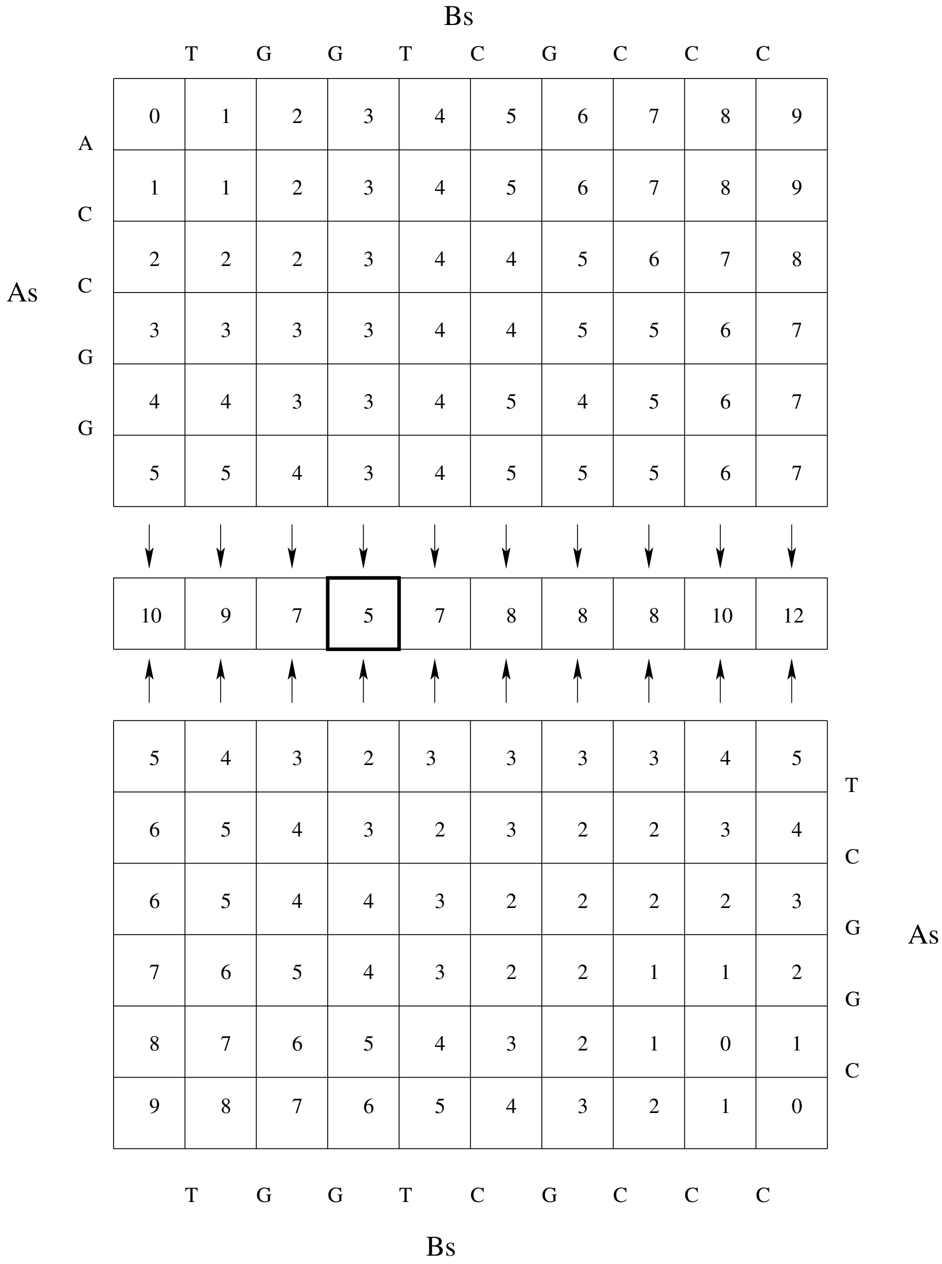

Figure 1.2: An illustration of the Hirschberg's algorithm.

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

1.3 Ukkonen's Algorithm

Ukkonen [62], and independently Myers [37], 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, 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 a match costs 0, and all other mutation costs are small positive integers. If however, the chosen costs do not meet these criteria, it may be possible to change the costs so that they do meet the criteria while leaving the alignment rank order unchanged [3].

Ukkonen's algorithm speeds up the basic DPA by recognising a number of facts about the DPA matrix: not all the entries of D are needed, the diagonals of D are 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. In terms of the U matrix this cell will be on the diagonal $ab = i - j$, and thus $U[i - j, D[i, j]] = i$. Given this correspondence between the D matrix and the U matrix, sometimes, for convenience, the operation of the Ukkonen algorithm will be discussed in terms of the D matrix although the Ukkonen algorithm does not use a D matrix. Ukkonen's algorithm for two sequences with Levenshtein costs is given in Listing 1.3. It is straightforward to generalise this algorithm to simple costs. Ukkonen's for simple costs shall be referred to the *Ukk2s* algorithm, referring to the *Ukkonen* style algorithm for *two* sequences with *simple* costs.

```
function Ukk(ab,d)

  if |ab|>d then return -infinity

  { Has this entry already been computed? }
  if computed(U[ab,d]) then return U[ab,d]

  dist = max( Ukk(ab+1, d-1),
              Ukk(ab,   d-1)+1,
              Ukk(ab-1, d-1)+1)

  { Extend the diagonal for a run of matches }
  while As[dist+1] = Bs[dist+1]
    dist = dist + 1

  U[ab,d] = dist
  return dist
endfunc

U[0,0] = max i s.t. As[1..i] = Bs[1..i]
editCost = 0
while Ukk(|As|-|Bs|, editCost) < |As|
  editCost = editCost + 1
```

Listing 1.3: Ukkonen's algorithm for Levenshtein costs (Ukk2s).

The function $Ukk(ab, d)$ computes how far along the sequence As can be reached on the ab diagonal of the D matrix for a cost of d. This is done by finding the distance that can be reached with a cost of $d-1$ on the same diagonal ab, and the two neighbouring diagonals $ab+1$ and $ab-1$. A loop then extends this distance while the sequences As and Bs match, corresponding to a run of matches down the diagonal of the D matrix. The method shown uses a recursive relationship with a *memo array* to store calculated values so no calculation need to be repeated. The alignment is recovered by tracing back through the U matrix following the choices made in the $max()$ function.

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. So, regions in the upper right and the lower left of the D matrix are able to be omitted from the calculation. There is a further saving when an run of exact matches occurs in both strings because the diagonals alongside the run of matches need not be calculated.

Figure 1.3 shows an example of the U matrix after aligning the sequences `ACCGGTCGGC` and `TGGTCGCCC`. The arrows indicate the path of an optimal alignment. The final diagonal is $ab = |As| - |Bs| = 10 - 9 = 1$. On this final diagonal the length of As is reached in the cell with an index cost of 5, thus the optimal edit cost is 5. Note that this example is the same as the example for the DPA shown in Figure 1.1. The corresponding cells of the DPA matrix that are computed by the Ukkonen matrix are shown in the DPA example as shaded cells.

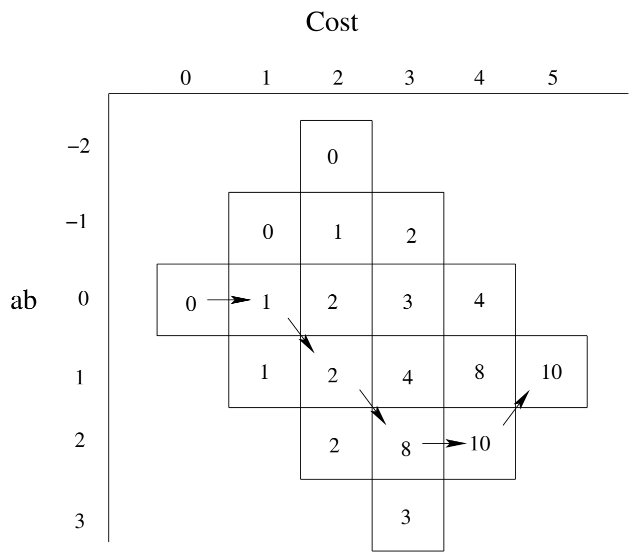

Figure 1.3: An example of the U matrix after aligning the sequences ACCGGTCGGC and TGGTCGCCC.

The worst case time complexity of Ukkonen's algorithm is $O(nd)$, although in most cases the average complexity of $O(n+d^2)$ is achieved. It is not immediately obvious which sequences cause the worst case performance of the algorithm. An example of such sequences is $As = a^k b^l$ and $Bs = b^l a^k$ where $k < l$. For Levenshtein costs these sequences have an edit distance of $d = 2k$. When aligning these sequences with Ukkonen's algorithm and Levenshtein costs, the inner loop (Listing 1.3) is iterated $l + (l-1) + (l-2) + ... + (l-k) = O(lk)$ times. The Ukk() function is called $O(d^2) = O(k^2)$ times. Thus the time complexity is $O(k^2 + kl)$ and since $k < l$ and $d = 2k$ this is $O(ld)$.

1.3.1 Ukkonen's Algorithm with Linear Gap Costs

Ukkonen's algorithm has been applied to aligning sequences using linear gap costs [39]. This algorithm will be referred to as the Ukk2l algorithm. As with the DPA for linear costs (Section 1.2.2), there are three distances instead of one in each cell of the U matrix. The extension of Ukkonen's algorithm to linear gap costs is similar to the extension of the basic DPA to the DPA for linear gap costs.

The Ukk2l algorithm is shown in Listing 1.4. The variables for the mutation costs are: `misCost` contains the cost of a mismatch, `startGap` contains the cost to start a gap, `contGap` contains the cost to continue a gap. There are three states in each cell, referenced with `matchState`, `insertState` and `deleteState`, corresponding to whether the last operation was a match (or mismatch), insert or delete respectively. These operation labels are for sequence `As` as the reference sequence.

A recent algorithm, Calign [12], uses a Ukkonen style method for aligning cDNA and genomic DNA. This algorithm uses restricted affine gap costs, which are essentially linear gap costs with a maximum cost for an insertion. This allows for large gaps as expected when aligning cDNA and genomic DNA by having a fixed cost for large gaps over a predefined size.

1.4 Three-way Alignment

The alignment algorithms discussed so far are all for the *two* sequence alignment problem. An obvious extension to this is the alignment of three sequences. Aligning three sequences can provide more information about how the sequences are related. Indeed, three way alignment can be useful for many applications such as sequence assembly and the building of evolutionary trees from DNA or protein sequences [50].

There are two commonly used cost functions for three way alignment, all-pairs costs and star costs. All-pairs costs give a cost to a three way alignment by summing the cost of the two-way alignment of each pair of sequences in the three way alignment. Star costs use the sum of

```
function Ukk_linear(ab, d, state)

  if |ab|>d then return -infinity

  { Has this entry already been computed? }
  if computed(U[ab,d,state]) then return U[ab,d,state]

  if (state == matchState) then
    m_dist = Ukk(d-misCost, ab, matchState) + 1
    i_dist = Ukk(d,         ab, insertState)
    d_dist = Ukk(d,         ab, deleteState)
  elseif (state == insertState) then
    m_dist = Ukk(d-startGap-contGap, ab+1, matchState)
    i_dist = Ukk(d-contGap,          ab+1, insertState)
    d_dist = Ukk(d-startGap-contGap, ab+1, deleteState)
  elseif (state == deleteState) then
    m_dist = Ukk(d-startGap-contGap, ab-1, matchState)  + 1
    i_dist = Ukk(d-startGap-contGap, ab-1, insertState) + 1
    d_dist = Ukk(d-contGap,          ab-1, deleteState) + 1
  endif

  dist = max( m_dist, i_dist, d_dist )

  { Extend the diagonal for a run of matches }
  while As[dist+1] = Bs[dist+1]
    dist = dist + 1

  U[ab, d, state] = dist
  return dist
endfunc

U[0,0] = max i s.t. As[1..i] = Bs[1..i]
editCost = 0
while Ukk_linear(|As|-|Bs|, editCost, matchState) < |As|
  editCost = editCost + 1
```

Listing 1.4: Ukkonen's algorithm for linear gap costs.

the cost between each sequence and a consensus sequence. This is shown diagrammatically in Figure 1.4.

Figure 1.4: All-pairs costs and star costs.

As an example, using Levenshtein costs, the alignment shown below has an edit cost of 9 under an all-pairs cost function since the first and second sequences have an edit cost of 3, the first and third sequences an edit cost of 4, and the second and third sequences a cost of 2. Under a star cost function the edit cost is 5, and a consensus sequence `TGCTG`. This is easily seen from summing the edit cost between the consensus sequence and each sequence in turn, which gives $2 + 1 + 2 = 5$. Note the last character of the consensus sequence could equally be a ‘G’, ‘T’ or ‘A’.

$$\begin{array}{cccccc} A & T & G & A & T & G \\ & | & | & & | & \\ - & T & G & C & T & T \\ & & | & | & | & \\ - & - & G & C & T & A \end{array}$$

For alignment of more than three sequences, it is possible to use all-pairs costs, star costs or *tree costs* [49]. Tree costs assume the sequences to be aligned are on the leaves of a tree, where each internal node represents an unknown ancestor sequences and has three neighbours. The idea of tree costs is to minimise the sum of the pairwise alignments between all neighbouring sequences in the tree. For three sequence, tree costs are the same as star costs. All-pairs costs, star costs and tree cost are shown for four sequences in Figure 1.5 with solid dots representing known sequences and hollow dots representing unknown sequences. When using tree costs, the leaves of the tree the sequences are on must be specified or inferred. Using all-pairs costs tends to produce an alignment that maximises the number of positions in which all of the sequence match. Whereas, tree costs minimises the number of mutations needed between sequences to reach a common ancestor.

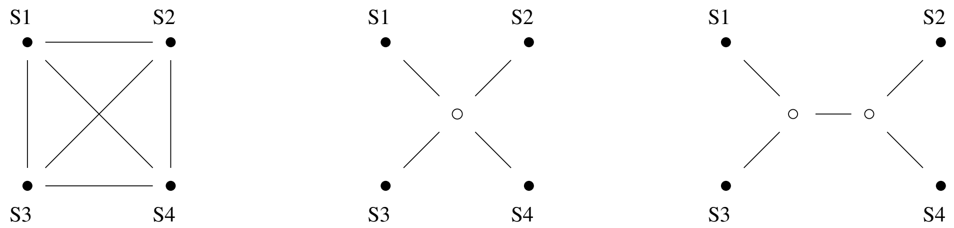

Figure 1.5: All-pairs costs, star costs and tree costs for four sequences.

Often when aligning sequences we are interested in making an inference about the most likely common ancestor(s) of the sequences. In standard sequence alignment with two sequences, the best inference that can be made is that the ancestor sequence lies somewhere “between” the two aligned sequences. The sequences do not give any other information about the ancestor. So, one of the aligned sequences could arbitrarily be chosen as the ancestor, and therefore the other sequence as the descendent. Alignment of three or more sequences allows an inference to be made about the ancestor sequence (or ancestors for tree costs). Star costs assume there to be one common ancestor between the observed sequences. Each observed sequence may be different distance from the ancestor sequence. The DNA sequences for the Transthyretin gene from a human, mouse and rat can be used as an example of this. The alignment of these three sequence under Levenshtein costs is given in Appendix B. As

Table 1.1: A summary of the $dirCost()$ and $charCost()$ functions from the DPA3s algorithm for both star costs and all-pairs costs.

	Star cost		All-pairs cost	
Alignment characters	`dirCost()`	`charCost()`	`dirCost()`	`charCost()`
`<x,x,x>`	0	0	0	0
`<x,x,->`	$indel$	0	$2 \times indel$	0
`<x,-,->`	$indel$	0	$2 \times indel$	0
`<x,x,y>`	0	mis	0	$2 \times mis$
`<x,y,->`	$indel$	mis	$2 \times indel$	mis
`<x,y,z>`	0	$2 \times mis$	0	$3 \times mis$

would be expected the mouse and rat gene are more similar to each other than to the human gene.

While standard algorithms for aligning two sequences cannot make a single inference about the ancestor sequence, the new algorithm developed in Chapter 5 can. This is because this low information alignment algorithm has a model for the ancestor sequence. So, it is possible to make an inference about how far the observed sequences have mutated from a possible ancestor sequence.

1.4.1 DPA for Three Sequences

The alignment of three sequences with simple costs can be computed using an extension of the basic DPA (Section 1.2.1) in $O(n^3)$ time and space. This algorithm shall be referred to as the DPA3s algorithm and is shown in Listing 1.5. The algorithm as presented here can be used for star costs or all-pairs costs depending on the definition of the `dirCost()` and `charCost()` functions. The `dirCost()` function calculates the cost of any insertions or deletions depending on the direction of the neighbour. The `charCost()` function calculates the cost of any mismatches. The computation of these functions are summarised in Table 1.1, where $indel$ is the cost of an insert or delete, and mis is the cost of a mismatch. The left column of this table contains a pattern for the possible characters being aligned. The three characters in each of these patterns can be arranged in any order. The cost of aligning the three characters is the sum of the `dirCost()` and `charCost()` columns for either star costs or all-pairs costs.

```
D[0,0,0] = 0

for i = 0..|As|
  for j = 0..|Bs|
    for k = 0..|Cs|
      if (i=0 and j=0 and k=0) then
        continue

      foreach (di,dj,dk) in Neighbours
        if not inMatrix(i+di,j+dj,k+dk) then
          continue

        D[i,j,k] = min(D[i,j,k],
                       D[i+di,j+dj,k+dk] +
                         dirCost(di,dj,dk) +
                         charCost(if (di) then As[i] else '-',
                                  if (dj) then Bs[j] else '-',
                                  if (dk) then Cs[k] else '-'))
      endforeach
    endfor
  endfor
endfor

editCost = D[|As|,|Bs|,|Cs|]
```

Listing 1.5: The DPA3s algorithm.

1.4.2 Other Three-way Alignment Algorithms

There has been much research into the three-way and more general k-way alignment problems. Allison [2] presented an algorithm for three-way alignment with simple costs based on Ukkonen's fast alignment algorithm. Allison's algorithm optimally aligns three sequences in $O(n + d^3)$ time on average and in $O(d^3)$ space.

Carrillo and Lipman [11] presented an algorithm for aligning k-sequences using sum of pairs costs. This algorithm can be viewed as a k-dimensional DPA computation where the volume of the k-dimensional hypercube to be computed is bounded. An optimal alignment of k-sequences can seen as a path through the k-dimensional hypercube. Every pair of the k-sequences corresponds to a two dimensional face of the hypercube. Thus, an optimal alignment of the k-sequences has a *projection* onto every two dimensional face corresponding to an alignment of each pair of sequences. The alignment of a pair of sequences from such a projection must be no better than the *optimal* alignment between that pair of sequences. So, to compute the optimal alignment of the k-sequences, the volume of the hypercube that must be computed can be limited by using the optimal alignment between every pair of sequences and a "guessed" k-sequence alignment. The "guessed" alignment provides an upper-bound on the optimal alignment, which can be used with the optimal pairwise alignment to bound the area of each two dimensional face of the hypercube that must be considered. The bounded area on each face provides a bound on the volume of the hypercube that must be computed. Carrillo and Lipman's algorithm is limited to all-pairs costs, this was extended by Altschul and Lipman [9] to tree costs and star costs.

1.5 Aligning Non-random Sequences

Most alignment algorithms, and indeed all those discussed so far, make the tacit assumption that the sequences to be aligned are random. That is, the alignment of any part of a sequence is just as important as every other part of the sequence. If, however, the sequences to be aligned are non-random, this should be taken into account when aligning the sequences. For example, consider a family of sequences that generally have two types of subsequences or regions. One type of subsequence is a series of repeated characters which we will consider to be a low information region. The other type of subsequence is more interesting and contains a high amount of information. Two sequences are to be taken from this family and aligned. It is important to have a good alignment between the high information regions of these sequences since a good match in these regions would imply the sequences are related. The low information regions are less important in the alignment since they are more likely to appear by chance in both sequences and hence give less information as to whether the sequences are actually related or not.

In Chapter 5 this intuition of aligning the significant parts of a sequence is made precise. A DPA style algorithm is developed to align sequences that are noisy observations of a non-random parent sequence. The standard alignment algorithm (Section 1.2.1) can be considered a special case of this algorithm. This non-random sequence alignment algorithm uses a mutation cost function similar to that of the basic DPA, and it is discussed how this can be extended to linear-gap type costs similar to those of Section 1.2.2. The check-point method developed in Chapter 2 can also be used to reduce the space complexity of the new non-random sequence alignment algorithm from $O(n^2)$ to $O(n)$.

Alignment of sequences can also be used to decide whether the sequences are related or not. If the sequences are non-random, and this is not taken into account when aligning the sequences, the sequences will appear more related than they really are. Previous work such as that by Fitch [17] and later Wootton [66] described how to deal with the many false-positives that arise if the non-random nature of the sequence is not taken into account. In Chapter 5 it is shown how the new alignment algorithm can be used to test the relatedness of sequences by apportioning importance to all parts of the sequences.

1.6 Taxonomy of Alignment Algorithms

There have been many algorithms introduced so far. The intent in this section is to clarify how new algorithms developed in this thesis relate to the previous work in the area.

Some of the alignment algorithms discussed in this thesis can be classified by the following attributes: alignment of two sequences versus three sequences; simple costs versus linear gap costs; the standard DPA versus Ukkonen's faster algorithm; normal versus space saving with Hirschberg's method or the new check-point method; alignment of non-random versus random sequences. For each of these five attributes, the first mentioned is typically the simpler case and the second the more general (and complex) and often more desirable. Ukkonen's algorithm is generally preferred over the DPA because the time complexity is reduced. Simple costs are a special case of linear gap costs thus linear gap costs are more versatile, and in many applications provide a better model of the sequence mutations. Optimal alignment of three strings is better than alignment of two sequences in many cases such as building evolutionary trees and multiple sequence alignment. Taking into account the randomness of the sequences can be important for a number of alignment applications.

From these five independent attributes there are $2^5 = 32$ different algorithms possible. The first four of these attributes and algorithms with them are summarized in Table 1.2. This table shows how the new algorithms developed in this thesis relate to previously known alignment algorithms in terms of the first four independent attributes. First the algorithms that do not use any space saving method will be discussed. The simplest of the algorithms is for two strings with the standard DPA and simple costs, this has been discussed briefly in Section 1.2.1. The next simplest algorithm uses linear gap costs instead of simple costs and was summarized in Section 1.2.2. In Section 1.3 the algorithm by Ukkonen [62] was discussed which is for two sequences and simple costs. The final algorithm for two strings again uses the Ukkonen speed-up with two strings but extends it to linear gap costs [39].

The simplest three string alignment algorithm is an extension of the DPA for two strings and uses simple costs. Ukkonen's speed-up can also be applied to three strings [2]. Three string alignment with linear gap costs is also possible; an algorithm for this problem based on the DPA was shown by Gotoh [21]. However, Gotoh's algorithm is somewhat limited. These limitations are pointed out, and a new algorithm is developed in Chapter 3 which Gotoh's is a special case of.

The main contribution of Chapter 3 is a combination of the more desirable attributes, an alignment algorithm for three strings with linear gap costs using the Ukkonen speed-up. The average time complexity of the algorithm for strings of length n, is shown to behave as $O(d^3 + n)$ where d is the edit cost.

The eight abovementioned algorithms have their space complexity reduced by applying a space saving method. Hirschberg [27] described this for the LCS problem, which is closely related to the standard DPA with Levenshtein costs. Myers and Miller [38] applied Hirschberg's method to the DPA for two sequences with linear gap costs reducing the space complexity to $O(n)$. Chapter 2 describes the application of the check-point method to this same algorithm also reducing the space complexity to $O(n)$. The check-point method is further developed in Chapter 2 and applied to Ukkonen's algorithm for aligning two sequences with both simple and linear gap costs. The main contribution of Chapter 3 is the linear gap cost algorithm for three sequence using the Ukkonen speed-up, that is, the Ukk3l algorithm. In Chapter 4 the check-point method is applied to the Ukk3l algorithm culminating in the Ukk3l_cp algorithm.

The new algorithm developed in Chapter 5 can also be classified by these attributes. This algorithm is for aligning two sequences that are non-random using a DPA style algorithm with simple costs in the first case with an explanation of how to extend to linear costs. It is also possible to apply the space saving check-point method of Chapter 2 to this new algorithm to reduce the space complexity to $O(n)$.

Table 1.2: Summary of the alignment algorithms.

		DPA-based		Ukkonen-based	
		2 sequences	3 sequences	2 sequences	3 sequences
Normal	Simple costs	Levenshtein [33] Sellers [52]	Various	Ukkonen [62]	Allison [2]
	Linear costs	Gotoh [20]	Gotoh [21][a]	Myers and Miller [39]	**Chapter 3**
Space saving	Simple costs	Hirschberg [27] Eppstein (unpublished)	**Chapter 2**[b]	Myers [37] **Chapter 2**[c]	**Chapter 2**[b]
	Linear costs	Myers and Miller [38] **Chapter 2**[c]	**Chapter 3**[b]	**Chapter 2**	**Chapter 4**

[a]For star costs Gotoh's algorithm is an approximation to the new algorithm developed in Chapter 3.

[b]Mentioned, but not described in detail

[c]Chapter 2 uses the check-point method for space saving, while the other uses Hirschberg's method.

Chapter 2

Sequence Alignment in Linear Space

2.1 Introduction

The work presented in this chapter has been published in Information Processing Letters [42].

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 chapter a new method is presented 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 method shall be referred to as the check-point method, for reasons that will become apparent later. This check-point method has several advantages over previous linear space methods, such as: simplicity, being easily adapted to different cost functions, and the practical advantage of easily being able to trade available memory for running time.

When aligning sequences of similar length, $\sim n$, the time complexity of the basic DPA is $O(n^2)$. The space complexity is $O(n^2)$ if an alignment is required or $O(n)$ if only the edit cost is desired. In Section 1.2.3 Hirschberg's [27] divide and conquer algorithm that allows the DPA to compute an alignment in $O(n)$ space was briefly explained. The new check-point method of this chapter is an alternative to Hirschberg's algorithm which has a number of advantages. This alternative has been briefly described by [28] who attributes it to Eppstein (unpublished), for the simplest case, the basic DPA with simple costs. In this chapter it is shown how this method may be applied to many other cost functions and how it may also be combined with Ukkonen's fast algorithm to give a flexible, fast and space efficient alignment algorithm.

Ukkonen [62] devised an algorithm that runs faster than the basic DPA, especially for similar sequences. On average, this algorithm has time complexity $O(n+d^2)$, where d is the edit cost between the two sequences. If an alignment is required, $O(d^2)$ space is needed, otherwise $O(d)$ space is required to determine just the edit cost. Ukkonen's algorithm was explained in Section 1.3. Here it is shown how this new check-point method can be applied to reduce the space complexity of Ukkonen's algorithm to $O(d)$ while still producing an alignment. This is discussed in detail for both simple and linear gap costs. Previously, Hirschberg's [27] method was used for retrieval of an alignment in linear space from Ukkonen's algorithm, however this is more complicated (see [37, 28]) and less versatile than the new check-point method presented in this chapter.

A recent paper by Grice *et al.* [23] uses an approach similar to that presented here, but with a different motivation. They wish to train a Hidden Markov Model [46] by using all possible paths through the DPA matrix. In this chapter the interest is in finding an optimal alignment. Grice *et al.* do suggest a method for the single best path, however, it is more complicated than the new check-point method, and the check-point method is also applied to the more advanced Ukkonen algorithm.

The naming scheme introduced in Chapter 1 is extended to include algorithms which result from have the check-point method applied. For example, when the basic DPA algorithm, also known as the DPA2s algorithm, has check-point method applied to it, the resulting algorithm shall be known as the DPA2s_cp algorithm. Likewise combining the check-point method with the Ukk3l algorithm of Chapter 3 the result is the Ukk3l_cp algorithm of Chapter 4.

2.2 The Basic DPA

The basic DPA algorithm, also referred to as the DPA2s algorithm, was described in detail in Section 1.2.1. This algorithm has a time and space complexity of $O(n^2)$. Hirschberg [27] showed how the space complexity could be reduced to $O(n)$ (see Section 1.2.3). Here an alternative to Hirschberg's method is developed that is superior in a number of ways.

2.2.1 Check-Pointing

The algorithm described below has the same benefit as Hirschberg's (that is, it reduces 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 2.4.1).

The idea of this algorithm is similar to Hirschberg's, that is, to repeatedly split the D matrix. To achieve this it is necessary to find a cell on the middle row of the D matrix, row p say (that is, $p = |As|/2$), that lies on the optimal alignment. An illustration of this is shown in Figure 2.1. We want to determine a column q such that the cell $D[p, q]$ lies on the optimal alignment. This is done by having every cell after row p carry the column index of the cell on row p 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 column index q. 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. Referring to the diagram, it can be seen that the shaded regions of the D matrix are not computed on the second pass. Only two rows of the D matrix are ever stored, the current row and the previous row.

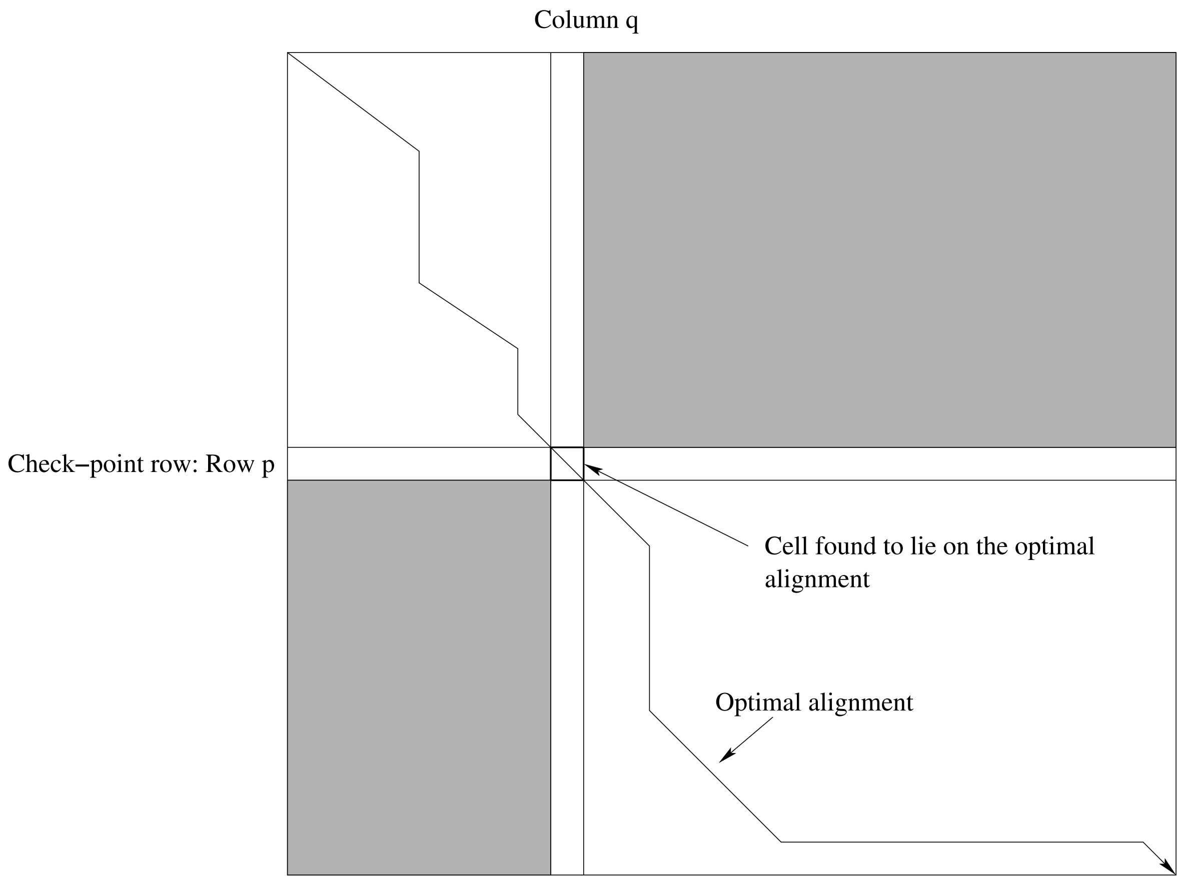

Figure 2.1: An illustration of the new check-point method on the DPA.

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

An example is given in Figure 2.2 using Levenshtein costs, that is a cost of 1 for an insert, delete or mismatch and a cost of 0 for a match. This example aligns the sequences `As=CGCA` and `Bs=AAGT` to determine an optimal alignment, there are in fact four 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 two rows are stored at any given time). The shaded cell on the check-point row is found to lie on the optimal alignment (that is, $D[p, q]$ from above). This cell is called the split cell because it is used to divide the problem into two parts. At this stage of the algorithm there are two sub-problems; the alignment of `CG` with `AAG` and of `CA` with `T`. These two sub-problems are shown in step two. In step three, each sub-problem from step two has been split further resulting in four sub-problems. Each of these consist of only two rows, so the alignments can be determined directly without the need for further check-pointing. The alignments produced from step three are concatenated together to produce an optimal alignment for the original sequences.

It is important to note that in this example, the values in the D matrix have been shown to be consistent at each level of recursion. To do this the edit costs in row p must be stored so they can be used at the next level of recursion. This storing of the contents of cells on a row is referred to as *check-pointing* a row. Thus the reason this method is referred to as the check-point method.

For the simple cost alignment with the DPA it is not strictly necessary to check-point any rows at all, and it is essentially this variant that Hirschberg [28] attributes to Eppstein. It is possible to restart the top left cell of each region with an edit cost of 0. However, the more complex algorithms such as linear gap costs or Ukkonen's algorithm *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 the middle row is *not* required to be saved. However, for consistency, this modification of the DPA will still be known as the 'DPA with check-pointing', or the DPA2s_cp algorithm.

The DPA2s_cp algorithm is given in Listing 2.1. It is worth noting that a large portion of this algorithm is the same as the DPA2s algorithm (see Figure 1.2.1). The purpose of the latter half of the DPA2s_cp algorithm is to determine the split of the D matrix and to perform the recursion. This general form of check-pointing based algorithms is maintained when applied to different cost functions (and even when applied to the Ukkonen algorithm, see Section 2.4.1). The fact that the new check-point method is to a large extent independent of

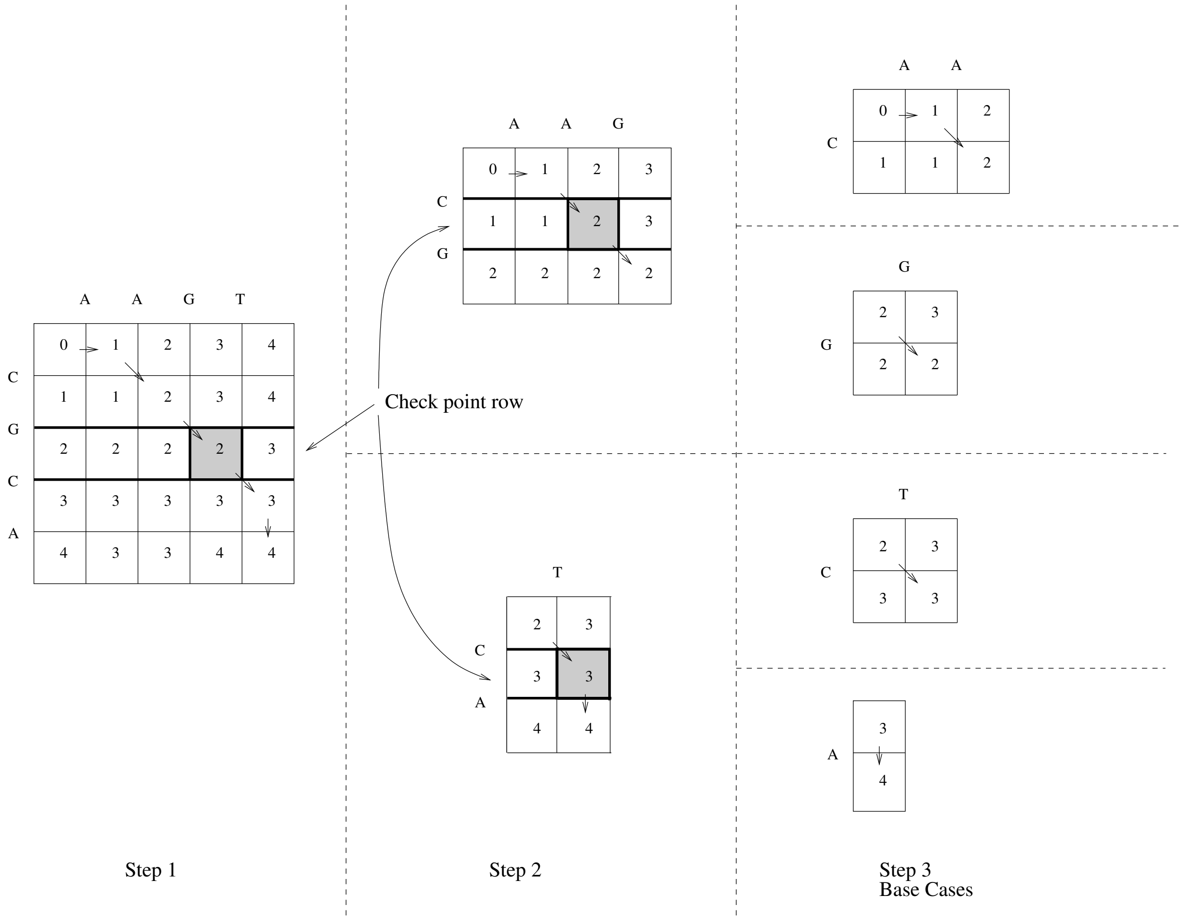

Figure 2.2: An example of check-pointing on the DPA matrix for sequences CGCA and AAGT, using Levenshtein costs.

the underlying algorithm is a major advantage of this method.

It is easily seen that the DPA2s_cp algorithm requires only $O(n)$ 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 check-point 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 row. So the work done by the DPA with check-pointing is $|D| \times (1+1/2+1/4+...)$, where $|D|$ is the size of the D matrix. Since $|D|$ is about n^2 the time complexity is $O(n^2)$. Figure 2.3 shows the experimental results confirm this analysis. The test data for all the sample runs 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).

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

  { Row to check-point }
  p = (sRow+fRow)/2

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

  for i = sRow+1..fRow
    D[i mod 2, sCol] = i-sRow
    f[i mod 2, sCol] = sCol

    for j = sCol+1..fCol
      { Standard DPA calculation }
      D[i mod 2, j] = min(D[i mod 2 ,    j-1] + insertCost ,
                          D[(i-1) mod 2, j + deleteCost ,
                          D[(i-1) mod 2, j-1)] +
                            (if As[i] = Bs[j] then
                               matchCost
                             else
                               changeCost))


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

    endfor
  endfor


  {Base Cases 1 or 2 rows}
  if (fRow-sRow = 1 or = 2) then
    {Determine alignment directly from D matrix}
    return
  endif


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

end.
```

Listing 2.1: The DPA2s_cp algorithm: simple costs DPA with check-pointing to determine the alignment.

Trading Space for Time

A major advantage of the new check-point method over Hirschberg's, is that in practice a check-point based algorithm can be sped up by keeping more than one check-point for each pass through the DPA matrix. The work for the next step is then reduced by the number of check-points kept. Figure 2.4 shows an example of this where two rows are check-pointed, at $|As|/3$ and $2|As|/3$. The shaded region does not need to be recomputed at the next recursion

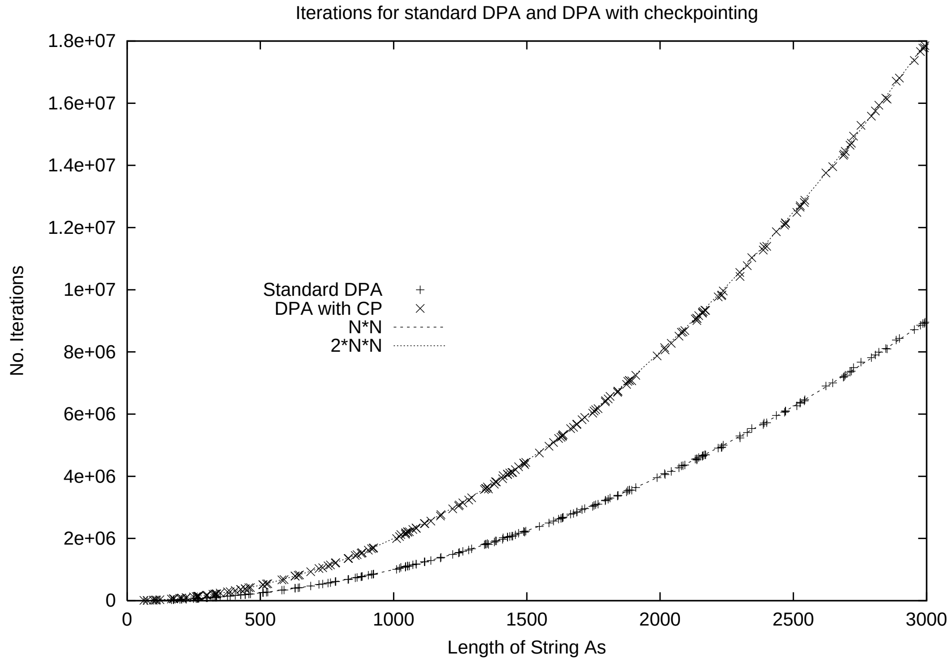

Figure 2.3: Comparison of the number of iterations of the loop in the basic DPA against that of the DPA with check-pointing. Both $\frac{N^2}{2}$ and n^2 are plotted. Note: the x-axis is the length of string As only, and while Bs will be of similar length it will differ slightly.

level. So in this example the area computed at each step is one third of the previous 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 useful feature because available memory can be traded for running time. A DPA based algorithm that uses the check-point method keeping x check-points, where $x \geq 1$, will run a constant k times as slow as the basic DPA. Here, $k = 1 + (x+1)^{-1} + (x+1)^{-2} + (x+1)^{-3} + ... = \sum_{i=0}(x+1)^{-i}$, which for the simplest case, $x = 1$, gives $k = 2$ as discussed in the previous section.

2.3 DPA with Linear Gap Costs

The simple costs used in the basic DPA tend to produce alignments that have many short indels. When aligning biological sequences, and indeed many other types of sequences, it is often better to have alignments that contain fewer, longer indels rather than many short indels. For this reason, linear gap costs are considered a better model for aligning biological sequences than simple costs. Under linear gap costs a run of insertions or deletions are treated as a single event and given a cost $a \times x + b$ where a and b are constants, and x the length of the run. Simple costs are a special case of linear costs with $b = 0$.

The basic DPA can be modified to compute an optimal alignment using a linear gap cost function (see [20]). Following the naming scheme adopted earlier, the DPA for linear gap

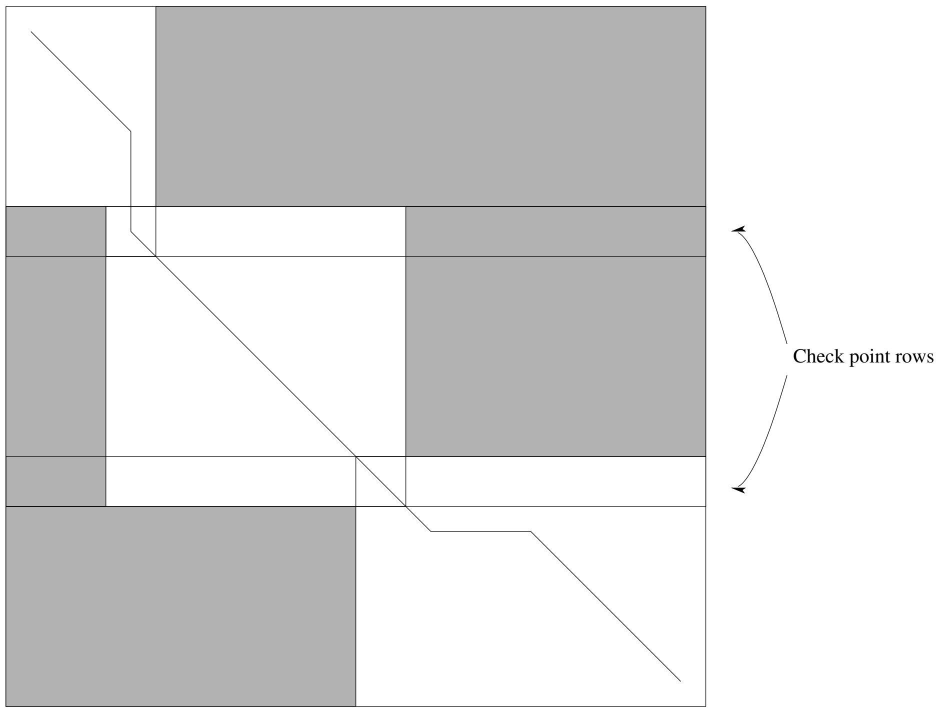

Figure 2.4: An example of the DPA matrix when two check-point rows are kept instead of one.

costs shall be known as the DPA2l algorithm. The DPA2l algorithm has in each cell of the DPA matrix 3 alignment costs — one each for the 3 possible *states* corresponding to the last step, match/mismatch, insertion or deletion. The DPA2l algorithm has time complexity $O(n^2)$ and space complexity $O(n^2)$, the same as for the DPA2s algorithm. Several methods have attempted to reduce the space usage of the DPA2l algorithm by constant factors [57, 65, 7, 22], although all still have a space complexity of $O(n^2)$. Myers and Miller [38] applied Hirschberg's method to reduce the space complexity to $O(n)$. Applying Hirschberg's method to the DPA2l algorithm requires the ability to combine the results of a forward pass and a reverse pass (as for Hirschberg's basic algorithm, see 1.2.3) to find an optimal split point. This is far more complicated than using the check-point method. The check-point method also has the practical advantage of being able to trade memory usage for execution time.

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 because of the 3-state nature of the alignment algorithm which needs to be restarted with the correct state information. As the DPA is being run forward the check-point row 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 the 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 row. 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). This is needed because the final cell of the D matrix, $D[|As|, |Bs|]$, contains three edit costs in its three states, the one with the lowest cost defines the end of the optimal alignment. Each state in the final matrix cell may have derived from a different cell on the check-point row. Thus each state in each matrix cell must carry the information about which check-point cell it is derived from. The same time/space complexities apply as for the simple cost case (the row check-pointing can be done in $O(1)$ time by using pointers to rows). The algorithm resulting from applying the check-point method to the DPA2l algorithm shall be referred to as the DPA2l_cp algorithm.

One advantage of the check-point method described here over Hirschberg's method is that the check-point based algorithms for more complex cost functions (for example, 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 row 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 difficult to find the split point. This difficulty arises after the forward and reverse pass of the base algorithm when the optimal split point must be calculated. For complicated cost functions it is unclear how to perform this calculation.

2.4 Ukkonen's Algorithm

Ukkonen [62] (and independently Myers [37]) 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 similar length, 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. There are two degrees of freedom in choosing mutation costs [3]. It is possible to scale all costs by a constant factor or to add a constant per character to the costs and leave the optimal alignment unchanged. So if the desired alignment mutation costs do not meet the above criteria for Ukkonen's algorithm it will often be possible to choose costs that do meet the criteria that are still functionally equivalent. Ukkonen's algorithm for two sequences and simple costs, referred to as the Ukk2s algorithm in this thesis, is discussed in more detail in Section 1.3.

Ukkonen's algorithm uses an alternative representation of the D matrix called the U matrix.

Each row of the U matrix corresponds to a diagonal of the D matrix, and each column of the U matrix to a contour of fixed cost in the D matrix. Each cell of the U matrix contains an integer representing the distance that can be obtained for a specific cost, along a specific diagonal of the D matrix. Using this correspondence of the U matrix to the D matrix it is often useful to discuss the Ukk2s algorithm in terms of the D matrix whilst in reality it operates on the U matrix. When discussing the Ukk2s algorithm in terms of *diagonals* it is important to remember that these are diagonals of the D matrix.

An example of the U matrix is shown in Step 1 of Figure 2.6 for the same example as shown for the DPA in Figure 2.2. In this figure, the U matrix has for diagonal $ab = -1$ and cost 2 the contents 2 (that is, $U[-1, 2] = 2$), 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 Figure 2.2) 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. Figure 2.5 shows an example that illustrates these "holes". In this figure a DPA matrix is shown for the alignment of the sequences `ATAGA` and `AGAGCGTAGC` using Levenshtein costs. The shaded cells indicate the D matrix cells that correspond to the U matrix cells that are computed when aligning these sequences with Ukkonen's algorithm. Note there is a region, or "hole", in the center of the D matrix that is not computed by Ukkonen's algorithm. These sequences have four different optimal alignments, two of which are found by Ukkonen's algorithm. The arrows in the figure indicate one of the optimal alignments. For a brief discussion of the complexity of Ukkonen's algorithm see section 2.4.2.

Each column of the U matrix is completely defined within terms of the previous column (for Levenshtein costs). Hence it is possible to calculate the edit cost by using only $O(d)$ space, while the whole U matrix is required if the alignment is desired. This is done by changing all matrix indexing on cost of the U matrix to be computed modulo 2 (that is, $U[ab, d] \rightarrow U[ab, d \bmod 2]$). This is identical to reducing DPA to $O(n)$ space when the alignment is not wanted (see start of section 1.2.1).

		A	G	A	G	C	G	T	A	G	C
	0	1	2	3	4	5	6	7	8	9	10
A	1	0	1	2	3	4	5	6	7	8	9
T	2	1	1	2	3	4	5	5	6	7	8
A	3	2	2	1	2	3	4	5	5	6	7
G	4	3	2	2	1	2	3	4	5	5	6
A	5	4	3	2	2	2	3	4	4	5	6

Figure 2.5: The DPA matrix for aligning the sequences ATAGA and AGAGCGTAGC. The shaded cells correspond to the cells of the U matrix that are computed by Ukkonen's algorithm.

2.4.1 Ukkonen's Algorithm in Linear Space

Myers [37] applied Hirschberg's method 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 $O(n+d^2)$, although in practice it probably behaves like $O(n+d^2)$. In fact Myers does not discuss the expected time complexity of his linear space algorithm. The reason is that the computational work is not distributed evenly over the Ukkonen matrix, making it impossible to split the work evenly into halves. The method described here to reduce the space complexity of Ukkonen's algorithm to linear, has the same advantages over Myers [37] as the DPA2s_cp algorithm has over Hirschberg [27]. 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-point method used on the DPA, on the D matrix, can be adapted for use on the U matrix with Ukkonen's algorithm. The outer loop of the Ukkonen algorithm works along the columns of the U matrix. So whereas rows were check-pointed in the DPA algorithm, columns will be check-pointed in the Ukkonen algorithm. When applying the check-point method to the DPA algorithm, either the rows or the columns may be check-pointed, whichever is the outermost loop. With the Ukkonen algorithm it is necessary to check-point on the columns. 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 DPA2l_cp algorithm it is necessary to store the contents of the check-point column of the U matrix. This is because the contents of the split cell (that is, the distance along the As string), is needed for recursion on the second half of the U matrix.

An example of this is shown in Figure 2.6, using the same sequences as Figure 2.2. 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, also called the split cell, is highlighted. In Step 1, the cell $U[ab = -1, cost = 2]$ is found to lie on the optimal alignment; this is the split cell. The problem is split into two parts, the alignment from $U[ab = 0, cost = 0]$ to $U[ab = -1, cost = 2]$ and from $U[ab = -1, cost = 2]$ to $U[ab = 0, cost = 4]$. Step 2 shows these two sub-problems each split again in two. The final step, Step 3, consists of four alignment problems that all happen to be base cases so the alignment can be determined directly without further recursion. The sub-alignments are joined to produce the final alignment.

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 Ukk2s_cp 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 (that is, 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-point method 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.

Listing 2.2 shows Ukkonen's algorithm with check-pointing added. This is for Levenshtein 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 single mutation cost. It is worth comparing this algorithm to the DPA2s_cp algorithm (Listing 2.1)

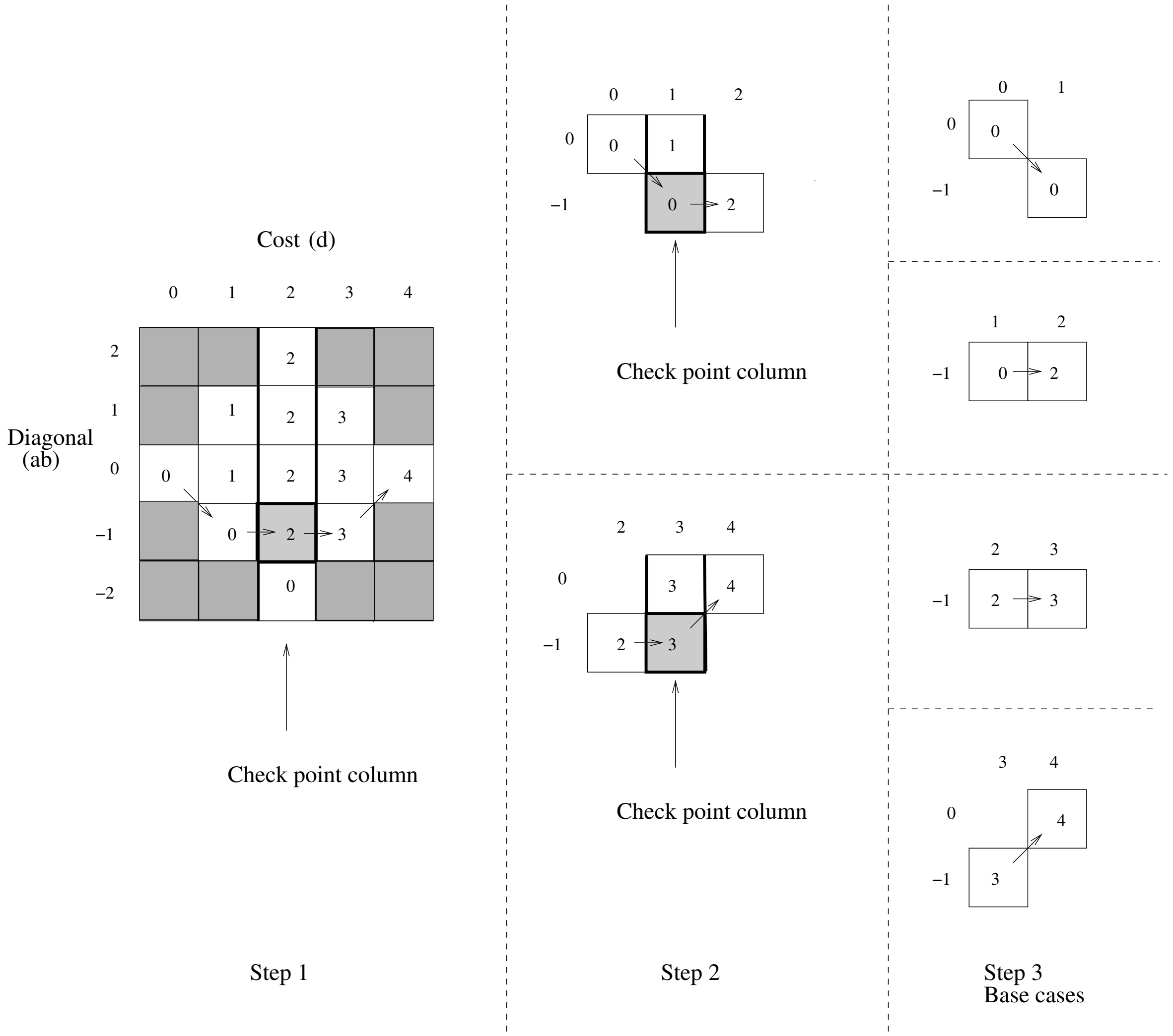

Figure 2.6: An example of the Ukk2s_cp algorithm: the check-point method with Ukkonen's algorithm for sequences CGCA and AAGT, using insertion, deletion and mismatch costs of 1 and a match cost of 0.

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-point method is applied to different algorithms and even different cost functions, is a major advantage of this method.

2.4.2 Complexity of Ukk2s and Ukk2s_cp

The worst case time 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 without any "holes". 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 because of the "holes" left in the D matrix. An example of these "holes" was given in Figure 2.5. A brief explanation of this expected time complexity follows (for a fuller explanation see [37]).

```
procedure CP_UKK(sDiag, sCost, sDist, fDiag, fCost)
    {First base case}
    if (sCost = fCost) then return

    U[sDiag, sCost mod 2] = sDist

    {Set check-point cost}
    q = (fCost+sCost)/2

    {Now do standard Ukkonen}
    { General step, iterated from sCost until U[fDiag, fCost] }
        U[ab,d] = max(U[ab+1, (d-1) mod 2],
                      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) then
            {Setup the check-point}
            CP[ab] = U[ab, d mod 2]
            f[ab, d mod 2] = ab
        elseif (d > q)
            { Carry check-point information}
            f[ab, d mod 2] = f[(ab,d) chosen in max()]
        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]. Recursion on the two subparts }

    CP_UKK(sDiag, sCost, sDist, p, q)
    CP_UKK(p, q, f[p], fDiag, fCost)
end.

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

Listing 2.2: The Ukk2s_cp algorithm: the check-point method on Ukkonen's algorithm with Levenshtein mutation costs.

Without Check-Pointing:

The number of iterations of the outer loop is always $\frac{d^2}{2}$.

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 $\frac{1}{\Sigma-1}$ (where Σ is the alphabet size). There are $\sim \frac{d^2}{2}$ positions in the matrix where such matches can occur.

So the expected number of inner loop iteration $= L + \frac{d^2}{2(\Sigma-1)}$

With Check-Pointing:

If we now consider the Ukk2s_cp algorithm, where the edit distance is already known so the U matrix can be split at $\frac{d}{2}$ we obtain the following:

Outer loop iterations $= \underbrace{\frac{d^2+1}{2} + \frac{d^2+2}{4} + \frac{d^2+6}{8} + ...}_{\sim \log_2 d \quad \text{terms}} \approx d^2 + \log_2 d \approx d^2$

Inner loop iterations $\approx L(1 + \log_2 d) + \frac{d^2}{2(\Sigma-1)} \times (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...)$

Figures 2.7 and 2.8 show plots for experimental data. These plots of the Ukk2s_cp algorithm loop counts do not take into account first working out the edit distance.

The expected time complexity of the Ukk2s_cp algorithm is thus $O(n \log_2 d + d^2)$; in practice the d^2 term dominates. The Ukk2s_cp algorithm implementation — not particularly optimized — was found to be typically 3.5 times as slow as the standard Ukkonen algorithm (see Figure 2.9). Run times of the Ukk2s_cp algorithm included running the cost only version of the Ukkonen algorithm first, followed by the Ukk2s_cp algorithm to determine the alignment. If the first iteration through the U matrix was check-pointed at $n/2$, as previously suggested, and the mutations were uniformly distributed, then the column $n/2$ would be a good approximation to $d/2$ and it would be expected that the Ukk2s_cp algorithm would be only 2.5 times as slow as the Ukk2s algorithm.

As with the DPA version, the DPA2s_cp 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).

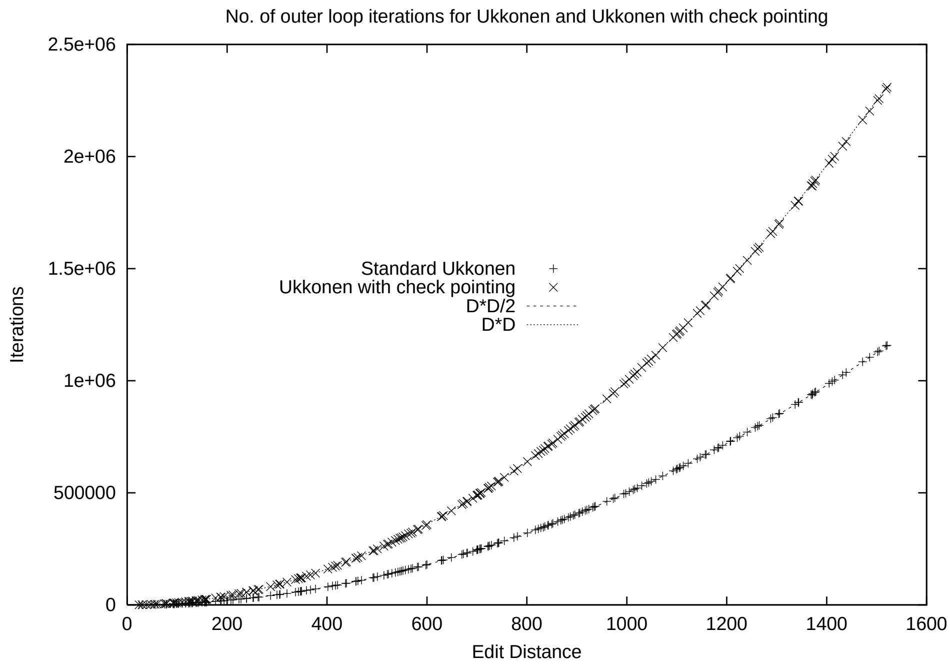

Figure 2.7: Comparison of iterations of the outer loop for the Ukk2s and Ukk2s_cp algorithms. String A was generated with length 5000, string B was mutated from this so will be of a similar, but slightly different length.

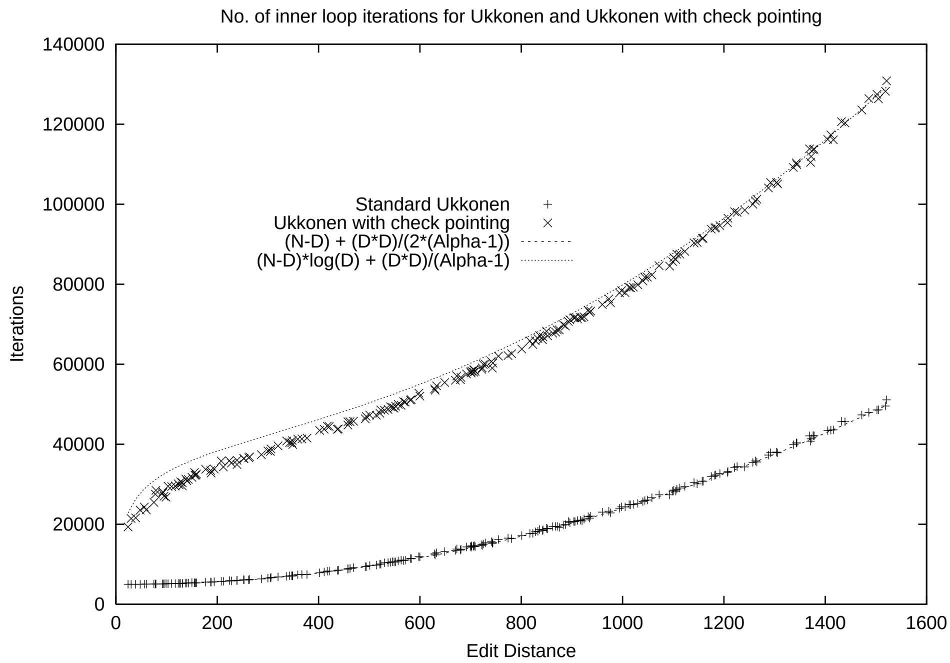

Figure 2.8: Comparison of iterations of the inner loop for the Ukk2s and Ukk2s_cp algorithms. String A was generated with length 5000, string B was mutated from this so will be of a similar, but slightly different length.

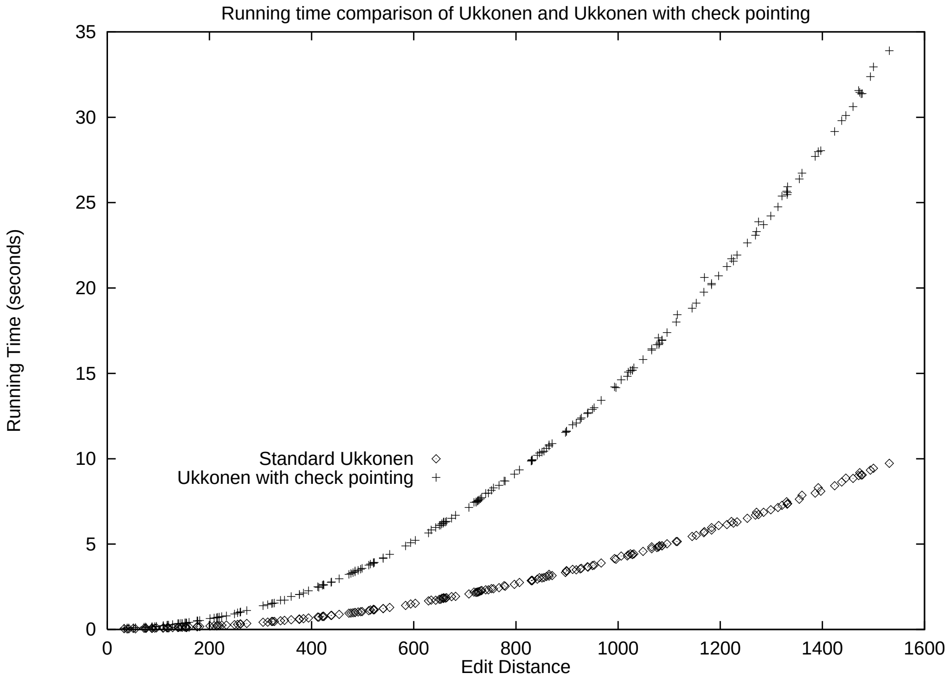

Figure 2.9: Actual running time of the Ukk2s algorithm and the Ukk2s_cp algorithm. String As was generated with length 5000, string Bs was mutated from this so will be of a similar, but slightly different length. (Timings done on a Pentium 100 with 16Mbytes RAM running Linux 2.0.0.)

2.5 Ukkonen's Algorithm with Linear Gap Costs

Ukkonen's algorithm has been applied to aligning strings using linear gap costs [39]. This algorithm shall be referred to as the Ukk2l algorithm. The check-point method can also be applied to this algorithm so an alignment can be determined in linear space, the resulting algorithm shall be called the Ukk2l_cp algorithm. The modifications to produce the Ukk2l_cp algorithm from the Ukk2l algorithm are very similar to the modifications to produce the Ukk2s_cp algorithm from the Ukk2s algorithm. Linear gap costs are charged as $a + bx$ where x is the length of a gap. As mentioned in Section 2.4.1 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 the DPA2l_cp algorithm, each cell of the matrix contains 3 values, one each 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 2.3.

Apart from these differences the Ukk2l_cp algorithm is the same the Ukk2s_cp algorithm.

This similarity of the check-point method when applied to different cost functions is a considerable advantage over the Hirschberg method employed by Myers [37]. Myers investigated only simple mutation and it would be difficult to extend the method he used to these more complex mutation costs. Whereas the check-point method could be used in a similar manner as presented here for even more complex costs such as piece-wise linear, and concave costs.

2.6 Aligning Three or More Sequences

It is possible to align three sequences optimally using the DPA3s algorithm discussed in Section 1.4.1. This algorithm requires $O(n^3)$ space to find an alignment. It is possible to reduce this space complexity to $O(n^2)$ by applying the check-point method producing an algorithm that shall be referred to as the DPA3s_cp algorithm. The DPA3s algorithm computes a three-dimensional cube consisting of a number of two-dimensional planes. Each plane is defined entirely in terms of the previous plane in the same way that each row of the DPA2s algorithm is defined by the previous row. Applying the check-point method to the

DPA3s algorithm requires the check-pointing of a *plane* rather than a row but is otherwise essentially the same the two sequence version.

It is possible to generalise the point mutation cost DPA to align k sequences using $O(n^k)$ time and space. It would be relatively straightforward to apply a generalised check-point method to this algorithm to reduce the space complexity to $O(n^{k-1})$. This generalisation of the DPA and check-point method is unlikely to be of any practical use, since k-sequence alignment using the standard DPA is impractical for reasonable length sequences when $k > 4$.

Allison [2] extended Ukkonen's algorithm to three sequences. This three sequence algorithm is referred to here as the Ukk3s algorithm. The Ukk3s algorithm has an average time complexity of $O(n + d^2)$ and space complexity of $O(d^3)$. It is possible to apply the check-point method to this algorithm reducing the space complexity to $O(d^2)$.

The extension of Ukkonen's algorithm to three sequences and linear gap costs produces an algorithm that shall be referred to as the Ukk3l algorithm. The Ukk3l algorithm is the major contribution of Chapter 3. This algorithm has a space complexity of $O(d^3)$ which can be reduce to $O(d^2)$ by the application of the check-point method resulting in the Ukk3l_cp algorithm. Due to the complicated nature of the Ukk3l algorithm, applying the check-point method is more difficult than for the other algorithms discussed here. Although it would be very difficult to imagine applying a Hirschberg-based method to the Ukk3l algorithm. The Ukk3l_cp algorithm is the major contribution of Chapter 4.

2.7 Conclusion

In this chapter the check-point method has been presented to modify alignment algorithms to produce alignments in linear space with little or no effect on the time complexity. This method had been previously applied to the basic DPA with simple costs. The advantages of this method over other methods were shown. A major advantage of this check-point method is that it is easier to adapt to more complex cost functions. Another advantage of the check-point method is that its application is similar even when applied to different alignment algorithms such as the DPA based algorithms and the Ukkonen based algorithms.

The check-point method also has the practical bonus of being able to run the program faster by using more check-points. This essentially gives the ability to trade memory usage for running-time.

It was also shown how to apply the check-point method to the fast algorithm by Ukkonen. This new alignment algorithm produces an alignment using $O(d)$ space and runs in $O(n \log_2 d + d^2)$ time on the average. Thus, for similar sequences, having the speed advantage of Ukkonen's algorithm while using only linear space.

Chapter 3

Aligning Three Sequences with Linear Gap Costs

3.1 Introduction

The work presented in this chapter has been published in the Journal of Theoretical Biology [43].

Alignment algorithms can be used to infer a relationship between sequences when the true relationship is unknown. Simple alignment algorithms use a cost function that gives a fixed cost to each possible point mutation — mismatch, deletion, insertion. These algorithms tend to find optimal alignments that have many small gaps. The biological processes that we are attempting to model when using alignment algorithms tend to involve mutations where "chunks" of the sequence are added or deleted at a time. The simple point mutation model is not a good fit for these processes. A better model is to use linear gap costs which give a start-up cost for a gap, then a lower cost to continue the gap. Therefore, alignment algorithms that use linear gap costs tend to favour a few, long gaps over many short gaps. For this reason linear gap costs are widely used when aligning biological sequences. Note that a point mutation model is used for modelling a change or mismatch.

In this chapter two new algorithms are presented to align optimally three sequences using linear gap costs. The first is based on the DPA for three sequences and shall be referred to as the DPA3l algorithm. The second algorithm is for the same alignment problem but is an extension of the Ukk2s algorithm for two sequences with simple costs by Ukkonen [62]. This new algorithm shall be referred to as the Ukk3l algorithm. For sequences of length n that have an optimal alignment with edit cost d the Ukk3l algorithm has a time complexity of

$O(d^3+n)$ on average, which is nearly always attained, and $O(nd^2)$ in the worst case. A finite state model is described for the generation of sequences from a parent sequence when using a linear function to cost a run of insertions or deletions. For this model it is shown how the probabilities of matches, changes, insertions and deletions in the finite state machine relate to the costs in the alignment algorithm.

In many circumstances it is important to align optimally more than two sequences. The obvious dynamic programming algorithm for k sequences requires $O(n^k)$ time to run which is infeasible for $k \geq 3$ with n of any reasonable length. Improved algorithms for k sequences have been studied and algorithms such as those by Carrillo and Lipman [11] and later by Altschul and Lipman [9] have been developed. These algorithms use the optimal alignment of pairs of sequences as a heuristic to limit the k-dimensional volume used to find an optimal alignment. The Carrillo and Lipman algorithm was discussed in Section 1.4.2.

For many applications, such as building of evolutionary trees [50] from sequences or in sequence assembly, it is desirable to align three sequences at a time. There are many multiple sequence alignment algorithms that use pairwise sequence alignment in an iterative method to build up a multiple alignment [41, 61, 58, 25]. Some algorithms such as MASCOT [26] use the extra information obtained from three-way alignment to improve the multiple alignment.

In an evolutionary tree the leaves of the tree are labeled with real, known sequences. The tree represents a possible evolutionary relationship between these sequences by the tree's internal structure. Every internal node of the tree has an associated inferred sequence and three neighbouring nodes or leaves. A three-way alignment algorithm is useful for making an inference of the sequence at each internal node by optimally aligning the sequences of the three neighbouring nodes or leaves. This process of inferring the internal nodes' sequences can be iterated until a stable solution is found. For this application it is important to have an efficient alignment algorithm. The use of linear gap costs in the alignments also produces better inferences of the sequences.

Gotoh [21] presented an algorithm for aligning three sequences with linear gap costs based on the simple alignment DPA. Gotoh's algorithm may be considered an approximation to the new DPA3l algorithm. An example alignment is shown that illustrates the different alignments that may be found with the new algorithms of this chapter.

To illustrate the usefulness of the new Ukkonen-based algorithm, Ukk3l, for three sequence over the DPA-based algorithm, DPA3l, programs were run implementing both algorithms on some real biological sequences. Three sequences were aligned using the Ukk3l algorithm in 15.5 CPU-minutes using about 1 Gigabyte of memory on a 1.2GHz AMD Athlon with

512MB of RAM. The program implementing the DPA-based algorithm, DPA3l, was unable to align these sequences since around 7 Gigabytes of memory would be needed and a projected run-time of over 2 CPU-hours. Although it would be possible to apply the check-point method of Chapter 2 to the DPA3l algorithm to reduce the memory requirement to around 11 Megabytes, the run-time would still be significantly longer than for the Ukk3l algorithm. The details of this example run is given in Section 3.5.

It may occur to the reader that if alignment of three strings is an improvement on two strings, why not four strings, or indeed k strings. Optimal alignment of k strings with a DPA type algorithm has a time complexity of $O(n^k)$ for strings of length n. This becomes prohibitive for long strings with k around 4 or above. The next improvement to consider would be Ukkonen's algorithm for k strings, which would be expected to have an average time complexity of $O(d^k + n)$ where the edit cost of the k strings is d. The problem here is that as the number of strings increases, the edit cost tends to increase also. Thus it is more likely that d will become larger than n, therefore Ukkonen's algorithm will be slower than the DPA version at some point. So, when aligning two strings with Levenshtein costs $d \leq n$, where n is the length of the longest sequence, Ukkonen's algorithm is always at least as fast as the DPA. However with three or more strings d may exceed n.

3.2 Alignment of Three Sequences

Ukkonen's algorithm for two sequences requires the numbering of the diagonals of the DPA matrix. Each cell of the D matrix is said to lie on the diagonal $ab = i - j$ where i and j are the indices of the cell. Figure 3.1 shows the numbering of some of the diagonals on the DPA matrix for two sequences of different length. In this example the final cell of the D matrix, $D[4, 7]$, lies on the diagonal $ab = -3$.

Extending this to three sequences is straightforward, although it can be difficult to visualize. In the three sequence DPA, Listing 1.5, the D matrix is a three dimensional cube with each cell having three indices, i, j and k. For the three sequence Ukkonen algorithm the diagonals of this cube are numbered with two indices, $ab = i-j$ and $ac = i-k$. The main diagonal of the D matrix is represented in Figure 3.2 for three sequences of equal length. This diagonal corresponds to $ab = 0$ and $ac = 0$. The numbering of the other diagonals in the three dimensional D matrix is shown in Figure 3.3. This figure is from the perspective of looking *along* the diagonal. The cells on the diagonals are labelled with their (i, j, k) reference as well the $diag(ab, ac)$ reference of the diagonal they lie on.

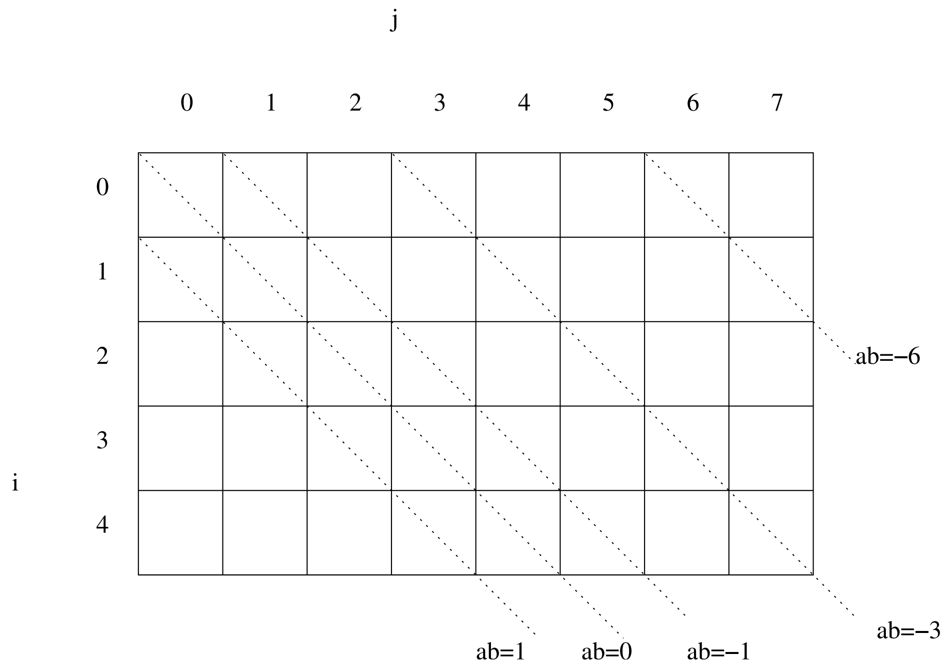

Figure 3.1: Numbering of diagonals of the D matrix for Ukkonen's algorithm for two sequences.

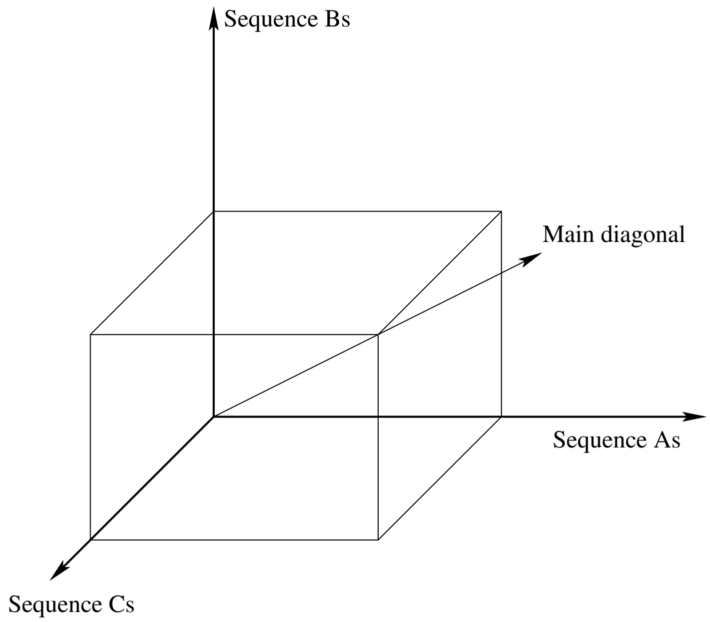

Figure 3.2: The main diagonal of a the D matrix for a three sequence algorithm.

3.3 Linear Gap Costs

The simple costs used in the basic DPA are not as biologically plausible as linear gap costs. Linear, also called affine, gap costs use a fixed mismatch (or change) cost, and a linear function for a run of insertions or deletions in the gap. The linear function gives a start-up cost to a gap then a lower cost for each character in the gap, thus a few long gaps are favoured over many short ones. This provides a better model for biological mutations than simple costs does.

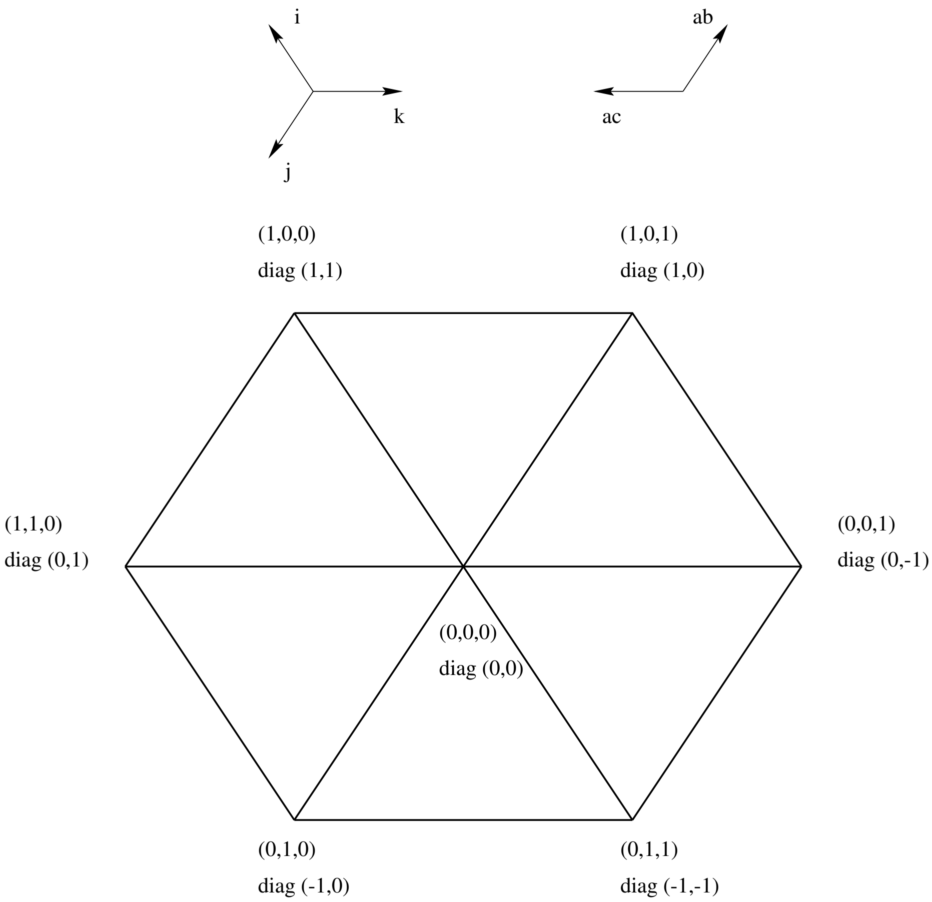

Figure 3.3: Numbering of diagonals for three-way alignment with Ukkonen's algorithm

It is important when considering alignment problems to be clear on the mutation model being used. Figure 3.4 shows two machines. The first machine is a mutation machine which takes a sequence, $S1$, and a set of mutation instructions as input, and produces a second sequence, $S2$, by applying those instructions to $S1$. The second machine, called a generation machine, takes a set of instructions as input and produces the two sequences. Note that the generation machine takes different types of instructions to the mutation machine. The mutation machine models the process of mutating one sequence into another. The generation machine produces both sequences from the instructions. While these machines may appear to be different, they can be thought of as equivalent since the generation machine can made to behave like the mutation machine by embedding sequence $S1$, in its instructions. The rest of this chapter shall use the mutation machine when discussing the mutation model.

This chapter concentrates on star-costs for three-sequence alignment. In terms of mutation machines this corresponds to having three separate machine all with the same instruction probabilities. Each machine is fed the same sequence, $S1$, but different instructions. The alignment problem is to infer $S1$ and the instructions fed to each machine from the three observed sequences.

In Chapter 5 the problem is considered where the input sequence $S1$ comes from a known sequence family, however $S1$ itself is unknown. What is known is two sequences each of which are the result of passing $S1$ through the above mutation machine. The problem is then to infer $S1$ and the mutation instructions given to the two mutation machines. This is different from the standard alignment view where only one mutation machine is considered.

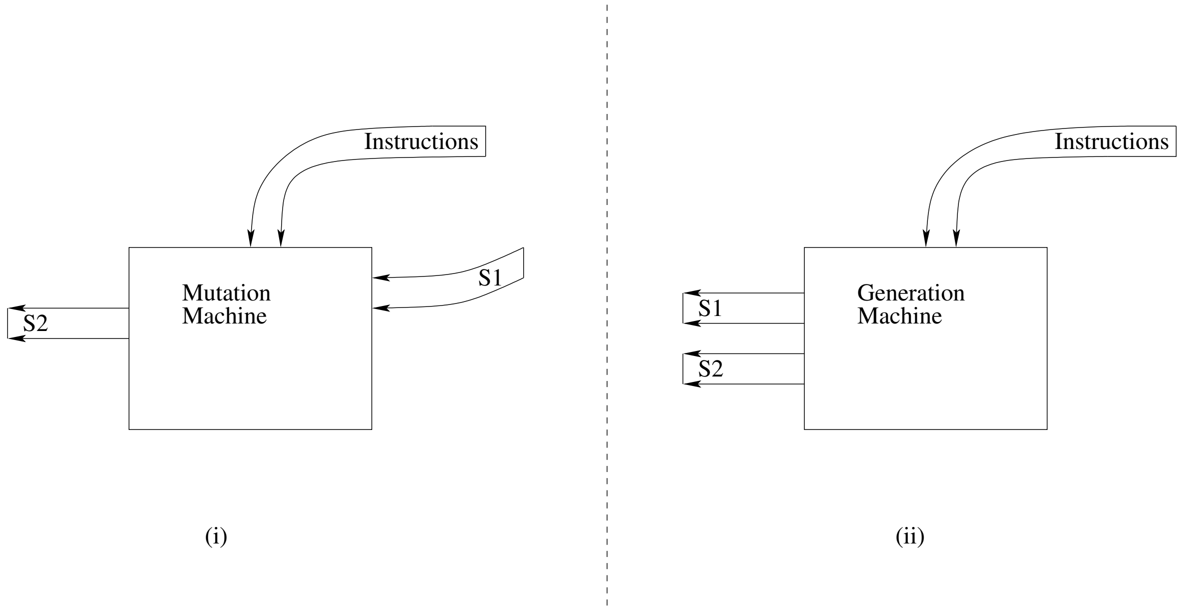

Figure 3.4: A mutation machine and a generation machine.

The instructions that are fed to the mutation machine determine the mutation model. For linear gap costs, the mutation instruction can be thought of as being generated by a finite state machine (FSM) with three states as in Figure 3.5. Note that `M` will be used to denote the *match/change* state, `I` to denote the *insert* state, and `D` to denote the *delete* state. The FSM can be seen as processing an input sequence a character at a time, and producing an output sequence a character at a time. Every transition into the *delete* state uses a character from the input sequence without producing any characters in the output sequence. Transitions into the *insert* state produce a character on the output (chosen from some distribution) without affecting the input. The "match/change" state takes a character from the input sequence and generates a character in the output sequence. Whether the character is copied correctly or mutated is determined with probability P_{change}.

It is normal when using linear gap costs to have the probability of continuing a gap higher than starting one. That is, $P(Ins|I) > P(Ins|M)$ and $P(Del|D) > P(Del|M)$.

Often it is more convenient to use *costs* instead of probabilities in alignment algorithms. Costs can be thought of as negative logarithms of the probabilities with multiplicative and additive constants. If the cost of a match (or copy) is set to zero, the other costs can be chosen from the probabilities in such a way as to leave the rank order of alignments unchanged [3]. It is common in the literature to have linear gap costs of the form $w(k) = a + b \times k$ for

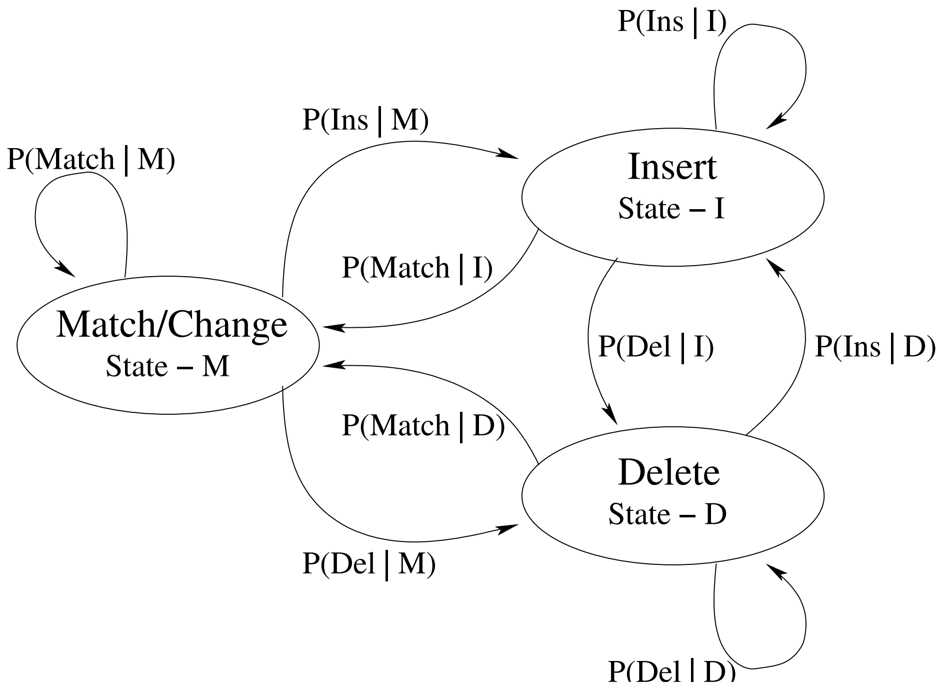

Figure 3.5: A 3-state Finite State Machine to produce one sequence from another.

a gap length of k ($k > 0$). The cost a is the cost of starting a gap, and b is the cost of continuing that gap (typical values, $a = 3$, $b = 1$). Hereafter, the costs in the FSMs are referred to in the form $C(Match|M)$. Using this notation: $C(Ins|M) = C(Ins|D) = a+b$ and $C(Ins|I) = b$; $C(Del|M) = C(Del|I) = a + b$ and $C(Del|D) = b$. It is necessary when using Ukkonen-based algorithms for matches to cost 0, thus: $C(Match|M) = C(Match|D) = C(Match|I) = 0$. And the mutation cost for copying a character incorrectly, $C(Change)$, is often set to 1. Modifying the costs to leave the rank ordering of alignments unchanged can be accomplished by multiplying all costs by a constant factor, or by adding (or subtracting) a constant value for each alignment character. This allows the costs to be modified to convenient small integers as required by Ukkonen's algorithm. This correspondence between probabilities and costs can be used to choose costs that match some desired probabilities, or the FSM probabilities can be calculated for commonly used alignment costs.

The mutation instructions fed to the mutation machine could be generated in many different ways. A five state FSM could be used which would correspond the piece-wise linear gap costs that are sometimes used in the literature. Even more complex models could be used, however these would lead to more complex alignment algorithms. This thesis shall limit the focus to the three state FSM for linear gap costs since this is the most commonly used model.

It is interesting to note that the probabilities, and hence the costs, used in these alignment algorithms are dependent only on the *instructions* to the mutation machine of Figure 3.4.

The sequences themselves are only important for whether two given sequence characters match or not. In Chapter 5 the model of mutation is extended to consider the probabilities of the characters of each sequence being context dependent. This amounts to having a model for the sequences whereas the standard alignment algorithms consider all sequences equally likely.

The probabilistic interpretation of alignment algorithms makes it clear when using linear gap costs that a geometric distribution is being used for the length of gaps. This is not an ideal distribution for the length of gaps, but it is practical because it leads to an efficient algorithm.

3.4 Alignment of Three Sequences with Linear Gap Costs

Alignment of three sequences with linear gap costs is a complicated extension of alignment for two sequences. There are three types of costs used when aligning more than two sequences: tree costs, star costs and all-pairs costs. For three sequences tree costs and star costs are the same. Star costs have an unknown "parent" sequence which the three known sequence are simultaneously aligned with. The final edit cost is the sum of the alignment costs of each of the three known sequences with this parent sequence. All-pairs costs, as the name suggests, determines the final edit cost as the sum the costs between each pair of sequences. The focus of this chapter is on star costs which, arguably, correspond to a real evolutionary process.

When using star costs, we consider that each of the three sequences has been generated independently from a common parent by a 3-state FSM. The alignment problem is to infer from the three sequences how they were generated, and thus the common parent sequence. Each of the three sequences has a corresponding FSM. So, to infer how the sequences were generated, the state each of the three FSMs, at every point of the alignment must also be determined. Note that if the sequences to be aligned are assumed to come from a common ancestor, then 3-way alignment, unlike 2-way alignment, can distinguish insertions from deletions and a different cost used if desired.

As in Section 3.3 `M`, `I` and `D` will be used to denote the match/change, insert and delete states respectively. Alignments will sometimes be shown with each character of the sequence subscripted by a `M`, `I` or `D` to show which state the FSM for that sequence is currently in. Consider the 3-way alignment of the sequences `xyyxz`, `xxz`, and `zz`:

Interpretation (i)

$$
\begin{array}{ccccc}
X_m & Y_i & Y_i & X_m & Z_m \\
| & & & | & | \\
X_m & -_m & -_m & X_m & Z_m \\
 & & & & | \\
Z_m & -_m & -_m & -_d & Z_m
\end{array}
$$

Interpretation (ii)

$$
\begin{array}{ccccc}
X_m & Y_m & Y_m & X_m & Z_m \\
| & & & | & | \\
X_m & -_d & -_d & X_m & Z_m \\
 & & & & | \\
Z_m & -_d & -_d & -_d & Z_m
\end{array}
$$

The first interpretation (i) infers the common parent sequence is `XXZ` while (ii) infers `XYYXZ`. Which is the more likely of these two depends on the probabilities in the FSMs. Recall that when one of the FSMs is in the insert state the other FSMs are idle. Thus in the first alignment above, the FSMs for the second and third sequence are idle while the two 'Y' characters are inserted in the first sequence.

At every point along the alignment, the state that each of the three FSMs is in must be inferred. Combining the three FSMs each having three states results in a FSM that has $3^3 = 27$ states. Some of these 27 possible states will never be inferred. For example, if a character in the parent sequence has been deleted from all three child sequences the corresponding states would be `DDD`, which could not be reasonably inferred. In fact it is necessary for at least one of the FSMs to be in the *match* state. The *insert* state is special because it generates characters without "using" the parent sequence, thus if a FSM is in the *insert* state, the other two FSMs can be considered frozen. Therefore it is sufficient to allow only one FSM to be in the *insert* at a time (for example, `IMI` is invalid). Ignoring the invalid, or useless combinations of states there are 16 remaining possible combinations of states, which are listed below.

```
MMM MMD MDM DMM MDD DMD DDM MMI
MIM IMM MDI DMI MID DIM IMD IDM
```

3.4.1 Using a DPA

Gotoh [21] presented an algorithm for the alignment of three sequences with linear gap costs. However, his algorithm only allowed combinations of states that contain at least

```
T G G T – – – C G A T G C T A G
| | | |
T G G T – – – A T G C T A G C T
| | | |       | | | | | | | | |
T G G T C T G A T G C T A G C T
```

Alignment from Gotoh's algorithm.

```
T G G T C – G A T G C T A G – –
| | | |       | | | | | | |
T G G T – – – A T G C T A G C T
| | | |       | | | | | | | | |
T G G T C T G A T G C T A G C T
```

Alignment from DPA3l and Ukk3l algorithms.

Figure 3.6: Example of a different alignment found by the Ukk3l algorithm and Gotoh's algorithm

two match states, which gives seven different combinations. These seven combination are: `MMM MMD MDM DMM MMI MIM IMM`. Gotoh's algorithm uses seven different matrices, one for each of these combinations. The computation performed on these 3D matrices is analogous to the 2D matrices for two sequences.

The new DPA for three sequences with linear gap costs is similar to Gotoh's, but allows for all plausible combinations of FSMs' states and thus better matches the model for the sequences. As an example consider the three sequences `ATGCTAGCT`, `CGATGCTAG` and `CTGATGCTAGCT` optimally aligned using the following costs: match cost 0, change cost 1, gap start-up cost 3, and continue gap cost 1. The optimal alignment of these sequences obtained by Gotoh's algorithm is shown in the first of the two alignments in Figure 3.6, and the alignment found by the new algorithm is shown second.

The optimal alignment from Gotoh's algorithm has an edit cost of 15, and the alignment from the new algorithm an edit cost of 14. The reason these different alignments are produced is because the new algorithm allows a run of inserts in the middle of a run of deletes, thus the first `T` in the third sequence is considered inserted. This is not possible with Gotoh's algorithm.

An optimal alignment for three sequences with linear gap costs can be found by using a DPA similar to that for two sequences. The main difference is that instead of three matrices, as for the DPA2l algorithm, there is a matrix for each of the 16 possible combinations of states of the three FSMs.

The heart of the algorithm is shown in Listing 3.1, where the three sequences to be aligned are known as `As`, `Bs` and `Cs`. This procedure calculates one cell in one matrix. The function `transitionCost` calculates the cost for changing FSM states, for example to go from the matrix for MID to the matrix for MDD would have a cost $C(Match|M) + C(Del|I) + C(Del|D)$. Remembering that when one of the FSMs is in the insert state the other two FSMs are considered “frozen”, so another example of a transition cost is the transition from MMD $\rightarrow$ MID which has a cost $C(Ins|M)$. The changeCost function simply calculates the cost for any change mutations. For example, if the states of the three FSMs were MDM and the corresponding characters were x-y, the cost would be $C(Change)$. If the FSMs' states were MMM and the characters xyz the cost would be $2 \times C(Change)$.

```
procedure DPAcalcCell(i, j, k, toMatrix)
    set i2,j2,k2 to index of neighbour
    bestCost = infinity

    for fromMatrix in AllMatrices
        cost = cell[i2,j2,k2,fromMatrix] +
               transitionCost(toMatrix, fromMatrix) +
                changeCost(toMatrix, As[i], Bs[j], Cs[k])

        bestCost = min(cost, bestCost)
    endfor
    cell[i,j,k,toMatrix] = bestCost
endproc.
```

Listing 3.1: Calculation of a cell for a DPA to find an optimal 3-way alignment with linear gap costs.

In Listing 3.1, the parameter toMatrix denotes the matrix to be calculated which is associated with a state for each of the FSM, such as MDM. Every possible transition is considered in turn moving from the fromMatrix matrix to the toMatrix matrix. A given cell $[i, j, k]$ in a given 3D matrix, toMatrix, is completely determined by the cells with index $[i2, j2, k2]$ from all matrices. The index $[i2, j2, k2]$ of this *neighbouring* cell is determined by the states of the FSMs for the matrix toMatrix . That is, a combination of states has one and only one neighbour in the 3D matrices. The following table lists the neighbours for some given FSMs' states if the cell to be calculated it at $[i, j, k]$. Note that if one of the FSMs is in the *insert* state the states of the other FSMs are irrelevant in determining the neighbour.

MMM	[i-1,j-1,k-1]
MDM	[i-1,j,k-1]
DDM	[i,j,k-1]
MIM	[i,j-1,k]
MID	[i,j-1,k]

To find the optimal alignment for three sequences with linear gap costs, the DPAcalcCell routine is called for each cell as shown in Listing 3.2. The edit cost is the cheapest cost in any cell from the 3D matrices at $[|As|, |Bs|, |Cs|]$. The optimal alignment is found by backtracking from here through choices made by the $min()$ function.

```
for i = 0 .. |As|
  for j = 0 .. |Bs|
    for k = 0 .. |Cs|
      for toMatrix in AllMatrices
        DPAcalcCell(i,j,k,toMatrix)
```

Listing 3.2: Calculating all cells for a DPA for 3-way alignment with linear gap costs.

3.4.2 Using Ukkonen's Algorithm

The DPA for three sequences with linear gap costs runs in $O(n^3)$ time, where the three sequences are of length approximately n. By employing Ukkonen's algorithm this can be reduced to $O(nd^2)$ in the worst-case, and $O(d^3 + n)$ on average for sequences that have an edit cost of d.

As with the DPA base algorithm, there are 27 matrices, one for each of the possible combinations of FSM states, which reduces to 16 matrices in practice.

The DPA3l algorithm has a simple order in which to calculate cells of the D matrix — simply row by row as seen in Listing 3.2. The order in which to calculate cells of the U matrix is quite complicated. It is very difficult to determine an explicit ordering to calculate them. To solve this, the cells of the U matrix are simply calculated as needed, and the result stored to save re-computation. The function `Ukk`, in Listing 3.3, is called whenever a value is required from the U matrix. If the required cell has been computed before, it is simply returned from the U matrix. If it has not been computed yet the `UKKcalcCell` function is called to calculate the value of the cell. The result is then stored in case it is required again. The `Ukk` function is in essence a wrapper around the `UKKcalcCell` function, memo-ing the result of the computation.

The heart of the algorithm is the function `UKKcalcCell`, shown in Listing 3.4. This function is used to calculate the contents of a single cell and is analogous to the `DPAcalcCell` function in Section 3.4.1. The function `transitionCost` is as before, and the function `countUnique` returns the number of unique characters among its three parameters. To keep the ends of the sequences from being overrun, the function `pastEnd` is used to ensure each step is valid. Note that the `UKKcalcCell` function uses the `Ukk` function to find the required values of the U matrix, which may result in recursive calls to the `UKKcalcCell` function. This method of calculating the cells of the U matrix is a "lazy" evaluation method

```
function Ukk(ab, ac, d, toMatrix)
    if not withinMatix(ab,ac,d,toMatrix) then
        return -infinity

    if calculated(ab,ac,d,toMatrix) then
        return U[ab,ac,d,toMatrix]

    dist = UKKcalcCell(ab,ac,d,toMatrix)
    U[ab,ac,d,toMatrix] = dist

    return dist
endfunc.
```

Listing 3.3: The Ukk() function implementing a memo-array wrapper around the UKKcalc-Cell() function.

because cells are only calculated when they are required. The benefit of using this method is that it is far simpler than stating explicitly the order for calculating the cells.

Each cell is completely determined by its neighbour at $[ab1, ac1]$. The index of this neighbour is determined by the state corresponding to the cell being calculated. This neighbouring cell is analogous to the neighbouring cell in the DPA3l algorithm. A loop iterates over all possible matrices (recall, each matrix corresponds to a single state of the combined 16 state FSM) at the cell $[ab1, ac1]$. The variables da, db and dc contain the value 0 or 1. A value of 1 in da indicates that the matrix being calculated corresponds to a step along sequence As. If $da = 0$, then the matrix being considered is in a delete state for sequence `As` or one of the other sequences is in the insert state. Likewise for variables db and dc for sequences `Bs` and `Cs`.

The possible changes between the three sequences are dealt with in the `if then else` statement as follows: If the `MMM` matrix is being calculated, then there are two possible costs depending on the characters of the sequences being all different, or two being the same. If the step has a null character, '-', then the cost of the step can again take two values depending on whether the non-null characters are equal. The final possibility is that there is only one non-null character, in which case there can be only one cost.

If the cell to be calculated corresponds to the `MMM` matrix, then, in terms of the D matrix, there is the possibility of extending the diagonal along a run of matches. This is done in the `extendDiagonal` function, see Listing 3.5. This function also uses the `Ukk` function to find, or calculate as needed, the values in the matrices. The `extendDiagonal` function checks all matrices at `U[ab,ac,d]` to find which reaches furthest along the `As` sequence. From this point, if there is a run of matches, the distance is extended to the end of the run. The furthest distance is then returned. This way of extending the diagonal is a simple extension of that for the Ukk2l algorithm, but it does have some non-obvious complications which will

```
function UKKcalcCell(ab, ac, d, toMatrix)
    set da,db,dc to direction of neighbour (0 or 1)

    ab1 = ab-da+dc
    ac1 = ac-da+dc
    bestDist = -infinity

    for fromMatrix in AllMatrices
       cost = d-transitionCost(toMatrix, fromMatrix)

       if (toMatrix equals 'MMM') then
          cost = cost - changeCost

       a1 = Ukk(ab1, ac1, cost, fromMatrix)

       if (pastEnd(a1,a1-ab1,a1-ac1)) continue

       if (countUnique(if (da) then As[a1] else '-',
                       if (db) then Bs[a1-ab1] else '-',
                       if (dc) then Cs[a1-ac1] else '-')
                  = 2) then
          { x--  -x-  --x  xx-  x-x  -xx  xxy  xyx  yxx }

          fromCost = cost
          dist = a1+da
       else                          { unique=3 }
          { xy-  x-y  -xy  xyz }

          a1 = Ukk(ab1, ac1, cost-changeCost, fromMatrix)
          fromCost = cost-changeCost
          dist = a1+da
       endif

       bestDist = max(bestDist, dist)

    endfor

    { Make sure it is an improvement }
    bestDist = max(bestDist, Ukk(ab, ac, d-1, toMatrix))

    if (toMatrix equals 'MMM') then
       bestDist = extendDiagonal(ab, ac, d)

    return bestDist
endfunc.
```

Listing 3.4: Calculation of a cell for Ukkonen's algorithm with three sequences and linear gap costs.

be discussed further in the next chapter.

To extract the optimal alignment, the U matrix must be back-traced through the choices made in the `max()` function. The whole U matrix is required for this, thus $O(d^3)$ space is needed. If only the edit cost is required then the algorithms needs only $O(d^2)$ space. Indeed, it is possible to obtain an alignment using only $O(d^2)$ space using the check-point method of Chapter 2 as will be shown in Chapter 4.

```
function extendDiagonal(ab, ac, d)
    bestDist = -1

    { Find the furthest distance of matrices U[ab,ac,d] }
    for matrix in AllMatrices
        bestDist = max(bestDist, Ukk(ab, ac, d, matrix)

    { Try an extend this distance along a run of matches }
    while ( As[bestDist] == Bs[bestDist] == Cs[bestDist] )
        bestDist = bestDist + 1

    return bestDist
endfunc.
```

Listing 3.5: The extendDiagonal() function used in the UKKcalcCell function.

3.5 Results

The program implementing the 3-sequence alignment with Ukkonen's algorithm and linear gap costs is fairly complex. To ensure the correctness of the program extensive testing was performed. Some of the more rigorous testing procedures are explained below.

The first test was to compare that the results of the program with the results of a program implementing the DPA version of the algorithm. The three input sequences were independent random sequences of random length over an alphabet of four characters. This test was performed for several thousand different alignments with sequence length varying from zero to about 2000, while the edit cost ranged from 0 to about 150. The edit costs produced by both algorithms were the same.

The second test was to take the alignment produced by the Ukkonen version of the program, and from that determine the inferred parent sequence. This parent sequence was then aligned with each of the three input sequences one at a time. The pairwise alignment was performed with a program implementing the DPA2l algorithm. As before, this test was performed several thousand times with similar ranges of lengths and edit costs as for the previous test. The sum of the three edit costs from the pairwise alignments was equal to the edit cost from the three way alignment.

To illustrate the usefulness of the new Ukkonen-based algorithm for three sequences over the DPA-based algorithm, the programs implementing both algorithms were run on some real biological sequences. The DNA sequences that were selected were from the Transthyretin gene for a human, a mouse and a rat. These sequences were aligned with the Ukk3l algorithm in 15.5 CPU-minutes on a 1.2GHz AMD Athlon with 512 Megabytes of memory.

About 1 Gigabyte of memory was used by the program, however 1.2 Gigabytes were allocated. Memory allocation is discussed in the next section. These sequences could not be aligned using the DPA-based algorithm since around 7 Gigabytes of memory would be needed with a projected runtime of over 2 CPU-hours. However, if it were possible to align these sequences with the DPA3l algorithm, it would find the same optimal alignment cost as the Ukk3l algorithm. Further details of these sequences, and the actual alignment, are given in in Appendix A. The advantage of linear gap costs over simple costs can be seen by comparing this alignment with the alignment in Appendix B which is for the same sequence but with Levenshtein costs. It is also useful to compare the three-way alignment with the three pair-wise alignments which are shown in Appendix C.

The time complexity of the algorithm presented in Section 3.4.2 is not obvious. The algorithm exhibits an average time complexity of $O(d^3+n)$. Figure 3.7 shows the number of calls to the `UKKcalcCell` function which asymptotes to d^3. The inner loop iterations are along runs of matches, figure 3.8 shows this to be less that $O(d^3)$. The 'n' comes from the fact that for similar sequences of length n there are $O(n)$ matches along the optimal alignment. The rest of the iteration in the inner loop comes from matches off the optimal alignment.

A log-log plot of the actual running time against edit cost is shown in figure 3.9. From this it can be seen the running time also asymptotes to d^3.

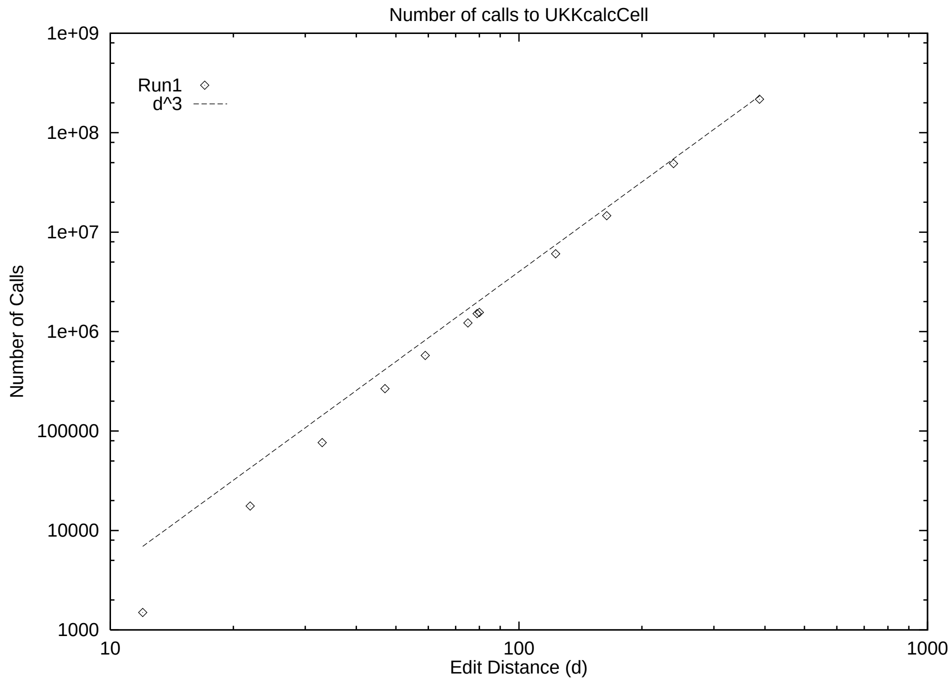

Figure 3.7: Log-log plot of the number of calls to UKKcalcCell against edit cost.

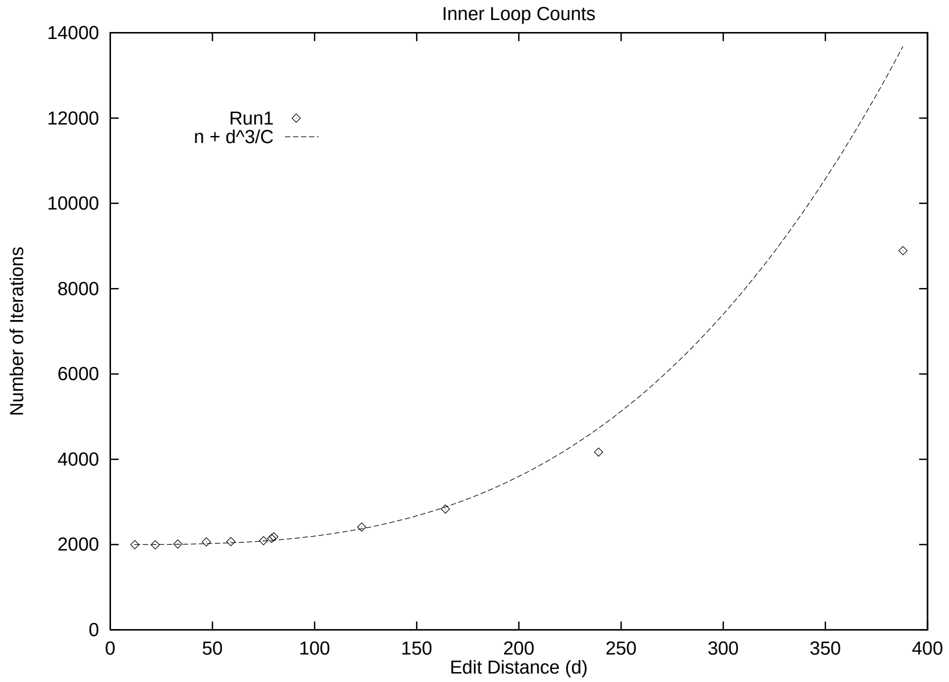

Figure 3.8: Plot of the inner loop iterations against edit cost for sequences of approximately 2000 characters.

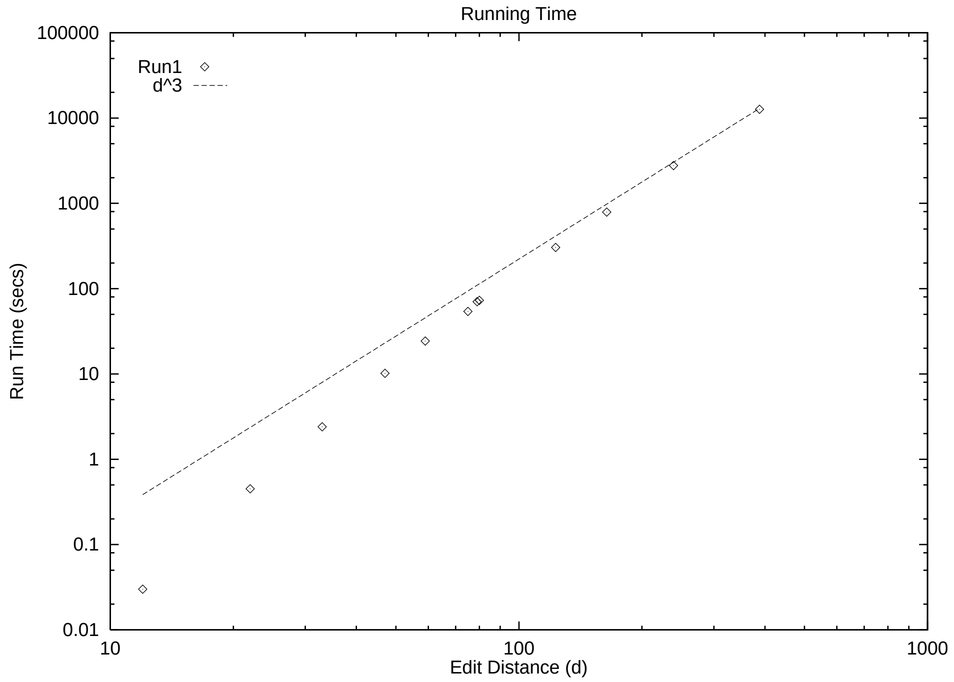

Figure 3.9: Log-log plot of running time versus edit cost.

3.6 Memory Management

In the same way as the DPA3s algorithm can be thought to operate on a three dimensional cube, the Ukk3l algorithm operates on an octahedron. The octahedron is "skewed" if the sequences are of different lengths. If the naive memory allocation is used, a bounding cube of this octahedron must be allocated. This is shown in Figure 3.10.

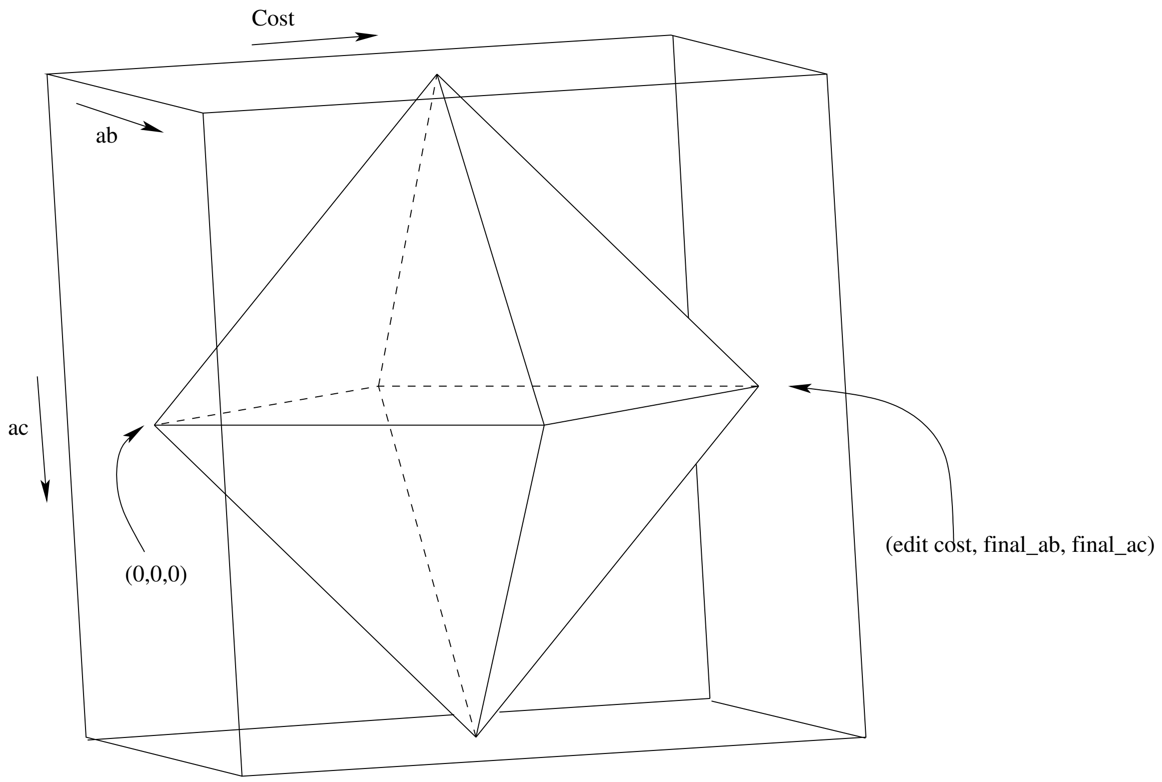

Figure 3.10: Visualisation of the memory needed by the Ukk3l algorithm (an octahedron) inside a bounding cube.

The algorithm to align three sequences using linear gap costs with Ukkonen's speed-up has a complicated memory access pattern. It is not straightforward to efficiently allocate memory so that only memory needed is allocated. A naive approach to memory allocation would be as follows: Since the U matrix is accessed as $U[ab, ac, d]$ a three dimensional "cube" could be allocated. It would be possible to allocate a plane of this cube as required. Each plane is of size $abSize \times acSize$, where for sequence lengths $|As|$, $|Bs|$ and $|Cs|$, $abSize = |As| + |Bs| + 1$ and $acSize = |As| + |Cs| + 1$, since ab can possible have values from $-|Bs|$ to $|As|$.

The above allocation strategy would be very wasteful because only a small portion of each plane would ever be used. For example, in the first plane, $d = 0$, only the cell for $ab = 0, ac = 0$ is used, and in the last plane, $d = finalCost$, only the cell for $ab = |As| - |Bs|, ac = |As| - |Cs|$ is used. A more efficient allocation strategy is difficult because the final edit cost is not known ahead of time, so the required size of each plane in the "cube" is not known. For simple costs it is possible, although difficult, to allocate the part of each plane that will be needed to compute up to the current edit cost. That is, instead of allocating the "cube" by planes in the ab,ac dimension, diagonal planes are allocated. For linear costs, however, this becomes very difficult.

Since it is wasteful to allocate a whole plane when only one cell is required, a different strategy was used for implementing the Ukk3l algorithm. Instead of allocating a whole plane at a time, small "square" was allocated around the cell. Thus each plane is split into squares and when space for a cell must be allocated, the whole square in the plane that the cell lies on is allocated. This method has the advantage of being fairly simple without wasting large amounts of memory.

The plot in Figure 3.11 shows the memory allocated by the Ukk3l program against edit cost. The memory allocated is an upper-bound on the actual memory used by the Ukk3l algorithm. It is clear from this data that the expected memory usage of $O(d^3)$ is achieved. The edit cost only version of the Ukk3l algorithm has expected space complexity of $O(d^2)$. The data plotted in Figure 3.12 confirms that the program implementing this algorithm achieves this complexity. The sequences used in producing this data were generated in the same manner as those in Section 3.5.

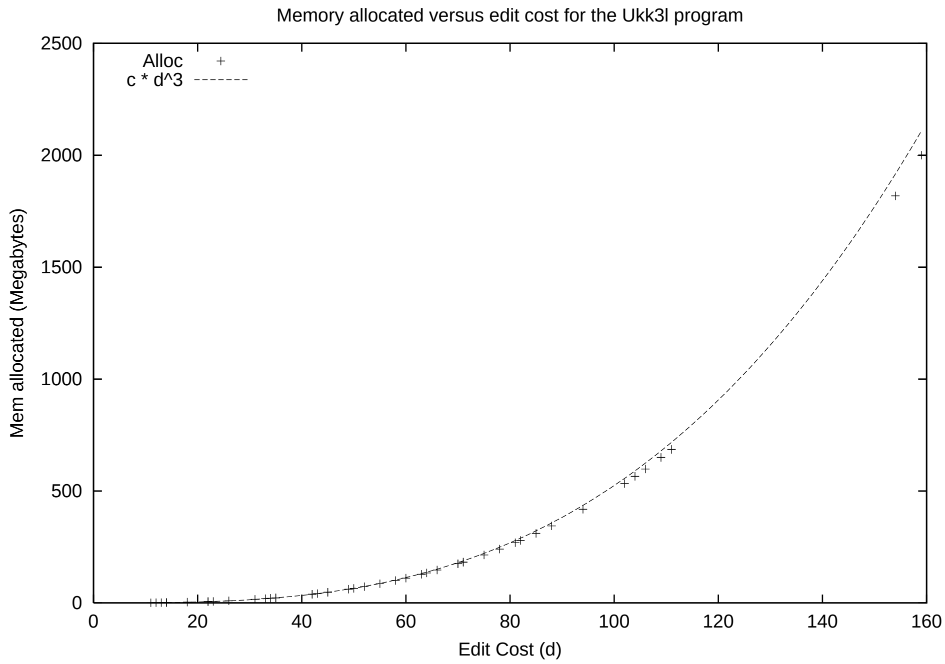

Figure 3.11: Plot of memory allocated versus edit cost for the Ukk3l program. Note 'c' is a constant.

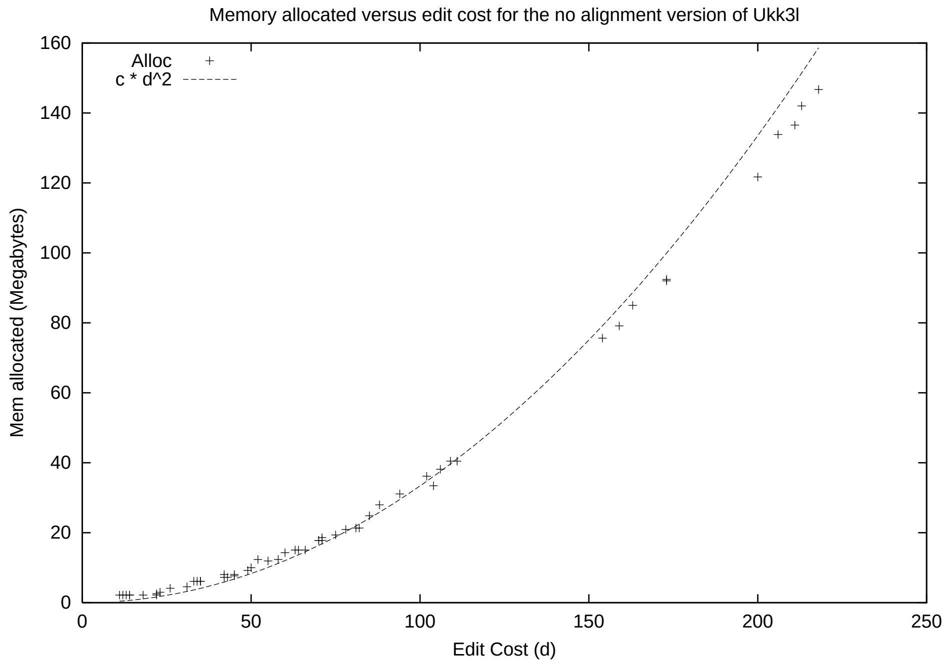

Figure 3.12: Plot of memory allocated versus edit cost for the Ukk3l program which does not recover an alignment. Note 'c' is a constant.

3.7 Conclusion

This chapter presented two new algorithms for optimally aligning three sequences with linear gap costs which have been published in a shorter form [43]. The first new algorithm, DPA3l, presented is based on the standard DPA. It was shown that this algorithm uses a generalisation of linear costs to three sequences under a star mutation model. This algorithm has complexity $O(n^3)$ in both time and space. The model of the three sequences from a common parent sequence was made clear, and it was shown how the costs used in the alignment algorithms relate to this model.

Using previously known algorithms as explanatory stepping-stones the second algorithm and major contribution of this chapter was presented: an algorithm for 3-string alignment using linear gap costs with the efficiency speed up of the Ukkonen algorithm [62]. This new algorithm produces the same results as the abovementioned DPA3l algorithm, however, the Ukk3l algorithm is significantly faster for similar sequences. The average time complexity of the was shown to be $O(d^3 + n)$ and use $O(d^3)$ space, or $O(d^2)$ space if only the edit cost is desired. An earlier alignment algorithm for three sequences with linear costs by Gotoh [21] was shown to produce inferior alignments than the DPA3l and Ukk3l algorithms.

Chapter 4

Check-Pointing on Ukkonen's Algorithm for Three sequences

4.1 Introduction

The Ukk3l algorithm presented in Chapter 3 exhibits space complexity of $O(d^3)$. It was shown that the memory usage of the Ukk3l program can grow large quickly. An example was given for part of a Transthyretin gene from a human, a mouse and a rat, which had an edit cost of 233 and required 1 Gigabyte of memory to align. In this chapter it is shown how to use the check-point method of Chapter 2 to reduce the space complexity of the Ukk3l algorithm to $O(d^2)$. This new algorithm shall be referred to as the Ukk3l_cp algorithm. The Ukk3l_cp algorithm is able to align the above Transthyretin sequences using only 273 Megabytes of memory (see Appendix A for further details). In Section 4.5 results of example runs are presented that show the Ukk3l_cp algorithm does indeed exhibit memory usage of $O(d^2)$.

The check-point method could also be used to reduce the space complexity of the DPA3l algorithm of Section 3.4.1 from $O(n^3)$ to $O(n^2)$. This is straightforward and warrants little explanation. Application of the check-point method to the Ukk3l algorithm is more difficult due to the complicated nature of the Ukk3l algorithm.

It is clear that if an alignment is not required, the Ukk3l algorithm can be reduced to $O(d^2)$ space complexity. This is done by only keeping the necessary cells in the U matrix. The array access $U[ab, ac, d]$ is modified to $U[ab, ac, d \bmod sliceSize]$ where $sliceSize$ is a constant of the number of slices of the U matrix that are required to be kept. This constant is a function of the maximum cost of a single edit operation.

Applying the check-point method to the Ukk3l algorithm is similar, but more complicated, than the application to the algorithms in Chapter 2. Visualisation of the U matrix can be difficult because of its three dimensional nature and octahedral shape. It can be helpful when considering the Ukk3l_cp algorithm to keep in mind the three dimensional DPA matrix and the numbering of the diagonals as shown in Figure 3.3. At first glance it may be thought that applying the check-point method to the Ukk3l algorithm would be as straightforward as applying it to the other algorithms in Chapter 2. There are however some difficult complications which shall be discussed.

Each stage of the Ukk3l_cp algorithm may be considered an alignment problem of its own with three sequences that happen to be subsequences of the main sequences. Further, this sub-alignment problem has a defined starting state and ending state. It is important when performing this sub-alignment that the ends of the subsequences are not overrun.

Another complication of the Ukk3l_cp algorithm is that it may find a different optimal alignment to the Ukk3l algorithm. This is not necessarily a problem since the Ukkonen style algorithms often find a different optimal alignment to the DPA style algorithms. Indeed, the Ukkonen style algorithms cannot, in general, find all the optimal alignments, while the DPA style can. This is due to the "greedy" nature of the Ukkonen algorithm which completely aligns a run of matches in one step. An example of an optimal alignment that cannot be found by a Ukkonen style algorithm is shown below for two sequences with Levenshtein costs, though it may be argued that this example is trivially different from the alignment the Ukk2s algorithm would find.

$$
\begin{array}{ccccc}
T & A & - & A & G \\
 & | & & | & \\
G & A & A & A & T
\end{array}
$$

4.2 The Ukk3l_cp Algorithm

First an overall description of the Ukk3l_cp algorithm shall be given, followed by the details of the modifications to the Ukk3l algorithm necessary to produce the Ukk3l_cp algorithm.

In the Ukk3l_cp algorithm not all of the U matrix is stored, only enough to calculate the subsequent cells. This is done as described earlier for the edit cost only version of the Ukk3l algorithm. That is, the U matrix accesses are modified from $U[ab, ac, d]$ to $U[ab, ac, d \bmod sliceSize]$ where $sliceSize$ is a constant dependent on the mutation costs.

At each recursive step the check-point cost, $CPcost$, is chosen to be half way between the starting and final cost for this step. However, on the first time through the final edit cost is unknown, so $CPcost$ is set to the cost d when $U[finalab, finalac, d] \geq |As|/2$. In other words, $CPcost$ is set to the cost when the half way point along As is reached on the final diagonal.

All cells of the U matrix, $U[ab, ac, d]$, are stored or check-pointed away if $CPcost \leq d \leq CPcost + sliceSize$, where $CPcost$ is the check-point cost and $sliceSize$ is the thickness of the U matrix. The check-point region is $sliceSize$ thick so that the U cells after the check-point *must* derive from at least one cell in the check-point region. Each cell of the U matrix carries information about which cell in the check-point region it derived from. The check-point cell lying on the optimal alignment can then be determined. At this point, two sub-alignment problems exist, the "start to the check-point cell" and the "check-point cell to the end".

4.2.1 The Details

The Ukk3l_cp algorithm is an extension of the Ukk3l algorithm. The heart of the Ukk3l_cp algorithm, the `UKKcalcCell()` function, is very similar to the one shown in Listing 3.4 for the Ukk3l algorithm. In the Ukk3l_cp algorithm the `pastEnd()` function is modified to ensure the ends of the *subsequences* are not overrun. The only change other than the U matrix indexing discussed earlier is to carry extra information in each cell regarding where in the check-point region it derived from. For each cell calculated, $U[ab, ac, d]$, if that cell is after the check-point region then information is stored in a `From` data structure for that cell containing which cell in the check-point region the $U[ab, ac, d]$ cell is derived from. It is simplest to put a `From` data structure in every cell of the U matrix.

The `Ukk()` function for the Ukk3l_cp algorithm is shown in Listing 4.1. This is a wrapper around the `UkkcalcCell()` function that stores the result in the U matrix to avoid unnecessary re-computation, in effect implementing a memo array. This function is similar to the one for the Ukk3l algorithm shown in Listing 3.3 except that the `Ukk()` function for the Ukk3l_cp algorithm also stores the check-point information if the cell just calculated is within the check-point region. The check-point region has a width of $CPwidth$, which is set equal to the maximum cost of a single alignment character.

The last important part of the Ukk3l_cp algorithm is the `UkkInLimits()` function which takes a starting cell and an ending cell and computes the alignment between these two points.

```
function Ukk(ab, ac, d, s)
    if not withinMatix(ab,ac,d,s) then
        return -infinity

    if calculated(ab,ac,d,s) then
        return U(ab,ac,d,s)

    dist = UKKcalcCell(ab,ac,d,s)
    U(ab,ac,d,s) = dist

    if (d >= CPcost) and (d < CPcost+CPwidth) then
        { This cell is in the check-point region}
        CP(ab,ac,d,s) = dist
    endif

    return dist
endfunc.
```

Listing 4.1: The Ukk() function for the Ukk3l_cp algorithm.

The `UkkInLimits()` function is shown in Listing 4.2. This function first checks whether the alignment is short enough to be determined directly, and if so, returns the alignment. Otherwise the new check-point region is set up to be half way between the start and end cost. Then the forward Ukkonen calculation is performed using the `Ukk()` function. The last cell contains the information describing from which cell in the check-point region it was derived. This `From` information is used to find the check-point cell and retrieve the distance stored therein. Recursion is then used from the starting cell to the check-point, and then from the check-point to the final cell. The two sub-alignments produced are concatenated and returned.

Each cell of the U matrix contains the distance that can be reached on a certain diagonal for a certain cost. The U matrix is reused for different edit costs. But clearing only the cells of the U matrix that are to be reused in the next step of the iteration would be complicated. As a practical solution to this problem, each cell of the U matrix contains the distance reached on As and the edit cost for which this distance was calculated. This provides an easy way to implement the `calculated()` function called in Listing 4.1. If the edit cost stored in the U matrix equals the cost to be calculated, then the cell has already been computed. This technique is also used in the edit cost only version of the Ukk3l algorithm.

A complication arises because the recursive steps of the Ukk3l_cp algorithm reuse the U matrix cells. Clearing the whole U matrix between recursive steps would be wasteful since many cells may be unused. Clearing only those cells that were used in the previous step would be difficult. A simple solution is to store the edit cost plus a $costOffset$ in each cell of the U matrix, then before each recursive call increase the $costOffset$ by the largest edit cost used.

```
function UkkInLimits(sab, sac, sCost, sState, sDist,
                     fab, fac, fCost, fState, fDist)

    if (fCost-sCost <= CPwidth) then
        { Base case. Can determine the alignment directly. }

        alignment = getAlignment(sab,sac,sCost,sState,sDist,
                                 fab, fac, fCost, fState, fDist)
        return alignment
    endif

    { Setup the check-point cost }
    CPcost = (fCost+sCost-CPwidth+1)/2;

    { Do the alignment calculation }
    Ukk(fab, fac, fCost, fState, fDist)

    { Extract the info from the last cell }
    fromInfo = From(fab,fac,fCost,fState)
    CPdist = CP(fromInfo.ab,fromInfo.ac,fromInfo.Cost,fromInfo.State)

    { Recursion for first half of the alignment }
    al1 = UkkInLimits(sab, sac, sCost, sState, sDist,
                      fromInfo.ab, fromInfo.ac, fromInfo.Cost,
                      fromInfo.State, CPdist)

    { Recursion for second half of the alignment }
    al2 = UkkInLimits(fromInfo.ab, fromInfo.ac, fromInfo.Cost,
                      fromInfo.State, CPdist,
                      fab, fac, fCost, fState, fDist)

    { Join the sub-alignments and return }
    return concat(al1,al2)
endfunc.
```

Listing 4.2: The UkkInLimits() function for the Ukk3l_cp algorithm.

4.3 Complications

4.3.1 The Free Transition

When using linear gap costs, a three-state FSM machine (see Figure 3.5) is used to model the sequence mutations. In three sequence alignment a combination for three of these three-state FSM is used, this is the *combined* FSM (see the end of Section 3.4). As was discussed for the combined FSM, each state transition is accompanied by the output of an alignment character. It is not necessary that this be the case for the Ukk3l algorithm, or indeed the Ukk2l algorithm. This shall be explained firstly for the simpler Ukk2l algorithm.

Recall in the Ukk2l algorithm there are three states in each cell of the Ukkonen matrix. One for the insert state corresponding to a vertical move in the DPA matrix, a state for deletions corresponding to a horizontal move, and a match/change state for the diagonal move. The Ukk2l algorithm is shown in Listing 1.4.

In the Ukk2l, as presented, it is possible to have a transition from an insert or delete state to the match state *without* any corresponding alignment characters. This transition shall be referred to as a *free transition* since there are no alignment characters output and the transition has zero cost. This free transition shall be illustrated by using code segments from the Ukk2l algorithm. Note that the code is shown here in a slightly different form than in Listing 1.4, but it is functionally equivalent.

The calculation for the insert state in the cell on the diagonal ab for a cost of d is:

```
U[ab, d, insertState] = max( U[ab+1, d-contGap           , insertState],
                             U[ab+1, d-startGap - contGap, deleteState],
                             U[ab+1, d-startGap - contGap, matchState])
```

The calculation for the delete state is similar. However, for the match state it is a little different. The following piece of pseudo-code shows this calculation. The first statement sets the initial distance in the match state cell, and the second part extends this distance while the two sequences to be aligned match.

```
U[ab, d, matchState] = max( U[ab, d-changeCost, matchState] + 1,
                            U[ab, d          , deleteState],
                            U[ab, d          , insertState])

while (As[U[ab, d, matchState] + 1] = Bs[ U[ab, d, matchState] - ab + 1])
   U[ab, d, matchState] += 1
```

Note that the match state is calculated from the insert and delete states at a cost of d, not $d - changeCost$. This comes about because of the way a diagonal is extended along a run of matches. The distance in the match state is taken and extended as far as possible. If the diagonal is not extended at all, the match state of the U matrix may contain an unexpected value if it were compared to the D matrix.

It is possible to remove this free transition from the algorithm, and therefore ensure consistency between the U matrix and the D matrix. This is accomplished by calculating the cell $U[ab, d, matchState]$ only using the values from the insert or delete state *if* they start a run of matches. However, the Ukk2l algorithm works with the free transition, and is simpler with the free transition than without it.

The Ukk3l and Ukk3l_cp algorithms have a free transition that is analogous to the one in the Ukk2l algorithm. In the Ukk3l and Ukk3l_cp algorithms this free transition allows the `MMM` state to take the value of that state that is furthest along As without extending the diagonal any further. It is important to note that this may cause values in the `MMM` state to be different from what would be expected if one were to look at the corresponding `MMM` state in the D matrix of the DPA3l algorithm.

It is possible to remove the free transition from the Ukk3l and Ukk3l_cp algorithms. This is done by taking each state in the cell $U[ab, ac, d]$ and attempting to extend each one along a run of matches. Only those states that are able to be extended along a run of matches are considered. The value that reaches furthest is put into the `MMM` state if it is larger than the current value in the `MMM` state. This complicates the algorithm somewhat but does ensure consistency between the U matrix representation and the D matrix representation.

The programs implementing the Ukk3l and Ukk3l_cp algorithms contain the free transition because it is simpler.

4.3.2 The Back-loop

In the Ukk3s algorithm (presented in [2]) there is a loop to look behind the current alignment characters. This is only necessary for steps which correspond to alignment characters of the form $< x, y, - >$ which has a deletion and a change. For Levenshtein costs this step would cost 2 while it may be possible to step from the same diagonal to the current diagonal for a cost of 1 if the characters *behind* these were $< z, z, - >$.

This so-called *back-loop* is to look behind the front alignment characters only if they contain a change as well as a deletion. The back-loop terminates when either; characters are found without a change; or the loop goes as far back as the distance contained in the preceding cell of the step off diagonal; or it is clear this step will not be as good as another onto the diagonal being calculated.

It appears that the back-loop is unnecessary in the Ukk3s algorithm. While some optimal alignments will only be found through the use of the back-loop, extensive testing has shown there always to exist another optimal alignment that can be found without using the back-loop. An example of an optimal alignment using Levenshtein costs that can only be found using the back-loop is:

$$\begin{array}{ccc} \mathrm{T} & \mathrm{G} & \mathrm{C} \\ | & & | \\ \mathrm{T} & \mathrm{T} & \mathrm{C} \\ & | & | \\ - & \mathrm{T} & \mathrm{C} \end{array}$$

and the optimal alignment found without back-loop:

$$\begin{array}{ccc} \mathrm{T} & \mathrm{G} & \mathrm{C} \\ | & & | \\ \mathrm{T} & \mathrm{T} & \mathrm{C} \\ | & & | \\ \mathrm{T} & - & \mathrm{C} \end{array}$$

Both these alignments have a cost of 2. While it seems the back-loop is unnecessary in the Ukk3s algorithm this is yet to be proven.

There is an analogous possible back-loop in the Ukk3l and Ukk3l_cp algorithms. This back-loop may only be used when determining a state that contains a delete and a mismatch, that is when calculating one of these three states: `MMD`, `MDM`, `DMM`. Consider state `MMD` and the characters aligned $As[i]$ and $Bs[j]$. If $As[i] \neq Bs[j]$ then the back-loop examines $As[i-1]$ and $Bs[j-1]$. If these are different it continues back until matching characters are found (that is, the alignment character contains a match and a delete), or until it is guaranteed that the step from this state would not be as good as another.

Like the free transition of the previous section, omitting the back-loop from the Ukk3l and Ukk3l_cp algorithms can lead to discrepancies between the U matrix and the D matrix of the corresponding DPA algorithm. This is not a problem because the Ukkonen style algorithm seems to always find an optimal alignment, but it can be important when comparing the D and U matrices.

The programs implementing both the Ukk3l and the Ukk3l_cp algorithms have both been coded with and without this back-loop. Over many extensive test runs no cases have been detected where the back-loop is required. As with the Ukk3s algorithm, it is thought that the back-loop is not required to find an optimal alignment for the Ukk3l or Ukk3l_cp algorithm.

4.3.3 Correspondence Between the U and D matrices

When using Ukkonen's algorithm on three sequences, some cells of the U matrix do not correspond to anything from the D matrix. A cell in the U matrix specifies how far along a certain diagonal of the D matrix can reached for a cost d, but there may not be any cells in the D matrix on that diagonal that have a cost d. As a simple example of this, consider aligning the sequences `ATA`, `ACA` and `AGA` with Levenshtein costs. The U matrix cell for the center diagonal at a cost of 1 is $U[0, 0, 1]$. This U matrix cell does not correspond to anything in the D matrix of the DPA3s algorithm. The question arises as to what to store in this cell of the U matrix. There seem to be two possibilities. One is to have the U matrix cell contain negative infinity ($U[0, 0, 1] = -\infty$ for this example), meaning that for this diagonal there is no distance along sequence As that can be exactly reached for exactly that cost. The second possibility is to have the U matrix store for diagonal ab, and a cost d, the *furthest* that can be reached on ab for a cost of *at most* d ($U[0, 0, 1] = 0$ for the above example).

The three sequence Ukkonen style algorithms can be made to work for either of these two possibilities. The simpler is to store in the U matrix how far can be reached on a diagonal for *at most* cost d. If the U matrix cell for cost d does not have a corresponding cell in the D matrix, then the value from the U matrix for cost $d - 1$ on the same diagonal is used.

4.4 Testing

The Ukk3l_cp algorithm is complex, and the program implementing it required extensive testing. The tests used to ensure correctness of the program are explained here. Most of these tests are the same as were used to test the Ukk3l program implementation of Chapter 3.

As a first check, the Ukk3l_cp program takes the alignment it produces and performs some "sanity" checks. Firstly, the three input sequences must occur exactly in each line of the alignment after the null '-' characters are removed. Secondly, the alignment's cost is recalculated and this cost should match the edit cost calculated by the Ukk3l_cp program.

Another test used is to take the alignment produced by the Ukk3l_cp program and choose a parent sequence consistent with this alignment. This parent sequence is then aligned with each of the three sequences in turn using the DPA2l program. The three edit costs from these pairwise alignment are summed together and must match the edit cost produced by the Ukk3l_cp program.

Another easy test is to use the Ukk3l_cp program to calculate the edit cost of three sequences, and then ensure this edit cost is the same as calculated by either or both of the Ukk3l and DPA3l programs. All these tests were performed using random input sequences of lengths varying from zero to a couple of thousand, varying edit costs, and varying mutation costs over many thousands of test runs.

4.5 Time and Space Usage

To show the time and space behaviour of the Ukk3l_cp algorithm, timing runs were performed. Three different programs were compared; a program implementing Ukk3l_cp, a program implementing Ukk3l, and a program implementing the Ukk3l algorithm for edit cost only. These programs are referred to in the plots as ‘checkp’, ‘alloc’ and ‘noalign’ respectively. The ‘alloc’ program failed to run for large edit costs because of insufficient computer memory. This is why the ‘noalign’ program is included in the comparison. It performs the same computation as the ‘alloc’ program — without returning the alignment — but uses far less memory.

The sequence data was generated in a manner that approximately controlled the edit cost between the sequences. Firstly one sequence was generated to be 2000 characters long, each character chosen to be uniformly random from an alphabet of 26 characters. Two copies were made of this sequence and each mutated until the edit cost between the three sequences was approximately the desired edit cost. Sequences were then generated for increasing edit costs. There were five runs for each edit cost, and each of the three programs were run on the same sequence data.

For determining the time complexity of the algorithm, the important factor is the number of cells of the U matrix that are calculated. This is the same as the number of calls to the `UKKcalcCell` function. The number of times this function was called is shown versus edit cost in a log-log plot for the three programs in Figure 4.1. From this it can be seen that the number of function calls for the ‘alloc’ and ‘noalign’ programs were the same. And the number of calls in the ‘checkp’ program was around fifty percent more. The three programs have $O(d^3)$ calls to the `UKKcalcCell` function.

The loop for extending diagonals in the `UKKcalcCell` function is also important. Figure 4.2 shows the number of iterations of this loop versus edit cost. As expected the number of iterations for the ‘alloc’ and ‘noalign’ programs are the same $O(n)$ (actually $L \approx n + \frac{d^3}{8 \times 26^2}$). The number of iterations for the ‘checkp’ program is a little erratic, though it tends to follow the expected $O(n \log d)$ behaviour. So the exhibited time complexity behaviour of the Ukk3l_cp algorithm is $O(d^3 + n \log d)$ as expected.

Figure 4.3 shows a log-log plot of the memory used for the three programs versus edit cost. The ‘alloc’ program uses $O(d^3)$ memory as expected. It can be seen that the ‘noalign’ and ‘checkp’ programs both use $O(d^2)$ memory, with the ‘checkp’ program using almost exactly three times as much memory as the ‘noalign’ program. The factor of the three comes from

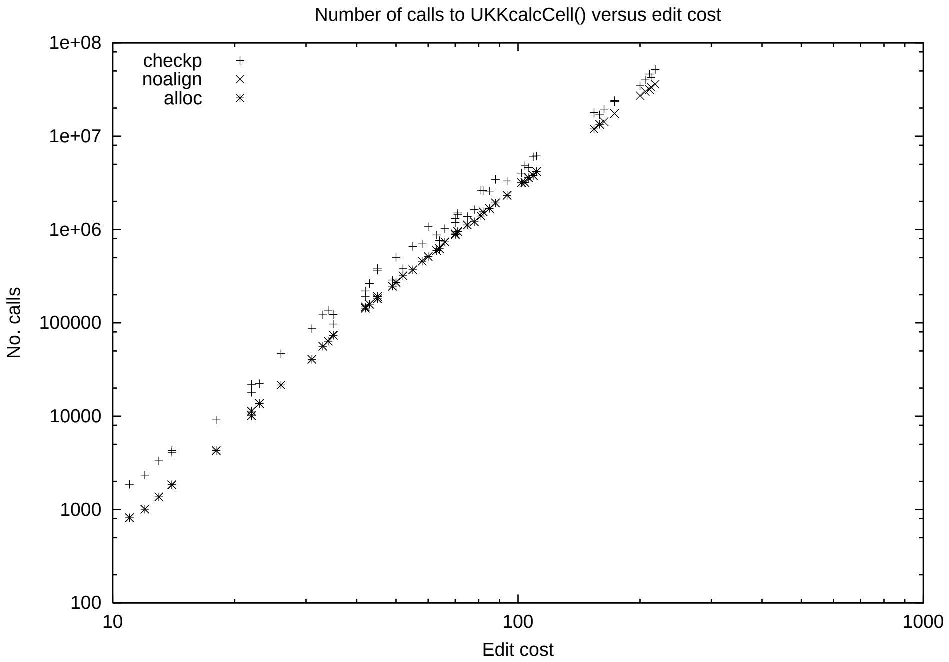

Figure 4.1: Log-log plot of the number of calls to `UKKcalcCell` against edit distance.

the extra information that must be stored by the 'checkp' program. Part of the extra storage for the 'checkp' program is used to store the check-point information. This increases the storage needs of the 'checkp' program by a factor of two over the 'noalign' program. The remaining storage is for the base case when the alignment is to be determined directly. In this case, extra information is stored per cell indicating which cell it derived from in the forward calculation simplifying the recovering of the alignment. This "back-trace" information is only needed in the base case. However, to simplify implementation, the storage for this information is allocated even when not performing the base case. This accounts for the rest of the increased memory usage of the 'checkp' program over the 'noalign' program.

4.6 Conclusion

Linear gap costs are often a better model for aligning sequences than simple costs. In fact, simple costs are a special case of linear gap costs. Optimal alignment of three sequences simultaneously generally leads to better alignments than pair-wise alignments. A tool for multiple alignment of proteins, MASCOT [26], uses three-way alignment instead of the traditional pair-wise alignment because three sequence alignments are more reliable. Gotoh's algorithm [21] aligns three sequences with linear gap costs using a generalization of the DPA algorithm. The Ukk3l algorithm presented in Chapter 3 improves upon this by correctly generalizing linear costs to three sequence and by using a Ukkonen style algorithm to align similar sequences much more quickly.

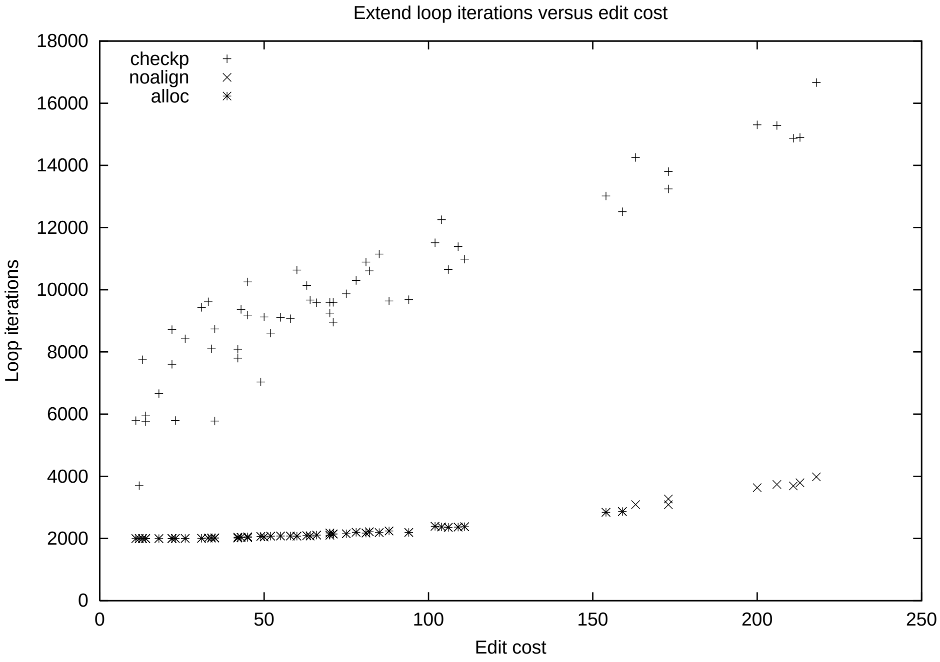

Figure 4.2: Plot of the inner loop iterations against edit distance for sequences of approximately 2000 characters.

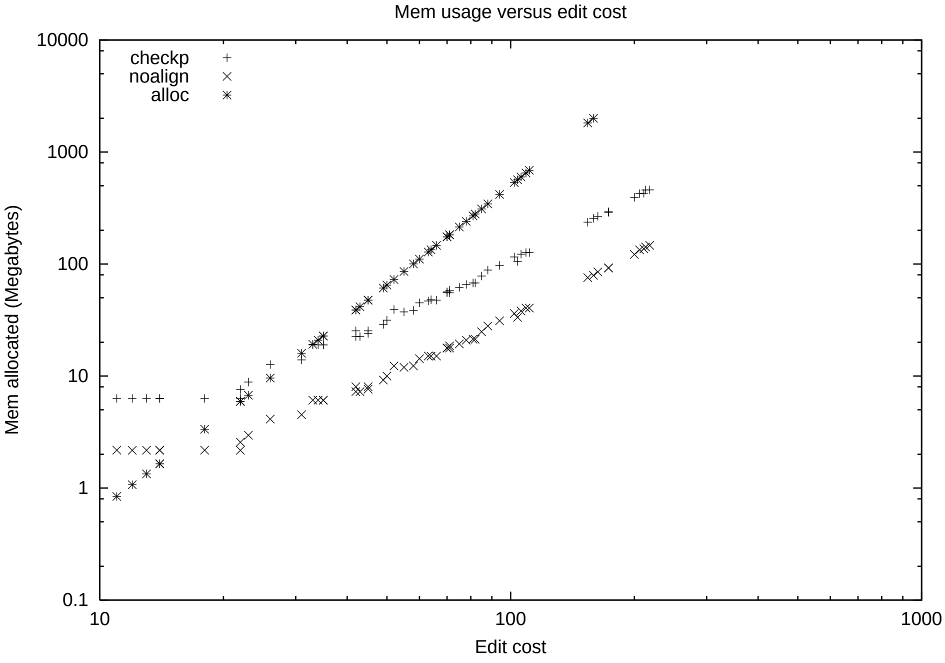

Figure 4.3: Log-log plot of memory usage versus edit distance.

This chapter developed a further improvement on the Ukk3l algorithm by incorporating the check-point method presented in Chapter 2 culminating in the Ukk3l_cp algorithm. This new algorithm has the speed advantages of Ukk3l algorithm for similar sequences while also requiring less memory. The memory usage of the Ukk3l_cp algorithm was shown to be $O(d^2)$ compared to $O(d^3)$ for the Ukk3l algorithm. This improvement comes at the cost of a small slow-down of the algorithm. However, in some circumstances the Ukk3l_cp program may actually be faster than the Ukk3l program. This may occur if the alignment problem caused the Ukk3l program to need more memory than was available, causing many virtual memory page swaps to and from disk. The reduced memory requirements of the Ukk3l_cp program may allow such an alignment problem to be solved using only the available RAM. In such circumstances the Ukk3l_cp program may indeed execute faster than the Ukk3l program. The Ukk3l_cp algorithm was shown to have an average time complexity of $O(d^3 + n \log d)$ compared to the Ukk3l algorithm which has an average time complexity of $O(d^3 + n)$.

The development of the Ukk3l_cp algorithm is a good example of the usefulness of the checkpoint method presented in Chapter 2. Attempting to apply the Hirschberg [27] forward-backward method to the Ukk3l algorithm would be extremely difficult.

Chapter 5

Alignment of Low Information Sequences

5.1 Introduction

The alignment algorithms discussed in preceding chapters have been for random sequences over a fixed alphabet. 'Random sequences' in this sense means that the sequences contain no structure. Alternatively, all parts of a sequence carry the same amount of information and thus the sequences are incompressible. Under the assumption that the sequences are uniformly random, the previously discussed DPA and Ukkonen based alignment algorithms are known to find an optimal alignment. However, DNA and many other sequences are not completely random and unstructured, and the issue arises of how to best align compressible sequences. In this chapter non-random sequences are considered and an algorithm for optimally aligning them is developed. Additionally, an algorithm to produce a measure for the relatedness of such sequences is also developed.

Typically alignment models try to maximize the number of matches and minimize the number of mutations. The edit distance criterion is an example of this. There is generally the tacit assumption that the sequences themselves are random (that is, incompressible). The compressibility of non-random sequences should be taken into account when aligning them. For example, consider sequences in which runs of repeated letters are common. The figure below shows a low information region (`AAA`) and a high information region (`AGT`) and partial matches that might occur in alignments:

```
. . . A A A . . .      . . . A G T . . .
      | | |                  | | |
. . . A A A . . .      . . . A G T . . .
```

The second partial match is more significant because `AGT` is more surprising under this model and therefore less likely to have occurred by chance in both sequences. Thus, the alignment incorporating this partial alignment should be given a lower cost (or equivalently a better score) than the first.

This chapter shows how to make precise the intuition from the above example. There are two distinct parts to the alignment problem — the problem of how to cost an alignment, and the problem of searching for the alignment with the lowest cost. While these are distinct problems, they are related. The search algorithm to find the optimal alignment is affected by the cost function, thus it is important to choose a cost function for which an efficient search algorithm exists. For this reason, the cost function used in this chapter is limited to an n^{th}-order Markov model with simple costs. The Markov models used in this chapter are n^{th}-order models that provide a probability distribution for the next character in a sequence dependent on the previous n characters.

Assume the sequences are non-random and emanate from mutations of a "parent" sequence that fits a Markov model. Matching high information regions accurately is more important than matching of low information regions. A new alignment method is developed for low information sequences which performs better than the standard DPA for data generated from mutations of an n^{th}-order Markov model. This new algorithm apportions weight to all regions of the sequences based on the information content of that region. It is important to remember that if a region is of 'low information', it does not mean that the region is uninteresting, but rather that the region is more probable in a statistical sense than a high information region.

Although this alignment problem is discussed with the application to DNA sequences in mind, it is not limited to them. The method discussed here is applicable to the alignment of any sequences that are noisy observations of some true non-random sequence.

It is important not to confuse the Markov models used here for the parent sequence with the profile Hidden Markov Models (HMMs) that are commonly used for modelling protein sequences [32]. The profile HMMs are often used to model a multiple alignment of protein sequences. Each position in the alignment sequence has three corresponding states, a match state, an insert state and a delete state. This type of HMM is used to model related protein sequences by having probabilities associated with each position for the likelihood of different amino-acids and the likelihood of deleting of inserting more amino-acids.

In Section 1.2.1 the basic DPA for two sequence alignment with simple costs was discussed. The new algorithm will be developed from this basic DPA. In Section 5.3.1 the model is described that produces the sequences be to aligned. The following section describes how to assign a score to a given alignment. Section 5.5 presents how to search for the optimal alignment when the sequence model is a zeroth order Markov model, and in Section 5.6 when that model is a first order Markov model.

Sequence alignment is also used to make an inference about the relatedness of sequences. One way of performing this inference is to make two different encodings of the sequences. The first encoding is for the hypothesis that the sequences are related. This first encoding uses both sequences together to produce an efficient encoding that makes use of any mutual information. The second encoding is for the hypothesis that the sequences are unrelated, also known as the 'null' model. The encoding for the null model is performed by efficiently encoding the sequences separately, then concatenating the encodings of the two sequences. When considering the two hypotheses the important factor is the length of the two encodings. Since the length of an efficient encoding of an event is equal to the negative logarithm of the probability of that event [53], the shorter the encoding, the more probable the event. Therefore, of our two hypotheses, the one with the shorter encoding is the more probable, and the better inference. Furthermore, the difference in length of the two encodings quantifies how much more probably one hypothesis is over the other. Since only the length of an encoding is important, the actual encoding need not be computed.

Alignment can be used for the joint encoding. If however we are only interested in the relatedness of the sequences the actual alignment is a nuisance parameter, so we should sum over all possible alignments [5, 6]. However, the algorithm described in this chapter does *not* sum over all alignments because it is not clear how to do this correctly for this new algorithm. Therefore, the joint encoding computed by the new algorithm will be slightly biased against inferring relatedness between two sequences.

If the sequences to be tested for relatedness come from a family of sequences that has some statistical bias this must be taken into account. If not accounted for, sequences that are unrelated (apart from coming from the same family) may actually appear to be related. This type of incorrect inference is often referred to as a false positive. This problem has long been recognized, for example Fitch [17] recommends randomising the sequences while preserving the statistical bias. Fitch discusses how a sequence can be randomly shuffled while still preserving the first-order Markov model statistics. The sequence to be tested for relatedness are first compared to obtain a score. Then to account for statistical bias in the composition of the sequences, the sequences are shuffled while maintaining the bias, then the shuffled sequences are compared. The shuffling and comparison are performed many times to obtain an average base score. If the score obtained from comparing the original sequences is significantly better than this base score, the sequences are said to be related. Wootton and Federhen [67] discuss a method to mask out the low information regions before testing. The idea is to pre-process the sequences and mask out regions that are of low information, or complexity, before they are tested for relatedness. This method is implemented in the common 'SEG' program for filtering amino-acid sequences, which may also be used as a plug-in filter for the protein database searching program 'BLAST' [8, 19]. It is incorrect to completely ignore low information regions, because such regions do contain some information, albeit less than the high information regions. The new algorithm presented in this chapter can be adapted to test the relatedness of sequences while apportioning the correct weight to each part of the sequences dependent on the information content. This is discussed further in Section 5.8.

The basic low information alignment algorithm presented in this chapter and the results in Section 5.9 have been published in the literature [44]. A variation of the algorithm that sums over parent sequences is presented in Section 5.8, and the results in Section 5.9.1 has been published [4]. Also in [4] a second algorithm for aligning low information sequences was presented. In this algorithm the sequence models are applied directly to the known sequences rather than a hypothetical parent sequence. Thus the second algorithm in [4] is for a different but related problem to the algorithm presented in this chapter.

5.2 Standard Sequence Alignment

The new alignment algorithm presented here takes into account the information content of the sequences when determining an optimal alignment. The optimal alignment that is found will in general be different to the optimal alignment if the information content of the sequences were not taken in account. An objective measure is required to compare the similarity of alignments. Such a measure could be used to compare different alignment algorithms by measuring the similarity of the optimal alignments they produce. For example, artificial sequences could be generated with a known true alignment, and this true alignment could be compared with optimal alignments from different algorithms.

The measure used in this chapter is rudimentary but does give an indication of the similarity of different alignments. Imagine the two alignments to be compared drawn on the standard DPA matrix. Figure 5.1 show two such alignments. An indication of the closeness of these alignments is given by calculating the number of matrix squares between the alignments.

The squares may be counted as halves if they lie one of the alignments. In this example the optimal alignments have an edit cost of 8, and the area between the alignments is 48 squares.

This measure is used in Section 5.9 to compare the new low information alignment algorithm to the standard DPA. This measure can also be useful in a different way. Often alignment algorithms produce many optimal alignments. The northern-most optimal alignment is the optimal alignment furthest to the top and right of the DPA matrix, and similarly the southern-most to the bottom and left. For the simple costs DPA, all optimal alignments lie between the northern-most and southern-most alignments. This new measure can be used to compare the similarity of the northern-most and southern-most alignments, and thus give an indication of the size of the 'envelope' of optimal alignments. Figure 5.1 shows the northern-most and southern-most optimal alignments for Levenshtein costs, all other optimal alignments lie between these two.

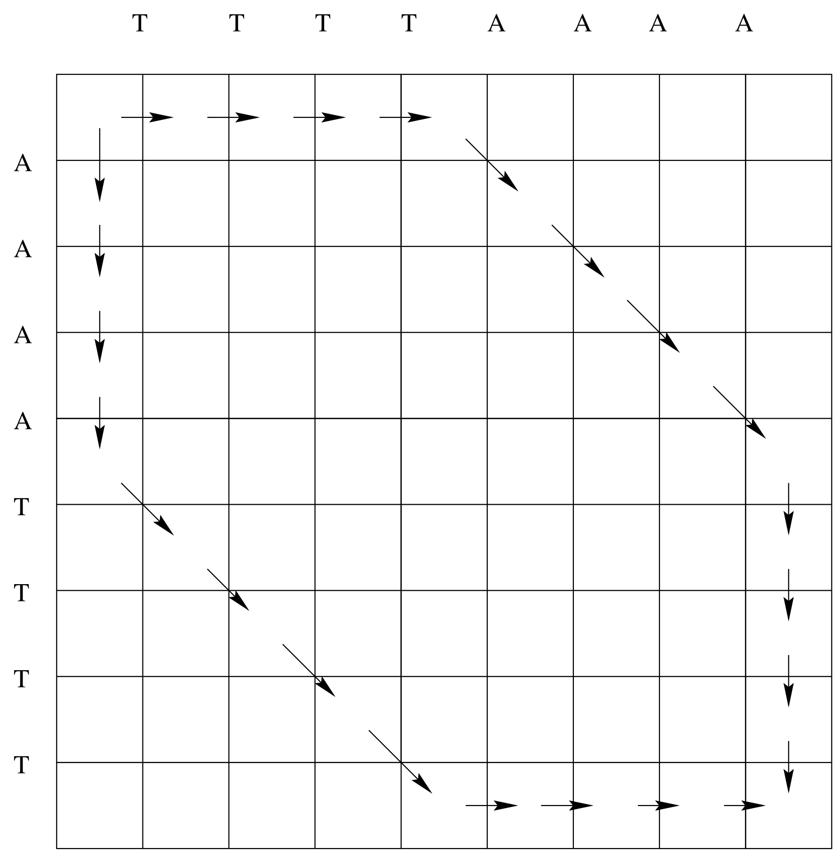

Figure 5.1: Northernmost and southernmost optimal alignments for aligning the sequences 'AAAATTTT' and 'TTTTAAAA' with Levenshtein costs.

More complex costs are sometimes used in alignments, such as linear gap costs (see Sections 1.2.2 and 2.3) where a run of n inserts/deletes have a cost of $a \times n + b$ (where a and b are constants). The algorithm presented here uses a cost function that is somewhere between simple costs and linear gap costs. Although it should be straightforward to extend the algorithm to use linear gap costs this has not been done as the focus of the algorithm in this chapter is on low information sequences.

5.3 Costing an Alignment

A family of sequences, F, is *non-random* (or *compressible* or *of low information content*) if there is some model M with a compression algorithm $m()$ such that, on average, $|m(S)| < |S|$ for sequences drawn from F. For $m()$ to be a sensible compression algorithm, there must exist a decompression algorithm, $m^{-1}()$, such that $m^{-1}(m(S)) = S$. It is desirable to use $m()$ in costing the alignments because it gives a precise measure of the information content of a sequence. However, $m()$ deals with a single sequence, not an alignment of two sequences, and thus a model of alignment that incorporates $m()$ is required.

For the new alignment model, imagine that there is one true, unknown, sequence, R, and two noisy observations of it, $S1$ and $S2$. Sequence R may be an ancestral sequence of $S1$ and $S2$. The sequence R is assumed to be compressible by $m()$ on average. Therefore, the model M can be used to give a prior probability for any given R. It is not required that the ancestral sequence R be a "real" sequence, it may be a hypothetical sequence that we wish to infer from the sequence $S1$ and $S2$. The problem first considered is inferring R given $S1$ and $S2$. The next problem considered is computing the relatedness of $S1$ and $S2$ under such a model.

This model is considered to be a good fit to the DNA sequence assembly problem [54, 55, 31], where there is some "true" (non-random [35, 45, 1, 48]) DNA sequence that is sequenced in fragments using techniques that introduce noise. These noisy fragments are then used to infer the true DNA sequence. The position of the fragments are unknown and must be inferred. This inference is done by finding the overlap between fragments, however low complexity regions can introduce problems. For example, if the original DNA sequence has a number of subsequences that are very similar, it can be difficult to correctly align the fragments from these regions. Using an alignment algorithm that correctly apportions less importance to these low complexity regions would help with this problem. The new alignment algorithm as presented in this chapter does not handle overlaps between sequences, but it would be straightforward to extend the algorithm for this.

5.3.1 Specifying the Model for Data Generation

We need to define how the sequences R, $S1$ and $S2$ come about, and therefore their mutual alignment. Assume we have a sequence, R, generated by a known model M. It is necessary to carefully state the method by which the noisy observations, $S1$ and $S2$, are mutated from R. Before defining the mutation model precisely, there are some useful properties the mutation model should have. The mutation model may be considered as a machine like the mutation machine shown in Figure 3.4. The essence of the mutation model is how the mutation instructions are produced. The form of the mutation model will affect the final algorithm for inferring R from $S1$ and $S2$. It is important that this algorithm be efficient. To this end it is useful to assume our mutation model processes the sequence R one character at a time. Each character of R may be copied directly to the output sequence, copied incorrectly, or deleted entirely. Between each character of R a number of characters may be inserted into the output sequence. With the goal of an efficient final algorithm, it is useful to use a geometric distribution on the length of such insertions. In standard alignment algorithms, the geometric distribution is implicitly used for gap lengths when using simple costs or linear gap costs.

With these desirable properties of the mutation model in mind, the exact method for generating sequence $S1$ and $S2$ from R is defined. To generate $S1$ (and likewise $S2$) from R, do the following:

1. For each character in R do the following :
    (a) Insert n characters, where n is sampled from the geometric distribution $Pr(n) = (P_{insert})^n(1 - P_{insert})$.
    (The mean of this distribution is $P_{insert}/(1 - P_{insert})$.)
    (b) Copy the character of R correctly (match), copy it incorrectly (mismatch), or do not copy it (delete), with the respective probabilities P_{match}, $P_{mismatch}$, P_{delete}. (Note: $P_{match} + P_{mismatch} + P_{delete} = 1$.)
2. At the end of R insert n characters, where n is sampled from the geometric distribution $Pr(n) = (P_{insert})^n(1 - P_{insert})$.

If the sequence, say $S1$, generated from R is desired to be of similar length to R on average, that is, $|S1| \approx |R|$, then $P_{delete} = P_{insert}/(1 - P_{insert})$. This will, on average, cause approximately same number of characters to be deleted from R as are inserted.

This method of generating the sequence $S1$ (and $S2$) from the sequence R can be represented as a finite state machine. Such a FSM is shown in Figure 5.2, which begins in the insert state and ends in the copy state. Each transition out of the copy state corresponds to one character of the R sequence, thus there $|R|$ transitions out of the copy state, and $|R|+1$ possible places for insertions.

It is interesting to compare this FSM with the three-state FSM for linear gap costs shown in Figure 3.5. If the three-state FSM were reduced to two states by removing the `Delete` state and the transition probabilities were set appropriately, then both machines would produce every sequence of instructions with the same probability. For the purposes of this comparison, the probabilities for the FSM in Figure 3.5 shall be referred to with a subscript one, for example, $P_1(Ins|I)$. The probabilities for the FSM in Figure 5.2 will referred to with a subscript two, for example, $P_2(ins)$. If the FSM transition probabilities are set as shown below then both machines will give the same probability to every sequence of mutation instructions. Note that in the FSM in Figure 5.2 there is a transition for both a copy and a change, while in the FSM in Figure 3.5 the `Match` transition is used to represent *both* a copy and a change.

$$
\begin{aligned}
P_1(Ins|M) &= P_2(ins) \\
P_1(Ins|I) &= P_2(ins) \\
P_1(Match|M) &= [P_2(copy) + P_2(change)] \times [1 - P_2(ins)] \\
P_1(Del|M) &= P_2(del) \times (1 - P_2(ins)) \\
P_1(Match|I) &= [P_2(copy) + P_2(change)] \times [1 - P_2(ins)] \\
P_1(Del|I) &= P_2(del) \times [1 - P_2(ins)]
\end{aligned}
$$

The character chosen when a mismatch is performed, and the characters chosen to insert are selected from a uniform distribution over A,T,G and C. This was done for simplicity, although it is easily modified so that the characters chosen depend on the character that is being mismatched from (for example, an ‘A’ may be more likely to mismatch to a ‘T’ than a ‘G’ or ‘C’).

By outlining this generating model explicitly, it is evident that there is a real distinction between an insertion and a deletion, thus, they must be handled differently. In this model insertions and deletions occur between the parent sequence R and either $S1$ or $S2$. Compare this with standard alignment algorithms where insertions and deletions are between the se-

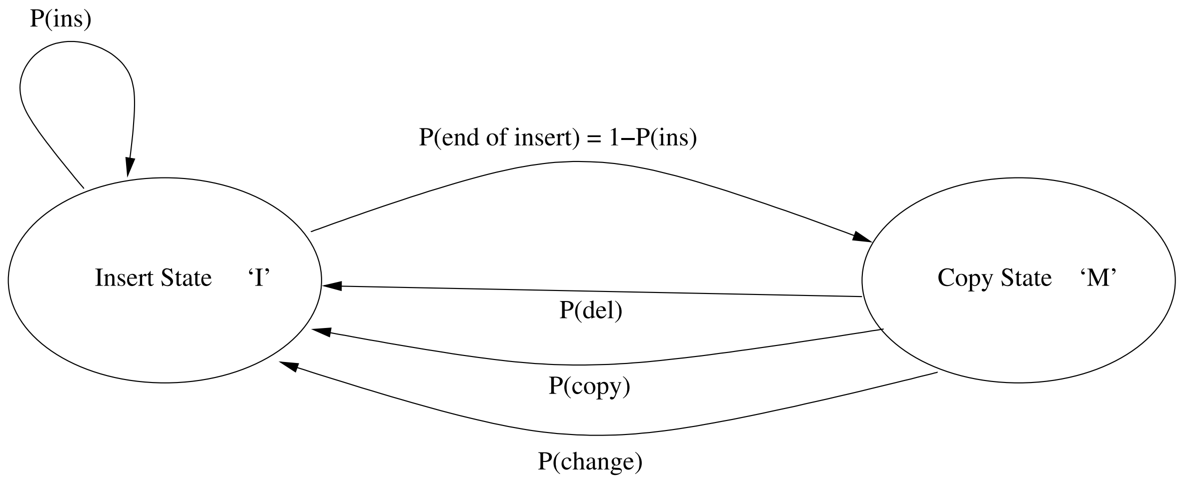

Figure 5.2: The finite state machine model for generating a sequence $S1$, as a 'noisy' copy of a parent sequence R.

quences being aligned. Since the sequences being aligned are considered 'equal', insertions and deletions are treated equally in standard alignment algorithms.

The above description of the mutation model is related to simple costs alignment, because mutations are treated as point mutations. The mutation model could be modified to be similar to linear-gap costs by having a probability to start an insertion, then a lower probability to continue the insertion. This would only be a simple modification to the above mutation model. Deletions may be handled in a similar manner by adding an extra state and having a probability to start deleting, and a different probability to continue deleting characters.

5.3.2 Model M of R

It is important that the sequence model is applicable to all possible sequences. To simplify matters, the sequence model considered first shall be a 0th order Markov model (MM), and later it will extended to a 1st order MM.

A Markov model is used to define the probability of any character in a sequence, dependent only on the previous n characters, where n is the order of the Markov model. So, for a zeroth order Markov model of a DNA sequence, the compression model, M, has a fixed probability for each character A,T,G and C of occurring irrespective of context. Some DNA sequences have a very biased nucleotide distribution, so a zeroth order MM would be a better for those sequences one that assumes all nucleotides are equally likely. An example of this would be sequences from the genome of the malaria causing parasite *Plasmodium falciparum*. An efficient encoding of a DNA sequence which is uniformly random would achieve 2 bits per nucleotide base. In comparison, a 0th order Markov model achieves about 1.7 bits/base for chromosome 3 of this organism (see Table 5.2).

R is generated by using M in the following manner:

1. Decide whether to append another character to R with probability P_y.
2. Choose the character to add from the 0th order MM.

The expected length of R is thus $P_y/(1 - P_y)$, so for R to be of any substantial expected length, P_y needs to be very close to 1.

5.4 Encoding R, $S1$, $S2$ and their Alignment

At this point, the generation model for the sequences R, $S1$,and $S2$ has been defined. It is now outlined how to assign a probability to how likely a particular instance of R, $S1$, $S2$ and their alignment is of being generated. This is obviously very closely related to the generating model of Section 5.3.1.

Since the probabilities become very small, it is convenient use the negative logarithm of probabilities. The negative logarithm of the probability of an event can be considered as the length of an efficient encoding of that event [53]. Indeed, the following discussion will focus on constructing a "message" which is an efficient encoding of R, $S1$ and $S2$. From the correspondence between message lengths and probabilities, the construction of a message is the same as determining the joint probability of the encoded events.

Given R, $S1$, $S2$ and their alignment, we wish to encode these efficiently. We will construct a message containing R encoded using $m()$, and $S1$ encoded as differences from R and $S2$ similarly. The shorter this message, the more likely this alignment is. The encoding is as follows:

1. Encode R using m().

2. $S1$ is encoded as a series of operations as follows:

   (a) For each character of R do the following:

      i. Encode run of n inserted characters in $S1$, by encoding n with $prob(n) = (P_{insert})^n(1 - P_{insert})$ and the characters each with uniform probability=$1/4$.

      ii. State whether this character of R has been matched, mismatched or deleted.

      iii. If mismatched, state mismatched character [1]with probability=1/4.

   (b) Encode a final run of inserts for $S1$.

3. $S2$ is encoded in a similar manner.

Note that this process is the same irrespective of the model M. The choice of M only affects the search. In this process characters that are inferred to be inserted are encoded from a uniform prior, as was discussed for the model of mutation in Section 5.3.1. It would be incorrect to encode the inserted characters using $m()$ since $m()$ only applies to the sequence R, and inserted characters do not come from R.

For most applications, sequences $S1$ and $S2$ will be available, and from these R and the alignment must be inferred. By using the encoding outlined above, it is possible to compare two (or more) inferred alignments. The next step is to search for the optimal alignment, which will be discussed in the next section.

As an example of this encoding, consider the two sequences `ATGAACTG` and `ACGTA`. Assume that under a certain model, M, the best inference for the parent sequence is `ACGCTG` and the best inference for the alignment between the two sequences and the parent is shown below.

R:	A	C	G	–	–	C	T	G
$S1$:	A	T	G	A	A	C	T	G
$S2$:	A	C	G	–	–	–	T	A

Using the above process for encoding, the encoding of these sequences and their alignment is now given. The encoding events shall be represented as follows: `E1` and `E2` represent the

[1]Need to consider if mismatches are allowed to the same character or not. These can be considered to be equivalent. Given an alignment which is generated with mismatches allowed to the same character, it can be interpreted as being generated with mismatches having to be to a different character by modifying P_{match} and $P_{mismatch}$ (and vice versa).

end of insertions in $S1$ and $S2$ respectively, E1 $=$ E2 $= 1 - P(ins)$; C1 and C2 represent a copy into $S1$ and $S2$ respectively, C1 $=$ C2 $= P(copy)$; c1 and c2 represent a change for sequence $S1$ and $S2$ respectively, c1 $=$ c2 $= P(change)$; D1 and D2 represent and deletion from $S1$ and $S2$ respectively, D1 $=$ D2 $= P(del)$; m(X) represents the encoding of the character X from the parent sequence using $m()$; u(X) represents encoding the character X using a uniform distribution. Note that an event I or c must be followed by an encoding using u() of the character that is inserted of changed to respectively. The encoding is shown below with the encoding on the lines starting with 'encoding:'. Each alignment character is shown above the encoding event that encodes that alignment character.

```
                  <A,A,A>                 <C,T,C>
encoding: E1 E2   m(A) C1 C2    E1 E2     m(C) c1 u(T) C2


                  <G,G,G>       <-,A,->   <-,A,->
encoding: E1 E2   m(G) C1 C2    I1 u(A)   I1 u(A)


                  <C,C,->                 <T,T,T>
encoding: E1 E2   m(C) C1 D2    E1 E2     m(T) C1 C2


                  <G,G,A>
encoding: E1 E2   m(G) C1 c2 u(A)    E1 E2
```

5.5 Search for Optimal R and Alignment (0th Order MM)

Given $S1$ and $S2$, it is necessary to search for the best R and alignment, where 'best' means the most likely (having the shortest encoding). In this chapter, the search is done by modifying the generic dynamic programming algorithm (DPA) commonly used for sequence alignment; the DPA was discussed in Section 1.2.1. To keep the explanation simple we will first discuss the search disregarding the possibility of inserts (i.e. for every character in $S1$ and $S2$ there is a corresponding character in R). Each cell, $D[i, j]$, in the D matrix, will contain the best cost of aligning (and inferring R) for the sequences $S1$[1..i] and $S2$[1..j]. To allow for this incremental approach to the costing of the alignment, the encoding scheme described in the previous section is slightly modified in the following manner: one character of R is encoded, followed by the corresponding operation for $S1$ (a match, mismatch or delete) then the operation for $S2$. Since we are now considering $S1$ and $S2$ at the same time, it is convenient to use a finite state machine that is the result of combining the FSM that produced $S1$ and the FSM that produced $S2$. Since we are ignoring insertions at this point the resulting FSM has two states as shown in Figure 5.3. For each character of the sequence R there is a transition out of the state 'Copy $S1$' followed by a transition out of state 'Copy $S2$'. Note that although these states are called 'Copy' the character may be copied correctly, copied incorrectly or deleted altogether. The state is called a 'Copy' state because the next operation depends on the character of the R sequence.

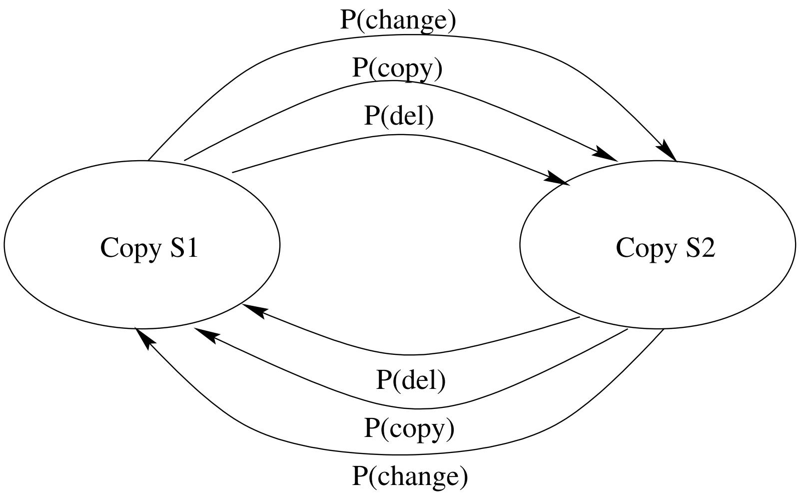

Figure 5.3: The combined finite state machine model for sequences $S1$ and $S2$ as independent noisy observations of a parent sequence R disallowing insertions.

Using this combined FSM, the calculation for D matrix cell $D[i, j]$ is as shown in Listing 5.1.

Finally, the cell $D[|S1|, |S2|]$ will contain the cost for the inferred R and for aligning that R with $S1$ and $S2$. R and the alignment can be found by back tracking through the D matrix. The algorithm, like the standard DPA, has a time and space complexity of $O(n^2)$ (where n is the length of the sequences) but with the multiplicative constant increased by the alphabet size. It is possible to apply the check-point method of Chapter 2 to this algorithm to reduce the space complexity to $O(n)$.

5.5.1 Handling Inserts

Performing an optimal search while allowing for insertions is more complicated. The new algorithm presented in this section shall be referred to as the 'lowinfo' algorithm. Coincidentally, this algorithm resembles that of Gotoh [20] used for linear gap cost alignment.

```
score = infinity;
for each Rchar in (A,T,G,C)
  { First the diagonal move, i.e. a match or mismatch }
  score = min(score,
       D[i-1,j-1] +            { Cost so far                             }
       mInc(Rchar) +           { Incremental cost of add another char to R }
       mis(Rchar, S1[i]) +     { Cost to encode mis/match for S1         }
       mis(Rchar, S2[i]))      { Cost to encode mis/match for S2         }

  { Now the vertical move, i.e. a character from S1 but not from S2 }
  score = min(score,
       D[i-1,j] +              { Cost so far                             }
       mInc(Rchar) +           { Incremental cost of add another char to R }
       mis(Rchar, S1[i]) +     { Cost to encode mis/match for S1         }
       del)                    { Cost to encode a delete for S2          }

  { And the horizontal move, i.e. a character from S2 but not from S1 }
  score = min(score,
       D[i,j-1] +              { Cost so far                              }
       mInc(Rchar) +           { Incremental cost of add another char to R  }
       del +                   { Cost to encode a delete for S1           }
       mis(Rchar, S2[j]) +     { Cost to encode mis/match for S2          }

D[i,j] = score
```

Listing 5.1: The calculation to determine the matrix cell at (i,j).

The encoding from the previous section must be modified to allow for inserts. This is done in the same manner as in Section 5.4, but simply rearranged. First, the run of inserts for $S1$ is encoded followed by a run of inserts for $S2$. Then a character from R is encoded, and the corresponding operation for $S1$ — copy, change and the character to change to, or delete — then the operation for $S2$. This imposes an order on the encoding of the sequences. Zero or more insertions in $S1$ are followed by zero or more insertions in $S2$ which is followed by a character of R and an edit operation for $S1$ and $S2$.

Combining the FSMs for generating $S1$ and $S2$ while allowing for insertions results in the FSM shown in Figure 5.4. As each character of sequence R is processed, there is a transition out of the state 'Copy $S1$' and a transition out of the 'Copy $S2$' state. This makes it possible only to remain in the 'Insert $S1$' or 'Insert $S2$' states, and this does not use a character of the R sequence. Note that in this FSM the transitions for inserting into $S1$ and $S2$ are completely separate, so it is possible to have different probabilities for these two events. However, since $S1$ and $S2$ are likely to be derived from the same mutation machine, the same probability, $P(ins)$, will be used for insertions in either $S1$ or $S2$. Likewise for the $P(copy)$, $P(change)$ and $P(del)$ transitions, the same probabilities will be used for both sequences.

The algorithm to find the best alignment of $S1$ and $S2$ under this model is, as expected, similar to that of the previous section. When a character of the R sequence is encoded, the FSM moves from the 'Copy $S1$' state to the 'Copy $S2$' state while encoding one of the

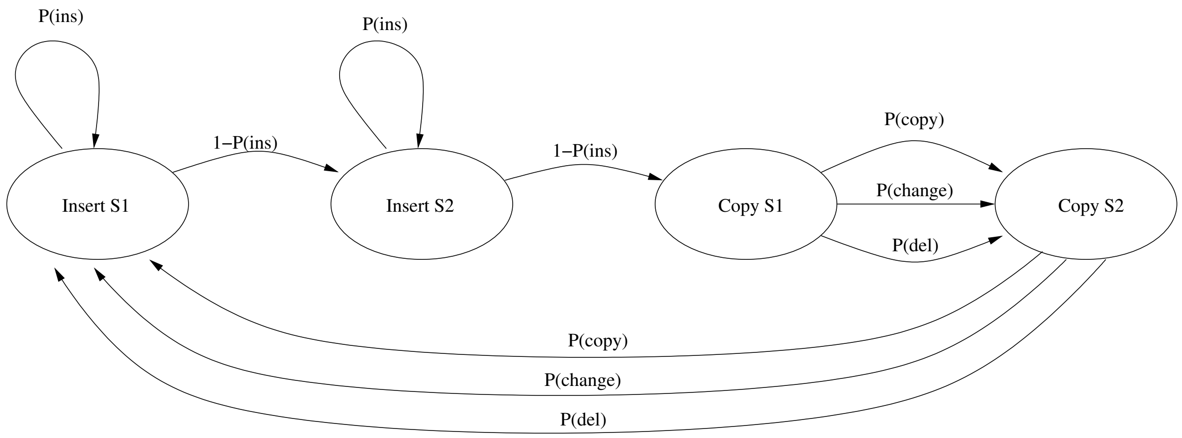

Figure 5.4: The combined finite state machine model for sequences $S1$ and $S2$ as independent noisy observations of a parent sequence R with insertions.

following operations: this character is copied to $S1$; deleted from $S1$; copied incorrectly to $S1$ and the actual $S1$ character. The FSM then moves from the 'Copy $S2$' state to the 'Insert $S1$' state while similarly encoding for $S2$.

In this DPA style algorithm each cell of the D matrix has two possible states corresponding to whether the underlying FSM is in the 'Insert $S1$' state or the 'Insert $S2$' state. The first shall be called 'State 1' and the second 'State 2'. The DPA matrix shall be considered with sequence $S1$ "vertically on the left" of the matrix accessed by the variable i, and sequence $S2$ "horizontally on the top" of the matrix accessed by the variable j, see Figure 5.5. So a vertical move in the DPA matrix is either an insertion in $S1$ or a deletion from $S2$. Likewise a horizontal move is either a deletion from $S1$ or an insertion in $S2$.

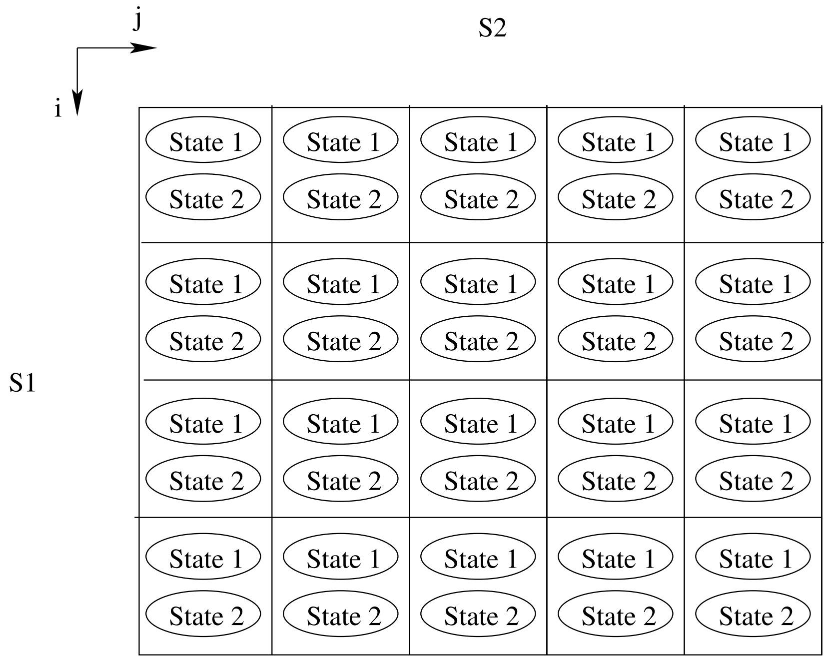

Figure 5.5: The DPA matrix for the lowinfo algorithm labelled with sequences $S1$ and $S2$.

In understanding the alignment algorithm, let us consider the various events we may wish to encode, and how these correspond to transitions through the FSM in Figure 5.4. To begin, let us consider the FSM is in the 'Insert $S1$' state and that we wish to encode an inserted character in $S1$. This is a simple matter of taking the transition to remain in the 'Insert $S1$' state while encoding an insertion event with probability $P(ins)$ and encoding the character using a uniform distribution over the possible characters. In terms of the DPA matrix this corresponds to a vertical move from 'State 1' in $D[i, j]$ to 'State 1' in $D[i + 1, j]$.

Next, consider the FSM is in the 'Insert $S1$' state but this time we wish to encode a character of sequence R and how this character was used in sequences $S1$ and $S2$. The FSM moves via the 'Insert $S2$' state to the 'Copy $S1$' state and encodes the character of R using $m()$. The appropriate transition for the desired edit operation for $S1$ is then taken to the 'Copy $S2$' state. Followed by a transition for the edit operation in $S2$ which puts the FSM back in the 'Insert $S1$' state. If the two operations were both 'copy' operations, then the probability associated with this series of transitions is $[1 - P(ins)] \times [1 - P(ins)] \times P(RChar) \times P(copy) \times P(copy)$ where $P(RChar)$ is the probability used to encode the character of R using $m()$. This corresponds to encoding an end of insertions for $S1$, an end of insertions for $S2$, the character of R, a copy into $S1$ event and a copy into $S2$ event. In terms of the DPA matrix, this example is a diagonal move from 'State 1' in $D[i, j]$ to 'State 1' in $D[i+1, j+1]$. However, if the mutation event for $S1$ were a deletion then this would be a horizontal move from 'State 1' in $D[i, j]$ to 'State 1' in $D[i, j + 1]$. Likewise if the mutation event for $S1$ were a copy (or a change), and the mutation for $S2$ was deletion, this would be a vertical move in the DPA matrix from 'State 1' in $D[i, j]$ to 'State 2' in $D[i + 1, j]$. It is important to note that this move direction in the DPA matrix is the same as in the previous paragraph, with the difference being that it encodes a different event; this one encodes a character of R and a deletion in $S1$, while the former is for an insertion in $S1$. Which of these two is more probable depends on the mutation probabilities, and more importantly on the cost of encoding the character in R.

We shall now examine what happens when the FSM is in the 'Insert $S1$' state and we wish to encode an inserted character in sequence $S2$. The FSM moves to the 'Insert $S2$' state and we encode the move with probability $1 - P(ins)$. The FSM then takes the transition to remain in the 'Insert $S2$' state, which we encode with probability $P(ins)$. We also encode the character to be inserted with probability from a uniform prior over that alphabet of characters.

We now consider the FSM to be in the 'Insert $S2$' state, and we wish to encode an inserted character in $S2$. The FSM will remain in the 'Insert $S2$' state encoding the insertion event with probability $P(ins)$ and the character to be inserted with probability taken from a uniform distribution. This is a horizontal move in the DPA matrix from 'State 2' in $D[i,j]$ to 'State 2' in $D[i,j+1]$.

Finally, consider the FSM to be in the 'Insert $S2$' state and we are to encode a character of R and the appropriate mutation events for $S1$ and $S2$. The FSM must move from the 'Insert $S2$' state through the 'Copy $S1$' and 'Copy $S2$' states to finish in the 'Insert $S1$' state. The events to be encoded are: the end of insertion for $S2$, the character of R, the mutation event for $S1$, and the mutation event for $S2$. Depending on the mutation event for $S1$ and $S2$ this can correspond to a horizontal, vertical or diagonal move in the DPA matrix.

Recall that 'State 1' is used to denote that the FSM is considered to be in the 'Insert $S1$' state and 'State 2' to denote the 'Insert $S2$' state. The transitions detailed in the preceding paragraphs are illustrated in Figure 5.6 with each transition labelled with the events that must be encoded for that transition. For each of these event labels, Table 5.1 lists the label, the event it represents, and the probability that event. It may be useful to refer to the FSM in Figure 5.4 when considering these transitions.

Table 5.1: Summary of the transition labels, the event represented, and the transition probability used in Figure 5.6.

Symbol	Event	Probability
I1	Insertion in $S1$	$P(ins)$ and the character
I2	Insertion in $S2$	$P(ins)$ and the character
E1	End of insertions in $S1$	$1 - P(ins)$
E2	End of insertions in $S2$	$1 - P(ins)$
R	A character of R is to be encoded	As given by $m()$
D1	Deletion from $S1$	$P(del)$
D2	Deletion from $S2$	$P(del)$
C1	Copy or change for $S1$	$P(copy)$ or $P(change)$ and the character
C2	Copy or change for $S2$	$P(copy)$ or $P(change)$ and the character

To reiterate, in terms of the D matrix of the DPA a vertical move corresponds to either an insertion in $S1$ or a deletion in $S2$. Both options are possible and the best option is the one that results in the shortest final encoding of the alignment. To handle an insertion in $S1$ the FSM must be in the 'Insert $S1$' state and remains in that state while the inserted character is encoded. When handling a deletion from $S2$, the FSM machine may currently be in the 'Insert $S1$' state ('State 1' of the DPA) or in the 'Insert $S2$' state ('State 2' of the DPA). In either case, the FSM moves through the 'Copy $S1$' state and the 'Copy $S2$' state then finishes in the 'Insert $S1$' state.

A horizontal move is similar in that it may be an insertion in $S2$ or a deletion from $S1$. The case of the deletion from $S1$ is much the same as above only with $S1$ and $S2$ swapped. The

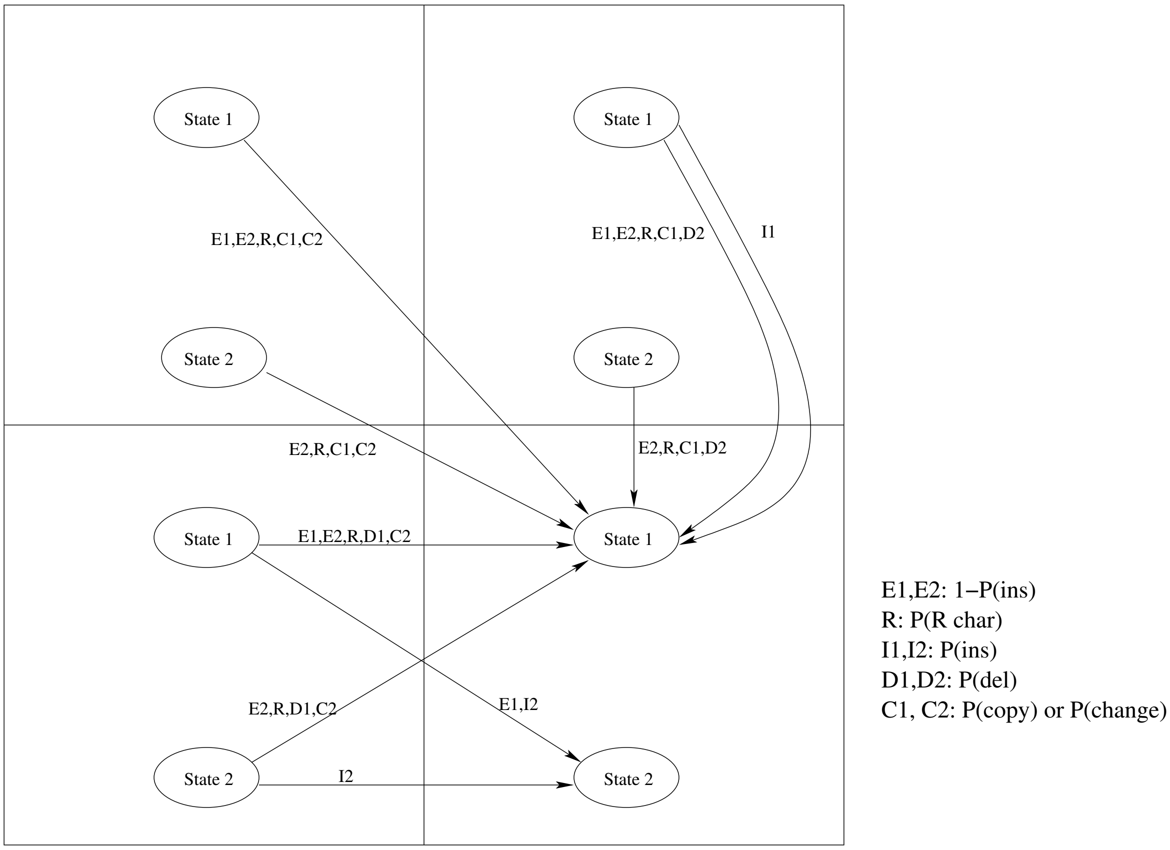

Figure 5.6: All possible transitions in the DPA matrix.

case of an insertion in $S2$ is similar in that if the FSM is in the 'Insert $S2$' state it simply remains in that state. However if the FSM is in the 'Insert $S1$' state then it switches to the 'Insert $S2$' state, encoding the end of insertions in $S1$, and encodes the inserted character of $S2$.

It is worth noting that under this model it is possible to follow a run of insertions in $S1$ with a run of insertions in $S2$. While to have a run of insertions in $S1$ after insertion in $S2$, a character of the R sequence must be encoded in between. This is due to the order of states of the FSM in Figure 5.4. Limiting the order that insertions must be encoded puts a canonical form on the encoding of an alignment. Thus the encoding of any given alignment is unique.

A diagonal move in the D matrix corresponds to a cycle through the FSM starting from either the 'Insert $S1$' state or the 'Insert $S2$' state ('State 1' and 'State 2' respectively in the algorithm) through the 'Copy $S1$' state and 'Copy $S2$' state to the 'Insert $S1$' state.

As an example, consider the following:
$S1$ contains a character 'A', and $S2$ has no corresponding character. This can be explained either as (1) an insert in $S1$ (R had no corresponding character), or (2) as a deletion in $S2$ (R had a character 'A').

Case 1 Probability of saying it is an Insert $\times$ probability stating inserted character.

$P_1 = P_{insert} \times \frac{1}{4}$

Case 2 Probability to end inserts for $S1$ and $S2$. Then character for R, a match and a delete.

$P_2 = (1 - P_{insert}) \times (1 - P_{insert}) \times P_y \times m(A) \times P_{match} \times P_{delete}$

Set P_1 equal to P_2 and solve for $m(A)$:

$m(A) = \frac{1}{4} \times P_{insert}/((1 - P_{insert})^2 \times P_y \times P_{match} \times P_{delete})$

Typical values:

- $P_{insert} = 0.05$
- $P_{match} = 0.9$
- $P_{delete} = 0.05$
- P_y is typically very close to 1.

For these typical values, $m(A) \approx 0.311$. Thus for the above typical values, if $m(A) > 0.31$, R will be inferred to have the character 'A' (case 2), otherwise it will be cheaper to state that R had no such character, so 'A' was inserted in the noisy observation (case 1).

5.6 First Order Markov Model

A 0th order Markov model is a special case of and often an inferior modeling tool to a 1st order Markov model. Indeed, some biological sequences are better modelled by higher order Markov models. Table 5.2 shows the average bits per base-pair for chromosome 3 of the malaria causing parasite *Plasmodium falciparum* using different order Markov models. For this sequence the best model is around a 5th or 6th order Markov model. The table shows that the encoding becomes less efficient for Markov models above 6th order. This is can explained by the fact the higher order Markov models have more parameters, and in this example there is not enough data to usefully estimate these parameters. In this Section, the focus shall be on a first order MM which is, in general, an improvement over the zeroth order MM of the previous section.

As previously noted, if M is changed, the only alteration needed to the encoding is in the encoding of the characters of R. However, the search is dependent on M, so it must be modified. Since a 1st order MM needs only the previous character for context information, $m()$ may still be used incrementally in the encoding of R.

Table 5.2: Average bits per base-pair for chromosome 3 of *Plasmodium falciparum* using different order Markov models.

Model	Average bits/char
Null model	2
0th order MM	1.72
1st order MM	1.71
2nd order MM	1.68
3rd order MM	1.67
4th order MM	1.65
5th order MM	1.64
6th order MM	1.64
7th order MM	1.65
8th order MM	1.67
9th order MM	1.69
10th order MM	1.73

The search for the optimal R and alignment when M is a 1st order MM is very similar to that in the previous section. If insertions are not allowed, it is simply a matter of adding a state for each of A,T,G or C to each cell of the D matrix. For each state it is necessary to calculate the encoding conditional on whether the previous character of R was an A, T, G or C, thus four times as much work is done than when M is a 0th order MM. The search is still $O(n^2)$, with the multiplicative constant increased by the size of the alphabet squared when compared to the standard DPA.

If insertions are allowed, we need two super-states, each containing states for A, T, G and C. These two super-states are analogous to the two states in the 0th order MM search. The calculation of encoding lengths is done by a combination of 1st order MM with no inserts and the 0th order MMs that allows inserts. To simplify the back-tracking step (to determine alignment), extra information is carried in each state about where it came from. Four of the cells of the D matrix are shown in Figure 5.7. Again this example is for the alphabet of DNA sequences. Each cell contains two states labelled 'State 1' and 'State 2', which are analogous to the states 'State 1' and 'State 2' in Figure 5.5 and in Figure 5.6 for the 0th order MM algorithm. Both of these states contain four encoding lengths for each possible alphabet character.

This algorithm was implemented and results of test runs are given in Sections 5.9. This algorithm will hereafter be referred to as the 'lowinfo alignment' algorithm, and the program implementing it as the 'lowinfo alignment' program.

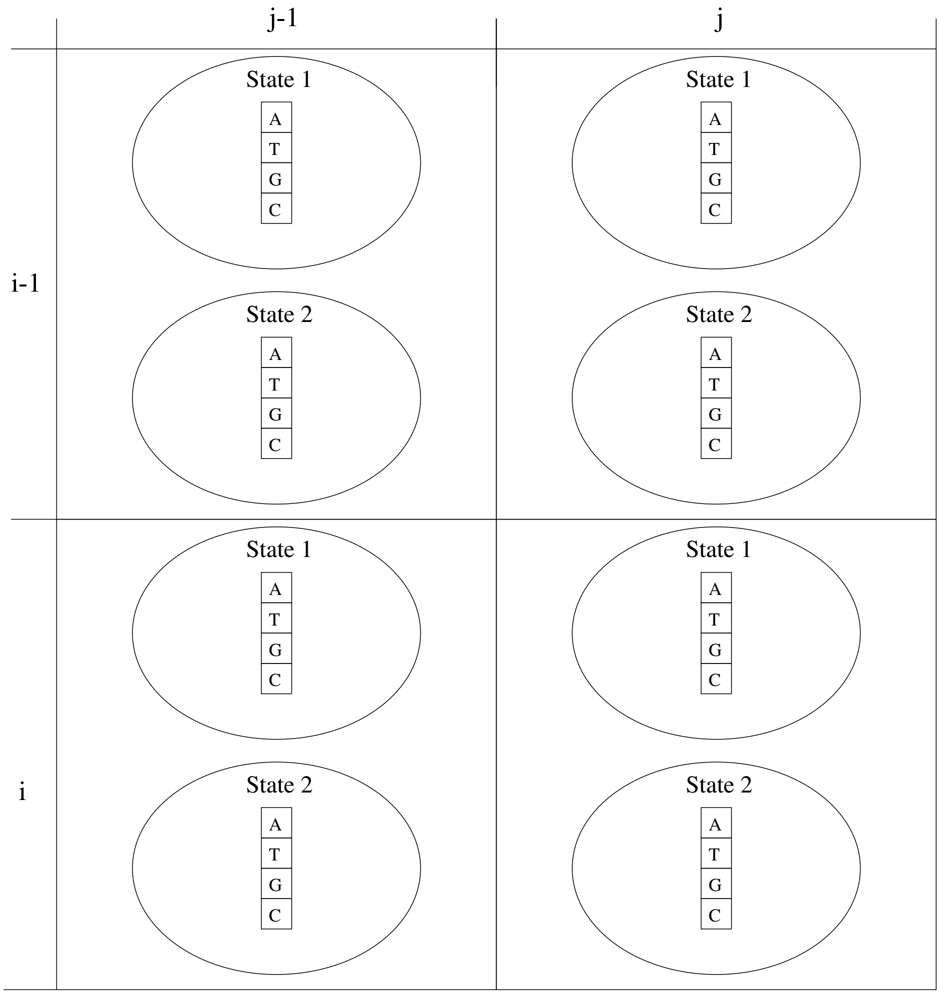

Figure 5.7: Contents of the DPA matrix cell for the lowinfo algorithm for a first order Markov model.

5.7 Null Encoding

To test the significance of an alignment from the lowinfo algorithm, a null encoding of the sequences is required. In this case, the null model is that $S1$ and $S2$ are noisy observations of two *different* true sequences. The length of the null encoding is the sum of the length for encoding $S1$ and $S2$ separately. The significance of an alignment can be judged by comparing its encoded length with that of the null encoding. If the alignment is a better explanation of the data its encoding will be shorter than the null encoding.

The null encoding is also useful in the case where *part* of $S1$ and $S2$ correspond to the same true sequence, such as in determining sequence overlap. The overlap region is encoded using the alignment encoding defined earlier, and the two regions of $S1$ and $S2$ that don't overlap are encoded using the null model. Thus it can be judged whether an overlap is significant by comparing it with the null model. This is useful for the sequence assembly problem.

The null encoding is calculated in a similar manner to the encoding of an alignment. $S1$ is encoded by first encoding R, then encoding differences from it. When searching for the optimal encoding, it is reasonable to ignore the possibility for a delete (a character not in $S1$ that was in R). Thus, the null encoding can be calculated in $O(|S1|)$ time.

The null encoding can be considered to be making an inference of the R sequence given just one observation, $S1$, and a sequence model for R, M. The algorithm to calculate the null encoding considers all possible sequences for R, up to the length of $S1$. That is, the algorithm does not consider any characters for R that do not appear in $S1$. While this is not ideal, it is considered reasonable for all but the most contrived models for R.

The algorithm for the null encoding of $S1$ processes one character of $S1$ at a time. For each position, each possible alphabet character is considered for the R sequence, and the possible encoding length is computed. The encoding length is also computed for the situation where this character was an insertion. When the model being used for the sequence R is a zeroth order MM, there is a single best cost (short encoding) kept by the algorithm for each character of the $S1$ sequence.

In the more interesting case where the model M is a first order Markov model, each position needs to keep $|\Sigma|$ encoding lengths, where $|\Sigma|$ is the size of the alphabet. These encoding lengths are computed from the preceding length by considering each in turn and choosing the shortest encoding which is the same as choosing the most probable. An insertion is encoded by keeping the same previous character for the R sequence. Figure 5.8 shows an example for a four character alphabet DNA (A,T,G and C) of how the encoding lengths for the 'A' character at position i depend on the lengths at $i-1$. The same is repeated for the other characters in the alphabet. Compare this diagram to the one in Figure 5.7; this diagram is essentially a simplified version of the earlier one.

5.8 Measuring the Relatedness of Two Sequences

Sequence alignment is also used as a measure of the relatedness of two sequences. Many sequences have a low information content. If this is not taken into account, the sequences may appear to be much more closely related than they actually are. As an example, consider two sequences selected at random from *Plasmodium falciparum* (discussed at the start of Section 5.6). Two such sequences are not directly related apart from the fact that they come from the same genome. If they were to be compared without considering that they come from a biased family, it is likely that we would infer the sequences were related since they both have such a biased base distribution.

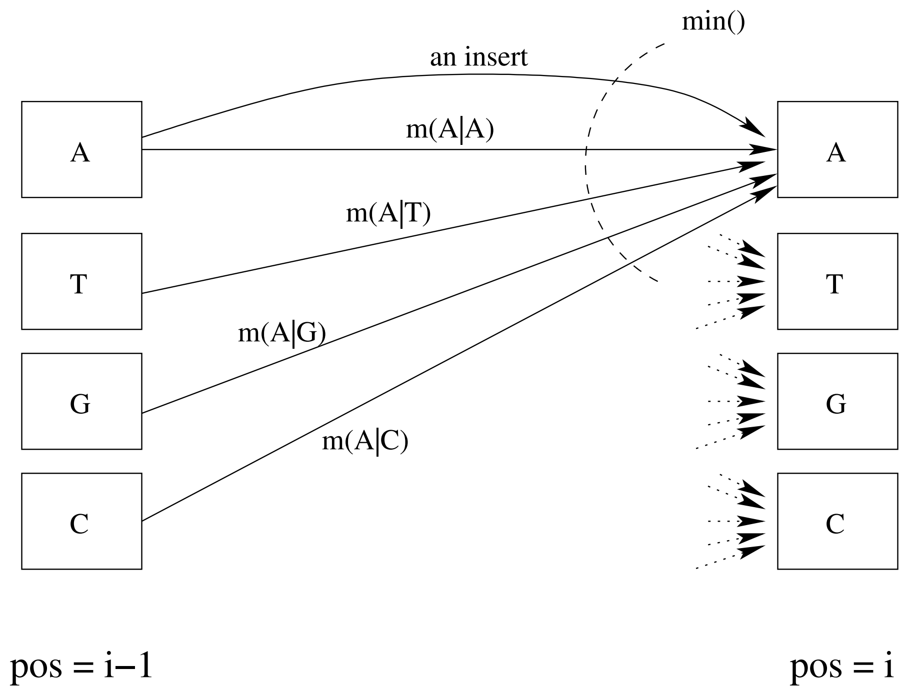

Figure 5.8: Computing the null encoding for a 1st order Markov model.

The new method described in this chapter for aligning sequences of low information content can also be used to measure the relatedness of the sequences. It is possible to use the algorithm as already described and compare this to the null encoding as a measure for relatedness. However this is not optimal. If all that we are interested in is whether the sequences are related or not, then the *parent sequence*, R, is a nuisance parameter, and we should sum over all possible parent sequences. Indeed, in this situation we do not even care about the *actual alignment*, only whether the sequences are related or not, so the actual alignment is also a nuisance parameter and we should sum over all possible alignments [5, 6]. If we choose just the optimal alignment, then the encoding length will be a little longer than if we summed over all possible alignments. One example of when this will make a difference is for sequences that have many "good" alignments, but no single "great" alignment. In this situation the sequences are likely to be related, but choosing just one of the alignments would infer that the sequences were unrelated. However, if we summed over all the alignments, all the "good" alignments would make a contribution allowing us to infer that the sequences were related. Unfortunately, it is unclear how to modify the lowinfo algorithm to sum over all alignments, so the lowinfo alignment algorithm will be used to sum over parent sequences, but not over alignments.

To test the relatedness of two sequences believed to fit the model described in Section 5.3.1, the algorithm described in Section 5.6 was modified to sum over a large subset of the possible parent sequences. All possible parent sequences up to a length of $|S1| + |S2|$ are considered.

The modified algorithm sums over the R sequences instead of choosing the best R. To this end, the cells that contain the encoding lengths, shown in Figure 5.7, are calculated slightly differently. The calculation for these cells involves choosing the minimum of the encoding lengths, or equivalently choosing the maximum of the probabilities. The steps into these cells constitute different hypotheses for the R sequence and are mutually exclusive, so the probabilities can simply be added. For example, all possible steps into, say the 'T' cell, from the vertical direction have probabilities summed together. The same is done for the horizontal and diagonal directions. The direction (vertical, horizontal or diagonal) which has the highest associated probability is chosen, and that direction is used to determine the current cell.

The algorithm actually uses encoding lengths rather than probabilities. So to get the same effect as adding probabilities, the function $logplus(a, b) = -log_2(2^{-a} + 2^{-b})$ is used where a and b are encoding lengths. The naive implementation for $logplus(a, b)$ has problems with underflow for the encoding lengths typically used. A better implementation is to use $logplus(a, b) = a - log_2(1 + 2^{a-b})$ provided that $a < b$. If the encoding length a is significantly shorter than b, say $b - a > 50$, then a good approximation is $logplus(a, b) \approx a$. That is, for encoding lengths that are very different, the $logplus$ function behaves like the min function.

It is often desirable to have the mutation costs inferred by the algorithm rather than fixed by the user. It is possible to modify the new lowinfo algorithm to infer the mutation costs. This is done by having each cell of the D matrix carry intermediate values for the mutation costs. For a given cell, these costs are calculated from the mutation costs in the cell that this cell is calculated from. The mutation costs are updated depending on what type of mutation was considered and the cost of this operation versus the cost of the cheapest operation at this step.

The algorithm begins with some reasonable mutation costs. The mutation costs produced after the first pass are then taken as the new mutation costs, and the algorithm is re-started. This is a simple 'gradient descent' method guaranteed to converge, albeit possibly to a local minima. In practice three iterations of the algorithm was enough for convergence. This algorithm was implemented, and some results of testing for the relatedness of sequences is shown in Section 5.9.1.

5.8.1 Null Encoding

When using the lowinfo algorithm to sum over possible parent sequences, it is important to modify the null encoding similarly so that a proper comparison can be made. If the same null encoding were used as was used in Section 5.7 then the null encoding may be slightly longer than otherwise and the alignment would appear more probable than it should.

The algorithm for the null encoding of sequence $S1$ that is used when summing over parent sequences is essentially the same as the null encoding shown in Section 5.7. The single difference is that instead of choosing the minimum encoding length for each cell at position i, the $logplus$ function is used to combine the encoding lengths, in effect summing the probabilities. This can be visualized by again examining Figure 5.8 and replacing the min function with the $logplus$ function.

Performing this computation produces the same result as taking each possible parent sequence, R, of length equal to $|S1|$, encoding R with the model M, and encoding $S1$ as differences from R. Taking the probability of each of these encodings and summing them to give a single value for the probability that $S1$ has come from a mutated copy of an unknown sequence taken from M.

Another possibility for the null encoding would be to apply $m()$ directly to $S1$. This is a reasonable approximation to the above method, unless the sequence model is extremely biased and/or the mutation probabilities are high.

5.9 Results

A criterion is needed to compare alignments if one is to claim that some alignment is 'closer' to the true alignment. The method used for measuring the difference between two alignments is to measure the area between them as described in Section 5.2. An alignment defines a path from the top left of the DPA matrix to the bottom right (Figure 5.1), thus, there is some area between any two different alignments. This area is a measure of how close those two alignments are and it can deal with alignments involving insertions/deletion of two sequences of different length.

The lowinfo alignment algorithm (of Section 5.6) was implemented in the lowinfo alignment program. Another program was implemented to generate sequences by the method described in Section 5.3.1. The program to generate the data had M being a first order Markov model.

This Markov model was also included in the lowinfo alignment program. The transition probabilities for the Markov model are given in the table 5.3.

Table 5.3: The MM transition probabilities used in testing.

	A	T	G	C
A	0.11	0.09	0.3	0.5
T	0.09	0.11	0.5	0.3
G	0.5	0.3	0.09	0.11
C	0.5	0.11	0.09	0.3

For testing purposes, it was desirable to have just one mutation parameter, p. To this end, p was used to define the mutation probabilities as follows: $P_{match} = (1 - 2p)$, $P_{delete} = P_{mismatch} = p$, $P_{insert} = p$. The lowinfo alignment program used the same MM and mutation probabilities as the generating program. In the lowinfo alignment program an insertion event is separate from a deletion, mismatch or match thus $P_{match} + P_{delete} + P_{mismatch}$ should equal 1. The standard DPA alignment has all four events being possible at the same time so, $P_{match} + P_{delete} + P_{mismatch} + P_{insert}$ should equal 1. For this reason the DPA alignment program's mutation probabilities are set to $P_{match} = (1-3p)$, $P_{delete} = P_{mismatch} = P_{insert} = p$, which is different to that used for the lowinfo alignment program. Whether these are the best mutation probabilities to use for the DPA alignment is not clear, but they seem reasonable. Other mutation costs were experimented with for the DPA alignment program, but they had little effect on the results.

Sequences were generated with $P_y = 100/101$ (see Section 5.3.1). As an attempt to make the results only be a function of the mutation probability, p, and not the length of the sequences, only sequences whose length fell between 90 and 120 characters were used.

The generated sequences were aligned using both the standard alignment algorithm, and the lowinfo alignment algorithm. The area between the DPA alignment and the true alignment, and the area between the lowinfo alignment and the true alignment were calculated. Figure 5.9 shows results plotted versus the mutation probability, p, used to generate the sequences. Often there are a number of optimal alignments for both the DPA and the lowinfo algorithms. Since these algorithms consider all optimal alignments as equally good, the northernmost alignment was chosen arbitrarily.

Choosing the northernmost optimal alignment with the standard DPA algorithm is straightforward. The $min()$ function is simply modified so when two or three steps are equal[2], the

[2]We are dealing with floating point numbers, so values a and b are considered equal if $|a - b| < \epsilon$ where $\epsilon = 10^{-7}$.

step is chosen from this order of preference: vertical first, diagonal next and finally horizontal. Selecting the northernmost optimal alignment from the lowinfo alignment algorithm is more difficult. Essentially the same ordering on steps is done as for the standard DPA, however a complication arises in the new alignment algorithm from the fact that the final cell of the D matrix contains a number of states that can correspond to different alignments. So the choice of the final state can affect the chosen optimal alignment. The problem arises when a number of these final states have an equal best score. The state that has its last step in the most northerly direction is chosen. This is not guaranteed to be the most northerly optimal alignment, but it is believed to be so, in all but the most pathological cases.

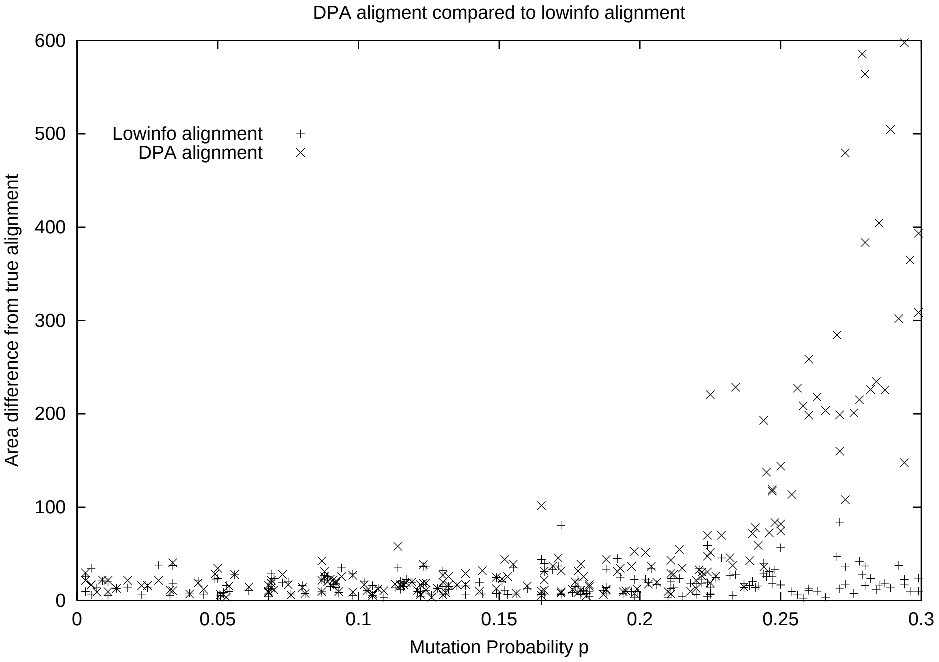

Figure 5.9: Comparison of the DPA alignment and the lowinfo alignment with the true alignment for 200 runs.

Another feature of alignment algorithms is the number of different optimal alignments they produce. Most applications require a single alignment out of the alignment algorithm, so having to choose from a number of different optimal alignments can be problematic. An arbitrary choice must often be made between them (above the northernmost alignment was arbitrarily chosen). So an alignment algorithm that produces similar optimal alignments is “better” than one that produces many different alignments (provided the true alignment is close to the optimal alignments). The method used for quantifying this similarity is to calculate the area between the northernmost and southernmost optimal alignments in the DPA matrix. Figure 5.1 illustrates the northernmost and southernmost optimal alignments. The area between them is a measure of the similarity of the optimal alignments produced by an alignment algorithm. This area will be called the envelope of optimal alignments. Figure 5.10 plots this envelope area for both the lowinfo and DPA alignment algorithms.

Table 5.4 gives a summary of the data plotted in Figures 5.9 and 5.10, except the data is collected from 1000 runs. From both plots, it is obvious that the DPA alignment algorithm deteriorates very rapidly when the mutation probability, p, is greater than about 0.2. For this reason Table 5.4 also shows a summary of the data when all data points with $p > 0.2$ are removed.

This data indicates that for sequences which fit the model described in Section 5.3.1, the lowinfo alignment algorithm produces alignments closer to the true alignment on average than the standard DPA alignment algorithm, especially when the rate of mutation is high. It is also evident that the optimal alignments found by the lowinfo alignment algorithm are 'closer' (by the envelope area measure) than those found by the standard DPA algorithm.

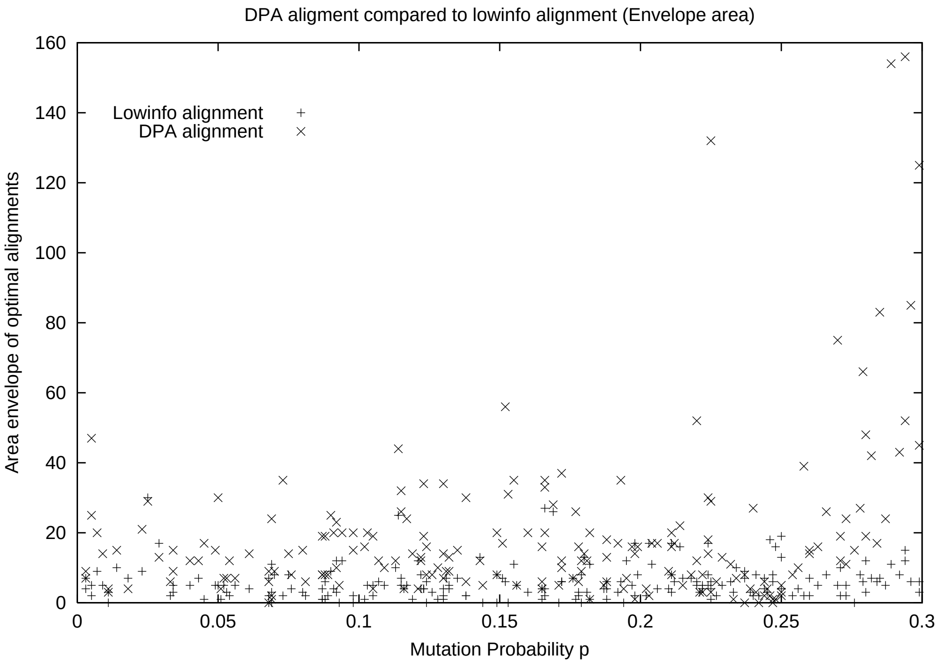

Figure 5.10: Comparison of the DPA alignment with lowinfo alignment for envelope of optimal alignments.

Table 5.4: Summary of results from Figures 5.9 and 5.10.

	All data		Probability $p < 0.2$	
	DPA	lowinfo	DPA	lowinfo
Total runs	1000	1000	652	652
Avg area between optimal and true alignments	70.7	18.6	19.7	16.7
Avg area in optimal alignment envelope	21.8	5.6	14.4	4.9

5.9.1 Results of Testing Relatedness

The lowinfo alignment algorithm that sums over parent sequences was used to measure the relatedness of different sequences. The Markov model used to generate the sequences is given in Table 5.5 (before normalization to give probabilities).

Table 5.5: The MM transition ratios used in testing relatedness of sequences.

	A	C	G	T
A	1	1	1	9
C	9	1	1	9
G	9	1	1	9
T	9	1	1	1

Pairs of sequences were generated from this model, so that each pair of sequences came from the same model but were otherwise unrelated. Each pair of sequences were then tested for relatedness using a standard alignment algorithm that implicitly assumes the sequences are random, and also with the lowinfo algorithm with prior knowledge of the Markov model. Sequences of length 1000 were generated and tests averaged over 100 trials. The results are summarised in Table 5.6. The first column shows the average difference between the encoding lengths for the null hypothesis and the optimal alignment. This difference is also known as the odd-ratio of the two hypotheses. A positive value implies the sequences are inferred to be related on average. The second column gives the standard deviation of these encoding length differences. The last two columns shows how many of the 100 trials the pairs of sequences were inferred to be unrelated and related.

As expected, using a standard alignment algorithm with the implicit uniform model of the sequences leads to many false positives when the sequences have a low information content. Table 5.7 gives results for the same set of tests, but with the unrelated sequences generated using a uniform model. This shows the behaviour of the algorithms is reversed. The new lowinfo algorithm always infers that the sequences are related while the algorithm using the uniform model correctly infers all sequence are unrelated. The conclusion from these test is that it is important to use a model for testing that is a good fit for the sequences being tested.

The tests summarised above are for unrelated sequences. The performance of the new algorithm on related sequence is shown by the next set of tests. In this set of tests a parent sequence was generated from either a uniform model or the Markov model given in Table 5.5. The parent sequence was then mutated to produce sequence $S1$, and mutated separately to produce sequence $S2$. Table 5.8 summarises the tests when the data was generated with a uniform model, and Table 5.9 when Markov model was used for generation. The first column of both tables shows which model was used for testing. The second column shows the mutation parameter, p, used in producing $S1$ and $S2$ from the parent sequence. The remaining columns summarise the results in a similar manner as the tables for the testing of unrelated sequences.

The first three rows of Table 5.8 is for sequences generated and tested with a uniform model and indicates results as expected. With a low mutation rate it is straightforward to infer the sequences are related while at higher mutation rates the evidence is insufficient to make this inference. The remaining three rows show that when using a Markov model to test these sequences, they are always inferred to be related. This is because the algorithm expected the sequences to come from mutations of the known Markov model, while the sequences in fact came from a uniform model. Therefore, the two sequences appear related.

The top half of Table 5.9 show testing assuming a uniform model with that were sequences generated with a Markov model. When the mutation rate is low, the sequences are similar enough that they are inferred to be related. However, as the mutation rate is increased there quickly becomes insufficient evidence for the uniform model to infer the sequences are related. The bottom half of the table shows similar behaviour, but the algorithm is able to find more evidence to infer relatedness at high mutation rates than when using the uniform model.

These results further illustrate the importance of using an appropriate model when testing the relatedness of sequences. Using an inappropriate model will produce misleading results. So for sequences that are incompressible, or uniformly random, using an algorithm that assumes a uniform model over the sequences is correct. While for sequences that contain

Table 5.6: Unrelated sequences generated from a Markov model.

	$-log_2$ odds-ratio		Number of sequences	
	average	std. deviation	Unrelated	Related
Uniform Model	13	13	11	89
Markov Model	-25	8	100	0

Table 5.7: Unrelated sequences generated from a Uniform model.

	$-log_2$ odds-ratio		Number of sequences	
	average	std. deviation	Unrelated	Related
Uniform Model	-40	12	100	0
Markov Model	118	21	0	100

some structure it is important to use an algorithm, such as the new lowinfo algorithm, that can correctly account for this structure.

Table 5.8: Related sequences generated from a Uniform model.

	Mutation parameter	$-log_2$ odds-ratio		Number of sequences	
		average	std. dev	unrelated	related
Uniform Model	0.05	57	17	0	100
	0.08	16	16	17	83
	0.12	-18	15	91	9
Markov Model	0.05	142	19	0	100
	0.08	110	20	0	100
	0.12	101	17	0	100

Table 5.9: Related sequences generated from a Markov model.

	Mutation parameter	$-log_2$ odds-ratio		Number of sequences	
		average	std. dev	unrelated	related
Uniform Model	0.05	59	17	0	100
	0.08	16	16	15	85
	0.12	-11	13	84	16
Markov Model	0.05	26	16	4	96
	0.08	5	13	31	69
	0.12	6	13	34	66

5.10 Conclusion

Standard alignment algorithms do not take into account the information content (or structure) of the sequences. The high information regions of sequences should be aligned more accurately than low information regions because they are less likely to occur by chance. In this chapter a new algorithm has been presented for performing sequence alignment in quadratic time. This new algorithm takes into account the randomness of a ‘parent’ sequence as given by a Markov model. The two sequences to be aligned are considered noisy observations of this parent sequence.

When the parent sequence is of no interest, it is a nuisance parameter. It was shown how to modify the new low information alignment algorithm to sum over the different possible parent sequences. A comparison was made of this new alignment algorithm and a standard alignment algorithm that demonstrated that this new algorithm finds a better alignment when the data matches the hypothesized model.

Using a standard alignment algorithm to find related sequences can lead to both false positives and false negatives if the sequences being compared have low information content. False negatives (sequences that are related but inferred to be unrelated) can arise if similar, high information regions, occur in both sequence. These high information regions give strong evidence that the sequences are related only if the sequence model is taken into consideration. It was shown that the new low information alignment algorithm can measure the relatedness of such sequences much more accurately. Sample runs on test data comparing the new algorithm to the standard alignment algorithm were shown that support this claim. When one cares only about the relatedness of the sequences, then the alignment itself is also a nuisance parameter. An extension of the lowinfo algorithm to sum over all possible alignments would be useful, however it is not clear how to do this.

In this chapter the lowinfo alignment algorithm used a zeroth or first order Markov model to model the parent sequence. It would possible to extend the algorithm to higher order Markov models, at the cost of the running time of the algorithm. For an order-n Markov model, and an alphabet size of $|\Sigma|$, there would have to be $2|\Sigma|^n$ states in each cell of the DPA matrix. It may also be possible to extend the parent sequence model to models other than Markov models. However, if $m()$ can not be used incrementally (i.e. encode the sequence $S[1..i]$ in $O(1)$ time given the encoding of $S[1..i-1]$), an optimal search is unlikely to remain $O(n^2)$. It is likely that some stochastic type search would be necessary for the more complex models.

Chapter 6

Conclusion

This thesis presented new algorithms for the sequence alignment problem. To understand these new algorithms, it is necessary to be familiar with many of the alignment algorithms from earlier literature. In Chapter 1 those algorithms that are required for understanding the later chapters were described. Of these algorithms, the important ones were discussed in detail. The simplest alignment problem involves two sequences with Levenshtein costs, that is, where a match has a cost of 0 and each point mutation has a cost of 1. This distance measure is also known as the Levenshtein distance [33]. The simplest algorithm for this alignment problem is a dynamic programming algorithm referred to in this thesis as the DPA2s algorithm. This algorithm requires $O(n^2)$ space and time to find an optimal alignment for sequences of length n.

In this thesis the focus of the alignment algorithms has been on biological sequences, particularly nucleotide sequence data, that is, DNA/RNA sequences. This thesis has concentrated on the computational aspect of these algorithms, especially the time and space complexity. For many alignment problems, and particularly for this type of biological data, linear gap costs are a better model than simple costs. Compared to simple costs, linear gap costs tends to favour fewer, longer gaps in an alignment. This is favourable for biological sequence data where some mutation events may insert or delete whole subsequences at a time. The DPA2l algorithm [20] was explained in Section 1.2.2 for optimally aligning two sequences using linear gap costs. Like the DPA2s algorithm, the DPA2l algorithm takes $O(n^2)$ time and space.

An improvement on the DPA2s algorithm was developed by Ukkonen [62, 63] — originally for the related, longest common subsequence (LCS) problem. Ukkonen's algorithm for two sequences and simple costs, referred to here as the Ukk2s algorithm, has an average time complexity of $O(d^2 + n)$ and space complexity of $O(d^2)$ where d is the edit cost of the sequences. Thus the Ukk2s algorithm runs much faster than the DPA2s algorithm for sequences which are similar. The Ukk2s algorithm was discussed in detail in Section 1.3. Applying Ukkonen's algorithm to linear gap costs [39] was also discussed and is referred to as the Ukk2l algorithm.

Ukkonen's algorithm requires that a match has a cost of zero, and that all other mutation costs are small positive integers. When using simple costs there are two degrees of freedom in choosing the actual values for the costs [3]. All costs can be multiplied by a constant, and all costs can have a constant per character added to them. These two transformations do not affect the rank order of alignments. Thus it is often possible to modify arbitrary costs to meet the requirements of Ukkonen's algorithm. Using these costs transformations, it is possible to modify costs so that a probabilistic view can be taken. The costs can be transformed in such a way that they can be viewed as negative logarithms of probabilities. Therefore, each mutation cost can be converted to a *mutation probability*. Where algorithms add costs together, this can be interpreted as taking the product of the related probabilities. Under this probabilistic view, most alignment algorithms have a uniform distribution over the characters of the sequences to be aligned. This need not be the case. Chapter 5 developed an algorithm for aligning sequences that are noisy observations of an unknown parent sequence that does not come from a uniform distribution.

Optimal alignment of three sequences simultaneously can produce a more robust alignment than using pairwise alignments. Three-way alignment is particularly useful for generating evolutionary trees from biological sequence data. The internal nodes of an evolutionary tree represents an unknown ancestor sequence. Each internal node has three neighbour sequences. Three-way alignment can be used to align the neighbour sequences and to make an inference for the sequence at the internal node. The extension of the DPA2s algorithm to three sequences is obvious, resulting in the algorithm referred to as the DPA3s algorithm which has a time and space complexity of $O(n^3)$. Three-way alignment may be done using all-pairs costs or star costs. Star costs are useful for the evolutionary tree problem, and they have been the main focus of this thesis for three sequences. Ukkonen's algorithm has also been applied to the alignment problem for three sequences with simple costs [2] resulting in the Ukk3s algorithm which optimally aligns three sequences in $O(d^3)$ space and $O(d^3 + n)$ time on average.

In Chapter 3, two new algorithms were developed for the alignment of three sequences with a star based mutation model and linear gap costs. Under the star mutation model the three sequences are assumed to be mutated copies of a hypothetical parent sequence. This model is particularly useful for evolutionary trees where each node has three neighbour sequences and one unknown internal sequence. It was shown how linear gap costs relate to a mutation model that can be expressed as a probabilistic three-state finite state machine (FSM). It is necessary for three-way alignment with linear gap costs to combine three of these three-state FSMs resulting in a twenty-seven state FSM.

The first new algorithm presented in Chapter 3, was the DPA3l algorithm which is a generalization of linear gap costs to three sequence using star costs. This algorithm finds an optimal alignment in $O(n^3)$ time and space. Gotoh's three sequence algorithm [21] for linear costs was shown to be a special case of the DPA3l algorithm. An example alignment was shown to demonstrate that the DPA3l algorithm is able to find a more desirable alignment that Gotoh's algorithm.

The second new algorithm from Chapter 3, and the major contribution of that chapter, was the Ukk3l algorithm. This algorithm is for the same alignment problem as the DPA3l algorithm, but makes use of the Ukkonen style speed-up. The Ukk3l algorithm runs much faster, and uses less memory than the DPA3l algorithm when aligning similar sequences. Sample runs of the Ukk3l algorithm were used to demonstrate the space complexity of the Ukk3l algorithm is $O(d^3)$ and the average time complexity behaves as $O(d^3 + n)$.

Hirschberg [27] developed a technique to modify the DPA2s algorithm using a divide-and-conquer technique to determine an optimal alignment requiring only $O(n)$ space. This was discussed in detail in Section 1.2.3. This technique has also been applied to the DPA2l algorithm [38] to produce an algorithm that requires only linear space.

Hirschberg's technique can, in theory, be applied to all the aforementioned algorithms, however it becomes quite difficult for the complicated alignment algorithms. Chapter 2 presented the check-point method for reducing the space requirement for a number of alignment algorithms. It was shown how to apply the check-point method to the DPA2s, DPA2l, Ukk2s and Ukk2l algorithms. For the DPA style algorithms, the space complexity is reduced to $O(n)$ while maintaining the time complexity at $O(n^2)$. The Ukkonen style algorithms have the space complexity reduced to $O(d)$ and an average time complexity of $O(d^2 + n \log d)$. Plots of sample runs were shown that confirm these time and space complexities.

The check-point method has the advantage of simplicity over Hirschberg's technique. Another major advantage of the check-point method is that more check-points can be kept to improve the run time, at the cost of using more space. This only affects the constants in the time and space complexity of the algorithm, but it is useful in practice to be able to trade memory usage for run time.

Chapter 4 combined the Ukk3l algorithm and the check-point method of the two preceding chapters resulting in the Ukk3l_cp algorithm. This new algorithm was shown to have the same advantages as the Ukk3l algorithm while having a reduced space complexity of $O(d^2)$. This reduction in space complexity came at the cost of slightly increasing the time complexity to $O(d^3 + n \log d)$. The Ukk3l algorithm is quite complex, and the application of the check-point method has some complications. These complications were discussed in detail. The development of the Ukk3l_cp algorithm illustrated an advantage of the check-point method over Hirschberg's technique because it is difficult to imagine being able to apply Hirschberg's technique to the Ukk3l algorithm.

All the aforementioned algorithms have the implicit assumption that the sequences being aligned are random. The aim of Chapter 5 was to remove this assumption. An algorithm was developed for aligning two sequences that are assumed to be mutated copies of an unknown parent parent sequence. This parent sequence may be non-random in the sense that it comes from a non-uniform sequence model, M. The parts of the sequences that are less likely to occur by chance, high information regions, have more influence on the alignment than regions that are likely to occur by chance, low information regions. This alignment algorithm was named the *lowinfo* alignment algorithm. The lowinfo alignment algorithm is for the specific case where the model, M, of the parent sequence is a first (or zeroth) order Markov Model. Test data showed the lowinfo alignment algorithm does indeed produce a better alignment than the standard alignment algorithms when the sequences are of low information content. This becomes much more obvious when there is a higher mutation distance between the sequences.

In Chapter 5, it was shown how to modify the lowinfo alignment algorithm to take into account most of the likely parent sequences when determining the alignment. Each possible parent sequence gives a weight to the alignment it implies based on how likely that parent sequence is. The lowinfo algorithm was then used to test the relatedness of sequences. It was shown that for unrelated sequences from a non-random source, the new algorithm correctly inferred they were unrelated; whereas a standard alignment algorithm not taking into account the low information content of the sequence often incorrectly inferred the sequences were related.

The beginning of this thesis summarised a number of different alignment algorithms that are related to, or necessary for an understanding of the new algorithms that are later developed. The simple but versatile check-pointing method was developed in Chapter 2 and it was shown how to apply this method to a number of alignment algorithms to reduce the space complexity. This work has been published in Information Processing Letters [42]. The two new alignment algorithms for three sequences and linear gap costs developed in Chapter 3 have been published in the Journal of Theoretical Biology [43]. Chapter 4 introduced a new algorithm combining the new contributions of the two previous chapters resulting in a fast algorithm for three sequences with linear gap costs which requires only $O(d^2)$ space. In Chapter 5 a new algorithm was developed to align sequences while accounting for the fact that the sequences may be non-random. This new algorithm was presented at the Australian Computer Science Theory Symposium [44].

Bibliography

[1] P. Agarwal and D. J. States. The repeat pattern toolkit (RPT): Analyzing the structure and evolution of the C. elegans genome. In *Second International Conference on Intelligent Systems for Molecular Biology*, pages 1–9, 1994.

[2] L. Allison. A fast algorithm for the optimal alignment of three strings. *Journal of Theoretical Biology*, 164:261–269, 1993.

[3] L. Allison. Normalization of affine gap costs used in optimal sequence alignment. *Journal of Theoretical Biology*, 161:263–269, 1993.

[4] L. Allison, D. R. Powell, and T. I. Dix. Compression and approximate matching. *The Computer Journal*, 42(1):1–10, 1999.

[5] L. Allison, C. S. Wallace, and C. N. Yee. Finite-state models in the alignment of macromolecules. *Journal of Molecular Evolution*, 35:77–89, 1992.

[6] L. Allison, C. S. Wallace, and C. N. Yee. Minimum message length encoding, evolutionary trees and multiple-alignment. *25th Hawaii International Conference on System Sciences*, 1:663–674, 1992.

[7] S. Altschul and B. Erickson. Optimal sequence alignments using affine gap costs. *Bulletin of Mathematical Biology*, 48(5-6):603–616, 1986.

[8] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Theoretical Biology*, 215:403–410, 1990.

[9] S. F. Altschul and D. J. Lipman. Trees, stars and multiple biological sequence alignment. *SIAM J. appl. Math.*, 49(1):197–209, February 1989.

[10] D. J. Burr. Designing a handwriting reader. *5th International Conference on Pattern Recognition*, 1980.

[11] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM J. appl. Math.*, 48(5):1073–1082, October 1988.

[12] K.-M. Chao, J. Z. amd James Ostell, and W. Miller. A tool for aligning very similar DNA sequences. *CABIOS*, 13(1):75–80, 1997.

[13] K.-L. Chung. A fast algorithm for stereo matching. *Information Processing Letters*, 63:57–61, 1997.

[14] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5:345–352, 1978.

[15] R. S. M. Dayhoff. Matrices for detecting distant relationships. In M. Dayhoff, editor, *Atlas of Protein Sequences*, pages 353–358. National Biomedical Research Foundation, 1979.

[16] N. R. Dixon and T. B. Martin, editors. *Automatic speech and speaker recognition*. IEEE Press, 1979.

[17] W. M. Fitch. Random sequences. *Journal of Molecular Biology*, 163:171–176, 1983.

[18] Y. Fujimoto and *et al*. Recognition of handprinted characters by nonlinear elastic matching. *3rd International Joint Conference on Pattern Recognition*, pages 113–119, 1976.

[19] W. Gish and D. States. Identification of protein coding regions by database similarity search. *Nature Genetics*, 3:266–272, 1993.

[20] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[21] O. Gotoh. Alignment of three biological sequences with an efficient traceback procedure. *Journal of Theoretical Biology*, 121:327–337, 1986.

[22] O. Gotoh. Pattern matching of biological sequences with limited storage. *CABIOS*, 3:17–20, 1987.

[23] J. Grice, R. Hughey, and D. Speck. Reduced space sequence alignment. *CABIOS*, 13(1):45–53, 1997.

[24] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Academy Science*, 89(10):915–919, 1992.

[25] D. G. Higgins and P. M. Sharp. CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73:237–244, 1988.

[26] M. Hirosawa, M. Hoshida, M. Ishikawa, and T. Toya. MASCOT: multiple alignment system for protein sequences based on three-way dynamic programming. *CABIOS*, 9(2):161–167, 1993.

[27] D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[28] D. Hirschberg. Serial computations of Levenshtein distances. In A. Apostolico and Z. Galil, editors, *Pattern Matching Algorithms*, pages 123–141. Oxford University Press, 1997.

[29] P. Hogeweg and B. Hesper. The alignment of sets of sequences and the construction of phyletic trees: an integrated method. *Journal of Molecular Evolution*, 20:175–186, 1984.

[30] L. Holm and C. Sander. Protein structure comparison by alignment of distance matrices. *Journal of Theoretical Biology*, 233:123–138, 1993.

[31] X. Huang. A contig assembly program based on sensitive detection of fragment overlaps. *Genomics*, 14:18–25, 1992.

[32] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov Models in computational biology: Applications to protein modeling. *Journal of Molecular Biology*, 235:1501–1531, 1994.

[33] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, 10(8):707–710, 1966.

[34] W. Miller and E. W. Myers. Sequence comparison with concave weighting functions. *Bulletin of Mathematical Biology*, 50(2):97–120, 1988.

[35] A. Milosavljević and J. Jurka. Discovering simple DNA sequences by the algorithmic significance method. *CABIOS*, 9(4):407–411, 1993.

[36] S. Mita, S. Maeda, K. Shimada, and S. Araki. Cloning and sequence analysis of cDNA for human prealbumin. *Biochem. Biophys. Res. Commun.*, 124:558–564, 1984.

[37] E. W. Myers. An O(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[38] E. W. Myers and W. Miller. Optimal alignments in linear space. *CABIOS*, 4(1):11–17, 1988.

[39] E. W. Myers and W. Miller. Row replacement algorithms for screen editors. *ACM Transactions on Programming Languages and Systems*, 11(1):33–56, 1989.

[40] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[41] C. Notredame, L. Holm, and D. G. Higgins. COFFEE: an objective function for multiple sequence alignments. *Bioinformatics*, 14(5):407–422, 1998.

[42] D. R. Powell, L. Allison, and T. I. Dix. A versatile divide and conquer technique for optimal string alignment. *Information Processing Letters*, 70(3):127–139, 1999. http://www.elsevier.nl/gej-ng/10/23/20/48/19/19/abstract.html.

[43] D. R. Powell, L. Allison, and T. I. Dix. Fast, optimal alignment of three sequences using linear gap costs. *Journal of Theoretical Biology*, 207(3):325–336, December 2000. http://www.idealibrary.com/links/doi/10.1006/jtbi.2000.2177.

[44] D. R. Powell, L. Allison, T. I. Dix, and D. L. Dowe. Alignment of low information sequences. *Australian Computer Science Theory Symposium*, pages 215–230, 1998.

[45] D. R. Powell, D. L. Dowe, L. Allison, and T. I. Dix. Discovering simple DNA sequences by compression. *Pacific Symposium on Biocomputing*, pages 597–608, January 1998. http://www-smi.stanford.edu/projects/helix/psb98/powell.pdf.

[46] L. R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, pages 4–16, January 1986.

[47] E. Reiner et al. Characterization of normal human cells by pyrolysis gas chromatography mass spectrometry. *Biomedical Mass Spectrometry*, 6(11):491–498, 1979.

[48] E. Rivals, O. Delgrange, J.-P. Delahaye, M. Dauchet, M.-O. Delorme, A. Hénaut, and E. Ollivier. Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in DNA sequences. *CABIOS*, 13(2):131–136, 1997.

[49] D. Sankoff. Minimal mutation trees of sequences. *SIAM J. Appl. Math.*, 28(1):35–42, January 1975.

[50] D. Sankoff, C. Morel, and R. J. Cedergren. Evolution of 5S RNA and the non-randomness of base replacement. *Nature New Biology*, 245:232–234, OCT 1973.

[51] H. Sasaki, Y. Sakaki, H. Matsuo, I. Goto, Y. Kuroiwa, I. Sahashi, A. Takahashi, T. Shinoda, T. Isobe, and Y. Takagi. Diagnosis of familial amyloidotic polyneuropathy by recombinant DNA techniques. *Biochem. Biophys. Res. Commun.*, 125:636–642, 1984.

[52] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26(4):787–793, 1974.

[53] C. E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423,623–656, 1948.

[54] R. Staden. A new computer method for the storage and manipulation of DNA gel reading data. *Nucleic Acids Research*, 8(16):3673–3694, 1980.

[55] R. Staden. Automation of the computer handling of gel reading data produced by the shotgun method of DNA sequencing. *Nucleic Acids Research*, 10:4731–4751, 1982.

[56] J. Sundelin, H. Melhus, S. Das, U. Eriksson, P. Lind, L. Traegaardh, P. A. Peterson, and L. Rask. The primary structure of rabbit and rat prealbumin and a comparison with the tertiary structure of human prealbumin. *J. Biol. Chem.*, 260:6481–6487, 1985.

[57] P. Taylor. A fast homology program for aligning biological sequences. *Nucleic Acids Research*, 12:447–455, 1984.

[58] W. R. Taylor. A flexible method to align large numbers of biological sequences. *Journal of Molecular Evolution*, 28:161–169, 1988.

[59] W. R. Taylor and C. A. Orengo. A holistic approach to protein structure alignment. *Protein Engineering*, 2(7):505–519, 1989.

[60] W. R. Taylor and C. A. Orengo. Protein structure alignment. *Journal of Theoretical Biology*, 208:1–22, 1989.

[61] J. Thompson, D. Higgins, and T. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4690, 1994.

[62] E. Ukkonen. On approximate string matching. *Foundations of Computation Theory*, 158:487–495, 1983.

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

[64] S. Wakasugi, S. Maeda, K. Shimada, H. Nakashima, and S. Migita. Structural comparisons between mouse and human prealbumin. *J. Biochem.*, 98:1707–1714, 1985.

[65] K. Watanabe, Y. Urano, and T. Tamaoki. Optimal alignments of biological sequences on a microcomputer. *CABIOS*, 1:83–87, 1985.

[66] J. C. Wootton. Simple sequences of protein and DNA. In M. J. Bishop and C. J. Rawlings, editors, *DNA and Protein Sequence Analysis*, pages 169–183. IRL Press, 1997.

[67] J. C. Wootton and S. Federhen. Statistics of local complexity in amino acid sequences and sequence databases. *Computers and Chemistry*, 17(2):149–163, 1993.

Appendix A

Sample Alignment of the Transthyretin Gene

The following is an optimal alignment of sequences from the Transthyretin gene from a mouse, a rat and a human. The GenBank IDs of the sequences are MMALBR [64], RATPALTA [56] and HUMPALA [36, 51]. The sequence length for the MMALBR, RATPALTA and HUMPALA sequences are 614, 595 and 615 characters respectively. The mutation costs used for the alignment are: 0 for a match, 1 for a mismatch, 3 to start a gap, 1 to continue a gap. Under these cost, the edit cost of the three sequences is 233. The optimal alignment is shown below with mismatches between sequences highlighted with an exclamation character.

The test runs were performed on a 1.2GHz AMD Athlon processor with 512 Megabytes of RAM and about 1 Gigabyte of swap space.

The new Ukk3l algorithm from Chapter 3 aligned these sequence in about 15.5 minutes of CPU time, but 32.8 minutes of real time. This low CPU utilisation is due to the fact the process had to swap heavily since it used about 1 Gigabyte of memory and allocated 1.2 Gigabytes.

The edit cost only version of the Ukk3l algorithm which does not actually output an alignment, found the optimal edit cost in 15.5 minutes of CPU time and about the same real-time. About 80 Megabytes of memory were allocated of which about 69 Megabytes were used.

The Ukk3l_cp algorithm from Chapter 4 was also used to align these sequences. The alignment took 17.2 CPU-minutes and 17.6 minutes of real elapsed time. About 273 Megabytes of memory were allocated while 235 Megabytes were actually used by the Ukk3l_cp program. It is interesting to note that while the Ukk3l_cp algorithm is in theory slower than the

Ukk3l algorithm, in this test the Ukk3l_cp program ran faster due to the fact that it did not require to swap virtual memory pages to and from disk.

These sequences could not be aligned using the DPA-based algorithm since around 7 Gigabytes of memory would be needed and a projected runtime of over 2 CPU-hours.

A consensus sequence can be chosen from the three-way alignment. The edit cost of each sequence with this consensus sequence can be calculated using a two sequence, linear gap algorithm. This was done using a program implementing the DPA2l algorithm. The edit cost between the consensus sequence and the individual sequences are as follows: MMALBR, 25; RATPALTA, 54; HUMPALA 154.

Appendix B shows a three way alignment of these sequence using Levenshtein costs, and Appendix C shows the three pair-wise alignments using linear gap costs.

```
HUMPALA     0: acagaagtccactcattcttggcaggatggcttctcatcgtctgctcctcct
                  ! !!     !!!!  !  !            ! !   !  !!
MMALBR      0: acacagatccacaagctcctgacaggatggcttcccttcgactcttcctcct
               !!!!!!!!!!!!!!!!!                       !  !
RATPALTA    0: -----------------cctgacaggatggcttcccttcgcctgttcctcct

HUMPALA    52: ctgccttgctggactggtatttgtgtctgaggctggccctacgggcaccggt
               !     !                       !        !!    !! !  !
MMALBR     52: ttgcctcgctggactggtatttgtgtctgaagctggccccgcgggtgctgga
               !               !      !               ! !
RATPALTA   35: ctgcctcgctggactgatatttgcgtctgaagctggccctgggggtgctgga

HUMPALA   104: gaatccaagtgtcctctgatggtcaaagttctagatgctgtccgaggcagtc
                       !                    !  !                 !
MMALBR    104: gaatccaaatgtcctctgatggtcaaagtcctggatgctgtccgaggcagcc
                       !
RATPALTA   87: gaatccaagtgtcctctgatggtcaaagtcctggatgctgtccgaggcagcc

HUMPALA   156: ctgccatcaatgtggccgtgcatgtgttcagaaaggctgctgatgacacctg
                   !! !! !     !  !! !       !    ! !!    ! !!!
MMALBR    156: ctgctgtagacgtggctgtaaaagtgttcaaaaagacctctgagggatcctg
                      !  !     !  !             !   !! !  !   !!
RATPALTA  139: ctgctgtcgatgtggccgtgaaagtgttcaaaaggactgcagacggaagctg

HUMPALA   208: ggagccatttgcctctgggaaaaccagtgagtctggagagctgcatgggctc
                     !              !   !!!                 !
MMALBR    208: ggagccctttgcctctgggaagaccgcggagtctggagagctgcacgggctc
                     !                    !
RATPALTA  191: ggagccgtttgcctctgggaagaccgccgagtctggagagctgcacgggctc
```

```
HUMPALA   260: acaactgaggaggaatttgtagaagggatatacaaagtggaaatagacacca
                 !  !  !   ! !           !! !    !   !   ! !
MMALBR    260: accacagatgagaagtttgtagaaggagtgtacagagtagaactggacacca
                                !!!!     !        !
RATPALTA  243: accacagatgagaagttcacggaaggggtgtacagggtagaactggacacca


HUMPALA   312: aatcttactggaaggcacttggcatctccccattccatgagcatgcagaggt
                   !         !          !     !        !!!!  !  !
MMALBR    312: aatcgtactggaagacacttggcatttccccgttccatgaattcgcggatgt
                   !         ! !              !          !   !  !
RATPALTA  295: aatcatactggaaggctcttggcatttccccattccatgaatacgcagaggt


HUMPALA   364: ggtattcacagccaacgactccggcccccgccgctacaccattgccgccctg
                  !                 !    !!    !         !  !
MMALBR    364: ggttttcacagccaacgactctggccatcgccactacaccatcgcagccctg
                              !        !
RATPALTA  347: ggttttcacagccaatgactctggtcatcgccactacaccatcgcagccctg


HUMPALA   416: ctgagcccctactcctattccaccacggctgtcgtcaccaatcccaaggaat
                 !     !        !!!                 !   !   !  ! !
MMALBR    416: ctcagcccatactcctacagcaccacggctgtcgtcagcaacccccagaatt
                       !                 !           !           !
RATPALTA  399: ctcagcccgtactcctacagcaccactgctgtcgtcagtaacccccagaact


HUMPALA   468: gagggacttctcctccagtggacctgaaggacgagggatgggatttcatgta
                  !    !!!  !!!!!!!!!!!!!!!!  !    !!!!    ! !!!!!
MMALBR    468: gagagactcagcc----------------caggaggaccaggatcttgccaa
                  !   !                       !        !
RATPALTA  451: gagggacccagcc----------------cacgaggaccaagatcttgccaa


HUMPALA   520: accaagagtattccatttttactaaagcactgttttcacctcatatgctatg
                !  !!  !  !      !   !   !  !    ! !! !!!   !! !!
MMALBR    504: agcagtagcatcccatttgtaccaaaacagtgttcttgctctataaaccgtg
                        !            !!
RATPALTA  487: agcagtagc-tcccatttgtactgaaacagtgttcttgctctataaaccgtg


HUMPALA   572: ttagaagtccaggcagagacaataaaacattcctgtgaaa------------
                   !  !!    !   !! !! !  !     ! ! !   !!!!!!!!!!!!
MMALBR    556: ttagcagctcaggaagatgccgtgaagcattcttattaaaccacctgctatt
                     !   !               ! !                 !!
RATPALTA  538: ttagcaactcgggaagatgccgtgaaacgttcttattaaaccacct-ttatt
```

```
HUMPALA   612: ---ggc
                !!!!!
MMALBR    608: tcattc

RATPALTA  589: tcattc
```

Appendix B

Simple Costs Alignment of the Transthyretin Genes

The following is the Levenshtein costs alignment of the same sequence as in Appendix A, namely, the Transthyretin genes from a human (HUMPALA), mouse (MMALBR) and rat (RATPALTA). This alignment was produced using the Ukk3l_cp program with a cost of 0 for match, 1 for change, 0 to start a gap and 1 to continue a gap. So while a linear gap costs algorithm was used, the costs were set to Levenshtein costs. The edit cost of these sequence using Levenshtein costs is 207.

It is interesting to compare this alignment with the alignment in Appendix A. The alignment with simple costs, as expected, has more, short gaps than the alignment with linear gap costs. A clear example of this can be seen by looking around the 480th character. The alignment in this region seems much more plausible in the linear gap cost alignment than in the simple cost alignment.

```
HUMPALA     0: acaga-agtccac--tcattc-ttggcaggatggcttctcatcgtctgctcc
                  ! ! !     !!! ! ! ! ! !            ! !   !  !!
MMALBR      0: acacaga-tccacaagc-t-cct-gacaggatggcttcccttcgactcttcc
               ! ! !!!  !!!!!! ! ! !!! !                   !  !
RATPALTA    0: -c-c----t------g---------acaggatggcttcccttcgcctgttcc


HUMPALA    48: tcctctgccttgctggactggtatttgtgtctgaggctggccctacgggcac
                   !     !                       !        !!    !!
MMALBR     48: tcctttgcctcgctggactggtatttgtgtctgaagctggccccgcgggtgc
                   !               !      !               ! !
RATPALTA   31: tcctctgcctcgctggactgatatttgcgtctgaagctggccctgggggtgc
```

```
HUMPALA   100: cggtgaatccaagtgtcctctgatggtcaaagttctagatgctgtccgaggc
               !  !        !                    !  !
MMALBR    100: tggagaatccaaatgtcctctgatggtcaaagtcctggatgctgtccgaggc
                           !
RATPALTA   83: tggagaatccaagtgtcctctgatggtcaaagtcctggatgctgtccgaggc


HUMPALA   152: agtcctgccatcaatgtggccgtgcatgtgttcagaaaggctgctgatgaca
                 !     !! !! !     !  !! !       !    ! !!    ! !!!
MMALBR    152: agccctgctgtagacgtggctgtaaaagtgttcaaaaagacctctgagggat
                          !  !     !  !             !   !! !  !   !
RATPALTA  135: agccctgctgtcgatgtggccgtgaaagtgttcaaaaggactgcagacggaa


HUMPALA   204: cctgggagccatttgcctctgggaaaaccagtg-agtctggagagctgcatg
                         !              !   ! ! !                !
MMALBR    204: cctgggagccctttgcctctgggaagacc-gcggagtctggagagctgcacg
               !         !                     !
RATPALTA  187: gctgggagccgtttgcctctgggaagacc-gccgagtctggagagctgcacg


HUMPALA   255: ggctcacaactgaggaggaatttgtagaagggatatacaaagtggaaataga
                      !  !  !   ! !           !! !    !   !   ! !
MMALBR    255: ggctcaccacagatgagaagtttgtagaaggagtgtacagagtagaactgga
                                     !!!!     !        !
RATPALTA  238: ggctcaccacagatgagaagttcacggaaggggtgtacagggtagaactgga


HUMPALA   307: caccaaatcttactggaaggcacttggcatctccccattccatgagcatgca
                        !         !          !     !        !!!!  !
MMALBR    307: caccaaatcgtactggaagacacttggcatttccccgttccatgaattcgcg
                        !         ! !              !          !   !
RATPALTA  290: caccaaatcatactggaaggctcttggcatttccccattccatgaatacgca


HUMPALA   359: gaggtggtattcacagccaacgactccggcccccgccgctacaccattgccg
                 !     !                 !    !!    !         !  !
MMALBR    359: gatgtggttttcacagccaacgactctggccatcgccactacaccatcgcag
                 !                 !        !
RATPALTA  342: gaggtggttttcacagccaatgactctggtcatcgccactacaccatcgcag


HUMPALA   411: ccctgctgagcccctactcctattccaccacggctgtcgtcaccaatcccaa
                      !     !        !!!                 !   !   !
MMALBR    411: ccctgctcagcccatactcctacagcaccacggctgtcgtcagcaacccca
                            !                 !           !
RATPALTA  394: ccctgctcagcccgtactcctacagcaccactgctgtcgtcagtaacccca




HUMPALA   463: ggaat-gagggacttctcctccagtggacctgaaggacgagggatgggattt
                !   !   !   !  !! !    ! ! !!! !! !  !   !  !!!!  !
MMALBR    463: g-aattgagagac-tcagc-ccag-g-a---g--g-accagg-at---cttg
                   !    !    !        !                !
```

```
RATPALTA  446: g-aactgagggac-ccagc-ccac-g-a---g--g-accaag-at---cttg


HUMPALA   514: catgtaaccaagagtattccatttttactaaagcactgttttcacctc-ata
                !!!!  !  !!  !  !      !   !   !  !    ! !! !  !
MMALBR    500: c--caaagcagtagcatcccatttgtaccaaaacagtgttcttgc-tctata
                             !             !!
RATPALTA  483: c--caaagcagtagc-tcccatttgtactgaaacagtgttcttgc-tctata


HUMPALA   565: tgctatgttag-aagtccaggcagagacaataaaacattcctgtgaaa----
               !! !!      ! !!  !   !   !! !! !  !     ! ! !   !!!!
MMALBR    549: aaccgtgttagcagctc-aggaagatgccgtgaagcattcttattaaaccac
                           !    !              ! !
RATPALTA  531: aaccgtgttagcaactc-gggaagatgccgtgaaacgttcttattaaaccac


HUMPALA   612: --g-----gc----
               !! !!!!!! !!!!
MMALBR    600: ctgctatttcattc
                 !!
RATPALTA  582: ctt-tatttcattc
```

Appendix C

Pair-wise Alignments of the Transthyretin Genes

Following are three pair-wise alignment using linear gap costs for the Transthyretin genes from a human (HUMPALA), mouse (MMALBR) and rat (RATPALTA). These alignments were produced using a program implementing the DPA2l algorithm with mutations costs of: 0 for a match, 1 for a mismatch, 3 to start a gap, and 1 to continue a gap.

The edit cost between each pair is as follows: MMALBR and RATPALTA, 79; MMALBR and HUMPALA, 176; RATPALTA and HUMPALA 193. Which shows that, as expected, the gene from the rat and the gene from mouse are more closely related than they are to the human gene.

```
MMALBR       0: acacagatccacaagctcctgacaggatggcttcccttcgactcttcctcct
                !!!!!!!!!!!!!!!!!                       !  !
RATPALTA     0: -----------------cctgacaggatggcttcccttcgcctgttcctcct


MMALBR      52: ttgcctcgctggactggtatttgtgtctgaagctggccccgcgggtgctgga
                !               !      !               ! !
RATPALTA    35: ctgcctcgctggactgatatttgcgtctgaagctggccctgggggtgctgga


MMALBR     104: gaatccaaatgtcctctgatggtcaaagtcctggatgctgtccgaggcagcc
                        !
RATPALTA    87: gaatccaagtgtcctctgatggtcaaagtcctggatgctgtccgaggcagcc



MMALBR     156: ctgctgtagacgtggctgtaaaagtgttcaaaaagacctctgagggatcctg
                       !  !     !  !             !   !! !  !   !!
```

```
RATPALTA  139: ctgctgtcgatgtggccgtgaaagtgttcaaaaggactgcagacggaagctg


MMALBR    208: ggagccctttgcctctgggaagaccgcggagtctggagagctgcacgggctc
                     !                    !
RATPALTA  191: ggagccgtttgcctctgggaagaccgccgagtctggagagctgcacgggctc


MMALBR    260: accacagatgagaagtttgtagaaggagtgtacagagtagaactggacacca
                                !!!!     !        !
RATPALTA  243: accacagatgagaagttcacggaaggggtgtacagggtagaactggacacca


MMALBR    312: aatcgtactggaagacacttggcatttccccgttccatgaattcgcggatgt
                   !         ! !              !          !   !  !
RATPALTA  295: aatcatactggaaggctcttggcatttccccattccatgaatacgcagaggt


MMALBR    364: ggttttcacagccaacgactctggccatcgccactacaccatcgcagccctg
                              !        !
RATPALTA  347: ggttttcacagccaatgactctggtcatcgccactacaccatcgcagccctg


MMALBR    416: ctcagcccatactcctacagcaccacggctgtcgtcagcaacccccagaatt
                       !                 !           !           !
RATPALTA  399: ctcagcccgtactcctacagcaccactgctgtcgtcagtaacccccagaact


MMALBR    468: gagagactcagcccaggaggaccaggatcttgccaaagcagtagcatcccat
                  !   !       !        !                    !
RATPALTA  451: gagggacccagcccacgaggaccaagatcttgccaaagcagtagc-tcccat


MMALBR    520: ttgtaccaaaacagtgttcttgctctataaaccgtgttagcagctcaggaag
                     !!                                  !   !
RATPALTA  502: ttgtactgaaacagtgttcttgctctataaaccgtgttagcaactcgggaag


MMALBR    572: atgccgtgaagcattcttattaaaccacctgctatttcattc
                         ! !                 !!
RATPALTA  554: atgccgtgaaacgttcttattaaaccacctt-tatttcattc
```

```
HUMPALA     0: acagaagtccactcattcttggcaggatggcttctcatcgtctgctcctcct
               ! !  !!!!!!!!!!!!!!!!!            ! !   !   !
RATPALTA    0: cctga-----------------caggatggcttcccttcgcctgttcctcct

HUMPALA    52: ctgccttgctggactggtatttgtgtctgaggctggccctacgggcaccggt
                     !         !      !      !         !!   !! !  !
RATPALTA   35: ctgcctcgctggactgatatttgcgtctgaagctggccctgggggtgctgga

HUMPALA   104: gaatccaagtgtcctctgatggtcaaagttctagatgctgtccgaggcagtc
                                            !  !                 !
RATPALTA   87: gaatccaagtgtcctctgatggtcaaagtcctggatgctgtccgaggcagcc

HUMPALA   156: ctgccatcaatgtggccgtgcatgtgttcagaaaggctgctgatgacacctg
                   !!  !           ! !       !  ! !    !  ! !! !
RATPALTA  139: ctgctgtcgatgtggccgtgaaagtgttcaaaaggactgcagacggaagctg

HUMPALA   208: ggagccatttgcctctgggaaaaccagtgagtctggagagctgcatgggctc
                     !              !   !!!                 !
RATPALTA  191: ggagccgtttgcctctgggaagaccgccgagtctggagagctgcacgggctc

HUMPALA   260: acaactgaggaggaatttgtagaagggatatacaaagtggaaatagacacca
                 !  !  !   ! !  !!!!      ! !    !!  !   ! !
RATPALTA  243: accacagatgagaagttcacggaaggggtgtacagggtagaactggacacca

HUMPALA   312: aatcttactggaaggcacttggcatctccccattccatgagcatgcagaggt
                   !           !        !              !! !
RATPALTA  295: aatcatactggaaggctcttggcatttccccattccatgaatacgcagaggt

HUMPALA   364: ggtattcacagccaacgactccggcccccgccgctacaccattgccgccctg
                  !           !     !  ! !!    !         !  !
RATPALTA  347: ggttttcacagccaatgactctggtcatcgccactacaccatcgcagccctg

HUMPALA   416: ctgagcccctactcctattccaccacggctgtcgtcaccaatcccaaggaat
                 !     !        !!!      !          !!  !   !  ! !
RATPALTA  399: ctcagcccgtactcctacagcaccactgctgtcgtcagtaacccccagaact

HUMPALA   468: gagggacttctcctccagtggacctgaaggacgagggatgggatttcatgta
                      !!!!  !!!!!!!!!!!!!!!!!      !!!!!   ! !!!!!
RATPALTA  451: gagggacccagccc----------------acgaggaccaagatcttgccaa

HUMPALA   520: accaagagtattccatttttactaaagcactgttttcacctcatatgctatg
                !  !!  !! !      !    !  !  !    ! !! !!!   !! !!
RATPALTA  487: agcagtagc-tcccatttgtactgaaacagtgttcttgctctataaaccgtg
```

```
HUMPALA   572: ttagaagtccaggcagagacaataaaacattcctgtgaaagg----------
                   ! !!! !  !   !! !! !    !   ! ! !   !!!!!!!!!!!!
RATPALTA  538: ttagcaactcgggaagatgccgtgaaacgttcttattaaaccacctttattt


HUMPALA   614: ----c
               !!!!
RATPALTA  590: cattc



HUMPALA     0: acagaagtccactcattcttggcaggatggcttctcatcgtctgctcctcct
                  ! !!     !!!!  !  !            ! !   !  !!
MMALBR      0: acacagatccacaagctcctgacaggatggcttcccttcgactcttcctcct


HUMPALA    52: ctgccttgctggactggtatttgtgtctgaggctggccctacgggcaccggt
               !     !                       !        !!    !! !  !
MMALBR     52: ttgcctcgctggactggtatttgtgtctgaagctggccccgcgggtgctgga


HUMPALA   104: gaatccaagtgtcctctgatggtcaaagttctagatgctgtccgaggcagtc
                       !                    !  !                 !
MMALBR    104: gaatccaaatgtcctctgatggtcaaagtcctggatgctgtccgaggcagcc


HUMPALA   156: ctgccatcaatgtggccgtgcatgtgttcagaaaggctgctgatgacacctg
                   !! !! !     !  !! !       !    ! !!    ! !!!
MMALBR    156: ctgctgtagacgtggctgtaaaagtgttcaaaaagacctctgagggatcctg


HUMPALA   208: ggagccatttgcctctgggaaaaccagtgagtctggagagctgcatgggctc
                     !              !   !!!                 !
MMALBR    208: ggagccctttgcctctgggaagaccgcggagtctggagagctgcacgggctc


HUMPALA   260: acaactgaggaggaatttgtagaagggatatacaaagtggaaatagacacca
                 !  !  !   ! !           !! !    !   !   ! !
MMALBR    260: accacagatgagaagtttgtagaaggagtgtacagagtagaactggacacca


HUMPALA   312: aatcttactggaaggcacttggcatctccccattccatgagcatgcagaggt
                   !         !          !     !        !!!!  !  !
MMALBR    312: aatcgtactggaagacacttggcatttccccgttccatgaattcgcggatgt


HUMPALA   364: ggtattcacagccaacgactccggcccccgccgctacaccattgccgccctg
                  !                 !    !!    !         !  !
MMALBR    364: ggttttcacagccaacgactctggccatcgccactacaccatcgcagccctg


HUMPALA   416: ctgagcccctactcctattccaccacggctgtcgtcaccaatcccaaggaat
                 !     !        !!!                 !   !   !  ! !
MMALBR    416: ctcagcccatactcctacagcaccacggctgtcgtcagcaacccccagaatt
```

```
HUMPALA   468: gagggacttctcctccagtggacctgaaggacgagggatgggatttcatgta
                  !    !!!  !!!!!!!!!! ! !     !   !!!!!!  ! !!!!!
MMALBR    468: gagagactcagcc----------caggaggaccagg------atcttgccaa


HUMPALA   520: accaagagtattccatttttactaaagcactgttttcacctcatatgctatg
                !  !!  !  !      !   !   !  !    ! !! !!!   !! !!
MMALBR    504: agcagtagcatcccatttgtaccaaaacagtgttcttgctctataaaccgtg


HUMPALA   572: ttagaagtccaggcagagacaataaaacattcctgtgaaagg----------
                   !  !!    !   !! !! !  !     ! ! !   !!!!!!!!!!!!
MMALBR    556: ttagcagctcaggaagatgccgtgaagcattcttattaaaccacctgctatt


HUMPALA   614: -----c
               !!!!!
MMALBR    608: tcattc
```