I

Sequence Alignments

1

Pairwise Sequence Alignment

Benjamin N. Jackson
Iowa State University

Srinivas Aluru
Iowa State University

1.1 Introduction

The discovery of biomolecular sequences and exploring their roles, interplay, and common evolutionary history is fundamental to the study of molecular biology. Three types of sequences fill complementary roles in the cell: DNA sequences, RNA sequences, and protein sequences. DNA sequences are the basis of genetic material and act as the hereditary mechanism, providing the recipe for life. RNA sequences are derived from DNA sequences and play many roles in protein synthesis. Protein sequences carry out most essential processes such as tissue building, catalysis, oxygen transport, signaling, antibody defense, and transcription regulation. The first part of this book will describe the alignment algorithms used to compare these sequences.

For the benefit of the reader unfamiliar with molecular biology, we provide a more detailed introduction to biological sequences. A DNA molecule is composed of simpler molecules known as nucleotides. The nucleotides are differentiated by the differences in their bases — Adenine, Cytosine, Guanine and Thymine, represented by A, C, G, and T, respectively. DNA naturally occurs as a double-stranded helix-shaped molecule, with each nucleotide in one strand pairing with a corresponding nucleotide in the other strand, with A pairing with T and G pairing with C and vice versa. Each strand has a direction, with the two strands having opposite directions. The DNA molecule is represented by the sequence of nucleotides of one strand in that strand's direction. Given one strand, the sequence of the other strand is obtained by reversing the known strand and substituting A for T, C for G, etc. This process is called generating the *reverse complement* and is important when comparing DNA sequences as either strand might be given for the DNA being compared.

Several different terms are used to describe DNA sequences. Each cell in an organism contains the same set of *chromosomes*, which are long DNA sequences. The set of chromosomes in an organism constitutes its *genome.* A *gene* is a contiguous stretch of DNA along a chromosome that codes for a protein or RNA. Genes consist of one or more coding regions called *exons* separated by non-coding regions called *introns.* The terms *promoter, enhancer,* and *silencer* are used to describe DNA sequences involved in regulating gene expression through protein interactions and are often located upstream of the gene. Genes and regulatory regions are often conserved (show high similarity or homology) across species.

An important function of DNA sequences is to code for protein sequences. Like DNA sequences, proteins are also sequences of simpler molecules, in this case amino acids. Amino acids are differentiated by their side chains. There are twenty possible side chains that distinguish the twenty different amino acids found in protein sequences. As with DNA, each of the twenty amino acids is represented by a unique character.

A protein is derived from a gene through an RNA intermediary. Similar to DNA, RNA is a sequence of nucleotides with the base Thymine replaced by Uracil. First, an RNA called *pre-mRNA* containing both exons and introns is copied from the DNA in a process called *transcription.* The introns are excised and the exons are spliced to form an mRNA. The mRNA is then *translated* into an amino acid sequence. A *codon* is three consecutive nucleotides in the mRNA that is translated to an amino acid in the corresponding protein. The mRNA is used as a template to generate an amino acid sequence of one third the length of the coding region. The code mapping the 64 possible codons to the 20 possible amino acids is common to almost all of life. The two step process of transcribing DNA to RNA and translating RNA to protein is popularly known as the central dogma of molecular biology.

Multiple forms of the same gene, known as *alleles*, cause genetic differences between individuals and are responsible for the genetic diversity of a species. Sometimes, variations in alleles lead to undesirable outcomes such as genetic diseases or increased susceptibility to diseases. The differences between alleles are often quite small. Sometimes a single nucleotide change can have a large effect on the resulting protein. DNA sequences are typically modified through insertions, deletions or substitutions. These underlying evolutionary mechanisms provide a starting point for sequence alignment algorithms.

Sequence alignments are intended to discover and illustrate the similarities, differences, or evolutionary relationships between sequences. The algorithms used for sequence comparison vary depending on the types of sequences being compared and the question being asked, giving rise to a variety of sequence alignment algorithms. In this chapter, we will present the basic sequence alignment algorithms, broadly characterized as *global alignment, semiglobal alignment,* and *local alignment.* Global alignment can be used to compare two protein sequences from a closely related gene family, two homologous genes, or two gene alleles. Semiglobal alignment can be used to piece together fragments of DNA from shotgun DNA reads and create a longer inferred sequence, useful in genome assembly. Local alignment can be used as a part of *multiple local alignment*, presented in Chapter 3, to find a common motif among protein sequences or conserved promoter sites in gene sequences. Chapter 2 presents spliced alignments, which are important when aligning DNA with RNA transcripts. Finally, Chapter 4 addresses the characteristics of the problem space, and how changing parameters affect alignment results.

1.2 Global Alignment

The global sequence alignment problem for two sequences is defined as follows. We call the set of unique characters in the input sequences an alphabet Σ. In the case of DNA sequences, that alphabet is $\Sigma = \{a, g, c, t\}$. A string X of length n is a sequence of characters $\langle x_1, x_2, ..., x_n \rangle$ such that $x_i \in \Sigma$. A *prefix* of X is a string of the form $\langle x_1, x_2, ..., x_i \rangle, 1 \leq i \leq n$. A *substring* of X is a string of the form $\langle x_i, x_{i+1}, ..., x_{j-1}, x_j \rangle, 1 \leq i \leq j \leq n$. For example 'aggctga' is a string with substrings 'aggc' and 'gctg', with 'aggc' also being a prefix of 'aggctga'.

A string of characters is the term traditionally used in computer science literature, and it is equivalent to the concept of a sequence in biology. We will use the term string almost exclusively in this chapter. However, it is important to adapt the string algorithms to the specific biological sequences of interest. For example, when comparing DNA sequences, it is important to compare the two input sequences, as well as the reverse complement of one sequence with the other input sequence.

Consider two strings $A = \langle a_1, a_2, ..., a_n \rangle$ and $B = \langle b_1, b_2, ..., b_m \rangle$. Conceptually we wish to create an alignment between the two strings, matching similar regions by aligning each character in string A with a character in string B. Additionally, we can insert gaps in each string (allowing for the possibility of deletions or insertions of sequences of characters). More formally, an alignment between A and B is the production of two new strings of equal length, A_L derived from A and B_L derived from B through insertions of a special gap character '-'. $A_L = \langle a_1, a_2, ..., a_l \rangle$ and $B_L = \langle b_1, b_2, ..., b_l \rangle$, where l is the alignment length, $\max(n, m) \leq l \leq n + m$. Both a_i and b_i may not be gap characters. a_i and b_i are said to be aligned with each other. If a_i is a gap, then b_i is said to be aligned with a gap in A, and vice versa. An example alignment between two strings 'aggctga' and 'agcttg' is shown below.

```
aggct-ga
ag-cttg-
```

The quality of the alignment is measured by its score, which can be thought of as a measure of how similar the two strings are. The score is the summation of the score of each pair of characters a_i and b_i. We will choose a simple scoring function that has roots in our evolutionary model. A character aligned with the same character, a *match*, is given a score α. This corresponds to a conserved character. A character aligned with any other character, a *mismatch*, is given a score β, and corresponds to a substitution. Finally, a character aligned with a gap, a *gap*, is given a score γ, and corresponds to either an insertion or a deletion in one of the strings.

$$score(L) = \sum_{i=1}^{l} score(a_i, b_i)$$

$$score(x, y) = \begin{cases} \alpha & x = y \\ \beta & x \neq y \\ \gamma & x = `-' \text{ or } y = `-' \end{cases}$$

Typically α is positive and γ and β are negative. We will consider the values $\alpha = 2, \beta = -1, \gamma = -1$. Given these values, the example alignment has a total score of 7.

We wish to find an alignment between the two strings that results in the highest score, called an *optimal alignment* between the two strings. A simplistic solution would be to score all possible alignments and chose (from) the highest scoring, but the number of such possibilities is exponential.

1.3 Dynamic Programming Solution

The solution can be sped up using dynamic programming. We see that the problem exhibits an optimal substructure. Consider an optimal alignment L between A and B. If we look at some part of that optimal alignment L' that aligns a substring A' of A with a substring B' of B, we wish to say, for optimal substructure, that L' is an optimal alignment between A' and B'. The proof is simple, using contradiction. If the alignment L' is not optimal, then there exists an alignment L_{new} between A' and B', with $score(L_{new}) > score(L')$. However, L_{new} can be substituted for L' in L, increasing the score of L. Therefore L is not optimal, a contradiction.

We can use the optimal substructure property to solve the problem more efficiently using the following formulation. In order to find the optimal alignment between the two strings, we find the optimal alignment between each prefix A_i of A and each prefix B_j of B, where A_i is the prefix of length i of A and B_j is the prefix of length j of B. Let a_i be the last character in A_i and b_j be the last character in B_j. There are three possibilities that can produce the optimal score.

1. Align a_i with b_j and optimally align A_{i-1} with B_{j-1}.
2. Align a_i with a gap and optimally align A_{i-1} with B_j.
3. Align b_j with a gap and optimally align A_i with B_{j-1}.

We will denote the optimal score of aligning A_i with B_j as $S[i,j]$. Think of a table that records the maximum score of aligning all possible pairs of A_i and B_i. The following recurrence describes how to fill the table using the ideas presented above.

$$S[i,j] = \max \begin{cases} S[i-1,j-1] + \delta(a_i, b_j) \\ S[i,j-1] + \gamma \\ S[i-1,j] + \gamma \end{cases}$$

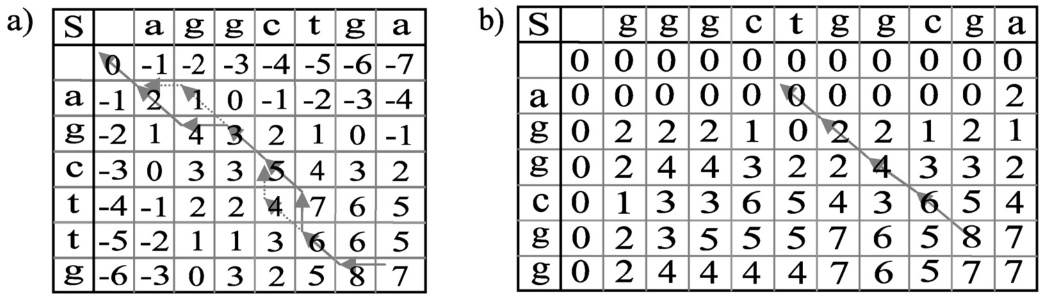
$$\delta(x,y) = \begin{cases} \alpha & x = y \\ \beta & x \neq y \end{cases}$$

All that remains is to specify the starting conditions. The score of aligning some prefix of A with none of B is the length of that prefix times the gap penalty. Formally, $S[0,j] = \gamma j$ and $S[i,0] = \gamma i$.

A sample table is shown in Figure 1.1. Rows in S correspond to characters in A with the first row corresponding to the empty string. Columns in S correspond to characters in B with the first column corresponding to the empty string. We can initialize the first row and first column of the table as described in the previous paragraph.

Notice that to fill a cell of the table using the recursive definition above, we need to know the value of three other cells — the cell to the north, the cell to the west, and the cell to the northwest. Therefore, if we start to fill the table row by row, from top to bottom and left to right, we will have already filled in these three cells before reaching the current cell.

The amount of time to fill in each cell is constant, so the total time to fill out the table is equal to the number of cells, or $O(nm)$. The space requirement is the same. When the algorithm is finished, the best alignment score is recorded in $S[n,m]$.


a)

S		a	g	g	c	t	g	a
	0	-1	-2	-3	-4	-5	-6	-7
a	-1	2	1	0	-1	-2	-3	-4
g	-2	1	4	3	2	1	0	-1
c	-3	0	3	3	5	4	3	2
t	-4	-1	2	2	4	7	6	5
t	-5	-2	1	1	3	6	6	5
g	-6	-3	0	3	2	5	8	7

b)

S		g	g	g	c	t	g	g	c	g	a
	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0	0	2
g	0	2	2	2	1	0	2	2	1	2	1
g	0	2	4	4	3	2	2	4	3	3	2
c	0	1	3	3	6	5	4	3	6	5	4
g	0	2	3	5	5	5	7	6	5	8	7
g	0	2	4	4	4	4	7	6	5	7	7

FIGURE 1.1: a) The score table for strings "agcttg" and "aggctga" with $\alpha = 2, \beta = -1, \gamma = -1$. The optimal alignment is found by starting at the southeast cell in the table and tracing the path back to the northwest cell. For these strings, more than one alignment produces the score, resulting in more than one possible path. (b) The score table for strings "aggcgg" and "gggctggcga" showing a local alignment. The alignment path through the table is shown with arrows. The optimal alignment is found by searching the table for the maximum value and then tracing a path until reaching a cell with score 0.

The table shown in Figure 1.1 aligns our two sample strings using the parameters $\alpha = 2, \beta = -1, \gamma = -1$. As indicated in the southeast corner cell, the best alignment has a score of 7.

We also wish to construct an alignment corresponding to this score, as it provides information about how the two strings are similar and different, or equivalently it illustrates the homology between the two sequences. We can think of the score in each cell as having a corresponding move, indicating which neighboring cell — north, northwest, or west — was used in producing that cell's score. If we trace these moves from $S[n, m]$ to $S[0, 0]$, called *traceback*, we can construct the alignment. Let's consider cell $S[i, j]$.

1. A diagonal move to $S[i, j]$ corresponds to aligning a_i and b_j.
2. A horizontal move to $S[i, j]$ corresponds to inserting a gap in A after a_i.
3. A vertical move to $S[i, j]$ corresponds to inserting a gap in B after b_j.

One possible way to complete the traceback is to store the moves made for each cell in addition to the score. However, this is unnecessary as the possible moves can be deduced from the score table by considering three cases for each cell.

1. If $S[i, j] - \delta(a_i, b_j) = S[i - 1, j - 1]$, a diagonal move could have been used to reach $S[i, j]$.
2. If $S[i, j] - \gamma = S[i, j - 1]$, a horizontal move could have been used to reach $S[i, j]$.
3. If $S[i, j] - \gamma = S[i - 1, j]$, a vertical move could have been used to reach $S[i, j]$.

Multiple move possibilities imply that there are multiple alignments that produce the optimal score. Figure 1.1 shows that the example strings have more than one alignment that produce a score of 7.

1.4 Semiglobal and Local Alignment

Our dynamic programming solution to the sequence alignment problem resulted in aligning *all* of A with *all* of B. This is called a global alignment, and was first applied in computational biology by Needleman and Wunsch [34]. However, in some cases, a global alignment is not that interesting. Consider the two strings 'agctgctatgataccgacgat' and 'atcata'. An optimal global alignment matches each character perfectly:

```
agctgctatgataccgacgat
a--t-c-at-a----------
```

But a more interesting alignment produces a mismatch and therefore a lower global score, but is much more biologically meaningful. Variations on the global alignment algorithm address this intuition, and were first presented in the context of biological sequences by Smith and Waterman [39].

```
agctgctatgataccgacgat
-------atcata--------
```

1.4.1 Semiglobal Alignment

The first variation is called a *semiglobal*, or *end gaps free* alignment. In this type of alignment, all gaps inserted before or after the string do not affect the score of the alignment. In other words, we are allowed to ignore a prefix of A or a prefix of B but not both. We are also allowed to ignore a suffix of A or a suffix of B but not both. This type of alignment might be interesting if we were assembling a genome from shotgun reads. We would expect high similarity between overlapping ends of two reads, but would not want to incur a penalty for ignoring the non-overlapping ends. Genome assembly is covered more thoroughly in Part III.

In the following discussion, we will use the term *exhausted* to describe the way in which the algorithm uses up characters from A and B. After calculating $S[i, j]$, the algorithm is said to have exhausted the first i characters from A and the first j characters from B.

The semiglobal alignment is achieved through two small modifications to the global alignment algorithm. The first modification addresses inserting gaps at the beginning of a string, or ignoring either a prefix of A or a prefix of B. In a global alignment, we started with an alignment score of 0 only when we had exhausted no characters from both A and B. This corresponded to initializing $S[0, 0]$ to 0. Now we wish to be able to start with a score of 0 after ignoring either a prefix of A or a prefix of B. This condition holds as long as we have not exhausted any characters from either A or B, which corresponds to the first row or first column of the table. Therefore we can achieve the result by initializing the first row and column of the table to 0.

The second modification addresses inserting gaps at the end of a string, or ignoring either a suffix of A or a suffix of B. Because we can only ignore a suffix of either A or B, the alignment must exhaust all the characters of either A or B. In terms of the table, this is the case in the last row or column. In a global alignment, we were required to exhaust the characters from both A and B, so the score appeared in the southeast corner of the table. In semiglobal alignment, the score is the maximum over the last row and column of the table.

The traceback of the alignment path through the table proceeds as in global alignment, however the traceback starts at the found maximum and ends at any cell in the first row or column.

1.4.2 Local Alignment

The second variation to global alignment allows even more flexibility than semiglobal alignment. In a *local* alignment, we wish to choose some substring A' of A and some substring B' of B such that A' aligned with B' produces the maximum score. In other words, while for semiglobal alignment we could ignore a prefix of either A or B and a suffix of either A or B, for local alignment we can ignore a prefix and suffix of both A and B.

Possible uses of local alignment include identifying a conserved exon in two genomic sequences and identification of a conserved regulatory region upstream of two genes. We are highly interested in the similar region shared between the two sequences, but are indifferent to remainder of the sequences. In this case, a local alignment would allow us to ignore the parts of the sequence that do not align well, while focusing on the region with the best local similarity.

We create a local alignment by extending the ideas used in semiglobal alignment. Instead of only starting our score at zero in the first row or column (allowing A or B to ignore prefix), we now have the possibility of starting our alignment score at zero in any cell in the table, allowing both A and B to ignore a prefix. This is done by modifying the equation presented in section 1.3.

$$S[i,j] = \max \begin{cases} S[i-1,j-1] + \delta(a_i, b_j) \\ S[i,j-1] + \gamma \\ S[i-1,j] + \gamma \\ 0 \end{cases}$$

We do not allow any cell in the table to take on a negative value. Setting the score of $S[i,j]$ to zero when it would have been negative corresponds to ignoring the prefixes A_i and B_j.

We can deal with ignoring suffixes as an extension of semiglobal alignment as well. Instead of looking for the maximum value over the last row and column, which restricts us to ignoring a suffix of either A or B, we search for the maximum value over the entire table, equivalent to ignoring a suffix of both A and B.

To do a traceback of the local alignment, start at the cell containing the maximum value and traceback until reaching a cell with value 0. An example local alignment with traceback is shown in Figure 1.1. The differences between global, semiglobal, and local alignments are summarized in Table 1.1.

	Global	Semiglobal	Local
Ignore Suffix	no	A or B	A and B
Ignore Prefix	no	A or B	A and B
Reset to Zero	$S[0,0]$	$S[i,0]$, $S[0,j]$	$S[i,j]$
Maximum In	$S[n,m]$	$S[i,m]$, $S[n,j]$	$S[i,j]$

TABLE 1.1 Differences between global, semiglobal, and local alignments

1.5 Space Saving Techniques

We have finished introducing the concept of string alignment. With the basics covered, the rest of the chapter will cover two classes of modifications on this initial concept. The first class are algorithmic improvements, modifications that improve the runtime or space complexity of the algorithms. Space saving techniques, the k-band formulation, and subquadractic alignments fall into this class. We will also cover qualitative modifications such as substitution matrices, normalized alignment, and different gap penalty functions. These modifications often introduce complexities that result in the algorithms taking more time or space, but have the benefit of producing more biologically valid results.

The first modification we discuss is quite important. As mentioned above, the dynamic programming solution described takes $O(nm)$ time and $O(nm)$ space, where n and m are the sizes of the strings. While the $O(nm)$ runtime is quite fast on any reasonably modern computer for string sizes up to a few hundred thousand characters, the space will become a factor before that. Two strings of size 20,000 will require around 1.6 GB of RAM for the dynamic programming table if each cell is a 4 byte integer. For this reason we are interested in reducing the space required to run the algorithm.

We will discuss a technique introduced by Hirschberg [21] that reduces the space requirement from $O(nm)$ to $O(n + m)$ while maintaining the runtime. Obviously, achieving a reduction in space required to hold the table to 160 KB from 1.6 GB is a useful improvement. The space requirement is the theoretical minimum because at the very least the input and output require $O(n+m)$ space to hold the strings themselves. We will first discuss the technique within the context of global alignment.

1.5.1 Preliminaries

If we are looking for the alignment score without producing the alignment, it is easy to reduce the space requirement to $O(n+m)$. Consider filling the table row by row, from top to bottom. To fill any row, we need access to values from the previous and current rows only. It is easy to envision an algorithm which uses two arrays of size m, corresponding to the previous and current rows. When the current row is complete, the two arrays swap roles and the algorithm continues.

If we store only two rows of the table, we cannot proceed with path traceback, because we have lost most of the information needed for this step. While the entire path cannot be found, there is a way to discover a small part of the path; there is enough information in the last two rows to construct the last bit of the path. This observation will serve as the basis for the first naive space optimal algorithm.

As shown in Figure 1.2, the path of the alignment can be described as a list of n intervals, one for each row of the table. Each interval can be defined by its endpoints. Notice that the endpoints of each subsequent interval either overlap or touch through the diagonal. This is because alignment path must move on either a diagonal or a vertical path from row i to row $i-1$. We will call the left endpoint of row i and the right endpoint of row $i-1$ the path intersection for row i. We can define the alignment path as $n-1$ intersections, if we consider that the northwest corner and southwest corner of the matrix will always be endpoints in the interval list.

Each intersection can be discovered using data from only two rows. To construct the entire path, we will build the interval list intersection by intersection. A naive approach finds one intersection per iteration, starting at the bottom of the table and working up to the top. For each iteration, the algorithm calculates the last two rows of the submatrix for A_i and B_j. It uses this information to discover the intersection for row i. This algorithm uses $O(n+m)$ space but runs in $O(n^2m)$ time.

1.5.2 Using Hirschberg's Recursion

The naive space saving algorithm has a harsh runtime penalty. Hirschberg introduced a divide and conquer approach that was first applied to the biological sequence alignment

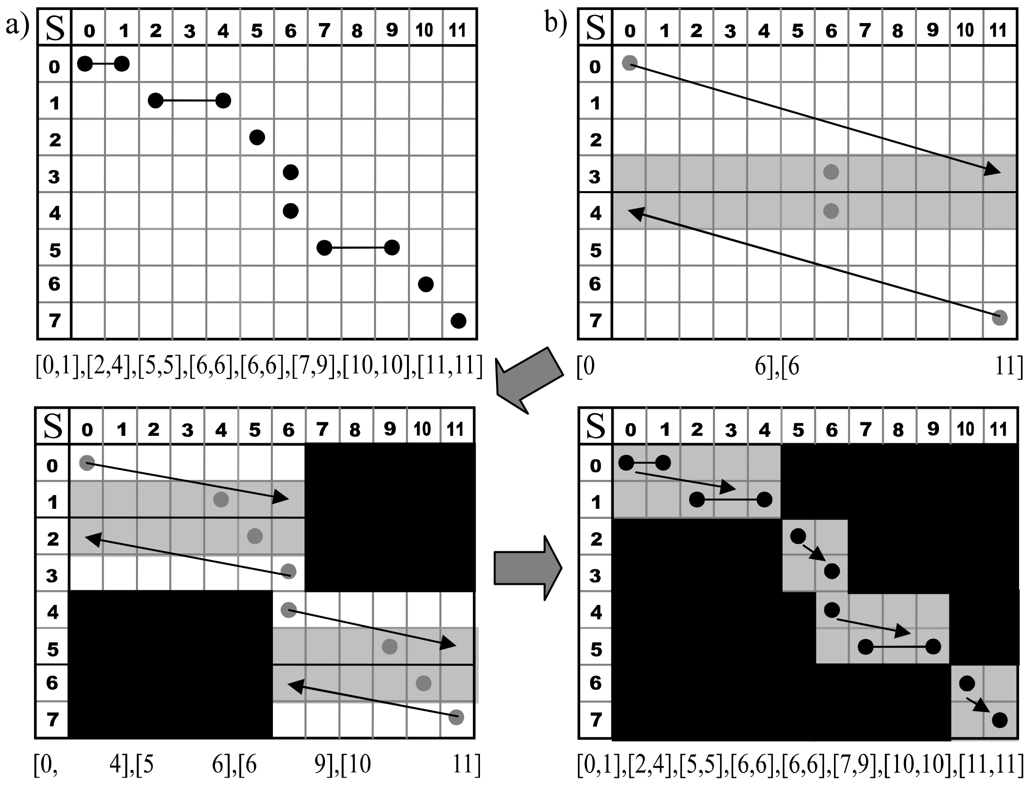


FIGURE 1.2: (a) The alignment path through the table S can be represented as a list of intervals, one per row. (b) Hirschberg's recursion allows us to construct the list in $O(nm)$ time using only $O(n+m)$ space. The black arrows represent the direction the DP algorithm is run on each sub matrix. The gray boxes represent the cells in memory at the end of each step. Partially known intervals are deduced from these cells, as shown by gray dots. In the last stage of the recursion, alignment is run forward in all sub matrices to complete the alignment path.

problem by Myers and Miller [33]. Instead of finding the intersections from bottom to top, we find the intersection in the center of the table first. This will allow us to eliminate more of the table for the next iteration. Refer to Figure 1.2 during the discussion.

To find the intersection for the center row, divide the table in half, with the centerline of the table $t = \lfloor \frac{n}{2} \rfloor + 1$. The top half of the table is $S[0...t-1, 0...m]$ and the bottom half of the table is $S[t...n, 0...m]$. On the top half of the table, run the space-saving algorithm to find the scores along the bottom row of the top half of the table, row $t-1$.

On the bottom half of the table, run the algorithm *backwards*. In other words, initialize the bottom row and right column and run the algorithm from right to left and bottom to top. This is the same as reversing the strings and running the algorithm forwards. The table cell formula is defined as:

$$S[i,j] = \max \begin{cases} S[i+1, j+1] + \delta(a_i, b_j) \\ S[i, j+1] + \gamma \\ S[i+1, j] + \gamma \end{cases}$$

When we finish running the algorithm backwards in the bottom half of the table, we will have scores for the top row of the bottom half of the table, which has index t. Now we will find the move between rows $t-1$ and t (the intersection for t) that produces the maximum score.

More formally, the intersection for t is defined by indexes i in $t-1$ and j in t, $i \leq j \leq i+1$, that maximize the function:

$$\max_i \begin{cases} S[t-1, i] + S[t, i] + \gamma & \text{j=i} \\ S[t-1, i] + S[t, i+1] + \delta(a_t, b_{i+1}) & \text{j=i+1} \end{cases}$$

Next divide the table into four quadrants. The northeast and southwest quadrants can be ignored, as the optimal path does not travel through them. We must recursively run the algorithm on the northwest and southeast quadrants, defined as $S[0...t-1, 0...i]$ and $S[t...n, j...m]$. The recursion will continue until the number of rows is 1 or 2, at which point the DP algorithm will be run forwards and the optimal path completed.

The problem approximately halves each iteration, because if one splits a rectangle into four quadrants and selects two, the area of the two selected quadrants is half of the original rectangle. The details are left out of this discussion but are easily solved. Because the sum $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$, the total runtime remains $O(mn)$. The algorithm remembers only $n+1$ cells for each half of the table during each iteration, and as a result the space requirement has been reduced to $O(m+n)$.

The ideas in this algorithm can be extended to both semiglobal and local alignments, as shown by Huang *et al.* [22]. We will consider the case of local alignments, as semiglobal alignments can be handled similarly. Consider the case in which there is exactly one maximum scoring path through the table. The idea can be extended to work with multiple such paths, which we will not consider here.

To solve the problem, we will first find each endpoint of an optimal alignment path, and then run a global alignment on the induced subtable. First run the algorithm forward using the local alignment recursion. As it proceeds, keep track the cell that contains the maximum score. This is the southeast corner of the subtable. Next run the algorithm backwards while keeping track of the cell that gives the maximum score from this direction (the maximum value will be the same). This cell is the northwest corner of the subtable. Run the linear space global alignment algorithm on the subtable defined by these two cells to produce the optimal local alignment.

1.6 Banded Alignment

Imagine that we are studying two orthologous DNA sequences, that is two DNA sequences that are thought to have evolved from the same ancestral sequence. Our two genes code for the same function and the species are evolutionarily close. Therefore, the two sequences are highly similar and are of similar length. Because of this the alignment path between the two sequences will remain close to the main diagonal. A banded alignment makes use of this observation to achieve faster runtime. The idea of a banded alignment was first proposed by Fickett [16].

Consider the subset of inputs in which A and B are highly similar and of the same length n. The k-band algorithms runs in time proportional to the difference between A and B. If A and B are similar enough, then the algorithm's runtime is $O(nk)$ for some small constant k. In the worst case, the runtime is still $O(n^2)$ with an additional constant multiplier of approximately 2.

The k-band algorithm ignores the part of the array distant from the main diagonal during its calculation. Let the value k denote a region of the table called the *k-band*, such that the following is true for the table.

$$S[i,j] \in k\text{-}band \Leftrightarrow |i-j| \leq k$$

For the purpose of the algorithm, all cells outside of the k-band are considered to have a score of $-\infty$, which will cause them to be ignored in the maximum calculation. The algorithm runs as normal, but will only consider those cells within the k-band. Figure 1.3 shows the k-band initialization for $k = 3$. In practice the cells marked $-\infty$ are not actually initialized and do not exist in memory, for this would defeat the purpose of the algorithm. They are shown for clarity.

The end of the algorithm's run will result in an alignment with some score s_k that represents the best alignment under the restriction that the alignment path does not travel

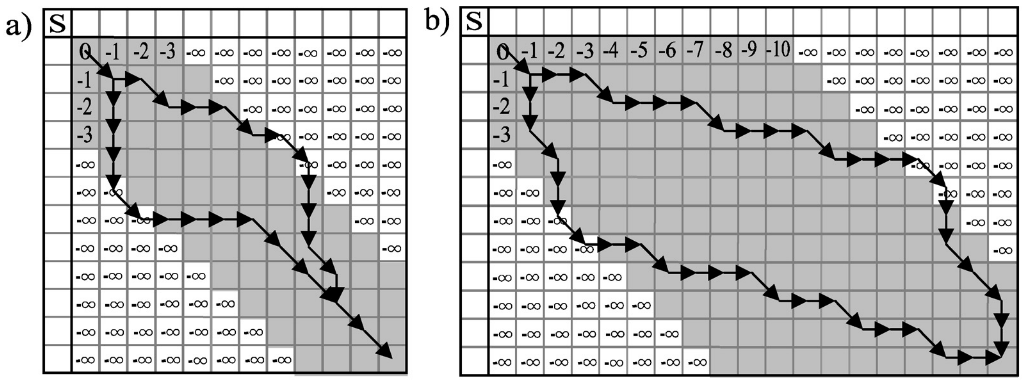


FIGURE 1.3: (a) The k-band initialization for $k = 3$. The k-band is shown in gray. The cells that conceptually take on $-\infty$ do not actually exist in memory. Two hypothetical best paths that travel outside of the k-band are shown. To achieve the best score, all diagonals must represent matches. Hence, the score of these two paths is $2(k+1)\gamma + (n-(k+1))\alpha$. (b) The k-band initialization for global alignment when $k = 3$ and $m > n$. The cost of the hypothetical best path traveling outside of the k-band is $(2(k+1)+m-n)\,\gamma + (n-(k+1))\alpha$.

outside of the k-band. To show that this alignment is optimal, we consider possible alignments that have paths that travel outside of the k-band. In other words, for some cell on the alignment path, we have $|i - j| > k$. The highest scoring such alignment would have exactly $k + 1$ characters from A aligned with gaps, $k + 1$ characters from B aligned with gaps, and all other $n - (k + 1)$ characters as matches. The score for this alignment would be

$$best_{k+1} = 2(k + 1)\gamma + (n - (k + 1))\alpha$$

If $s_k >= best_{k+1}$, then the alignment s_k is known to be an optimal alignment, because it beats the best score of any possible alignment that travels outside of the k-band region.

However, if $s_k < best$, then we cannot be sure that s_k is optimal. To solve this problem and maintain our worst case runtime, we double k and rerun the algorithm. In the worst case we keep doubling k until $k \geq n$, at which point all elements in the table are in the k-band. The runtime in the worst case is the sum of all iterations, which is still $O(n^2)$, because the number of elements considered increases exponentially until the number of entries considered constitutes the entire table.

While repeatedly doubling k produces an algorithm that remains asymptotically optimal, it may increase the runtime unnecessarily for some applications. In practice we may wish to only find an alignment if the similarity between two strings is high. If the score of the best alignment is below some minimum threshold, T, then the strings are considered dissimilar and we are no longer interested in finding the alignment. If this is the case, we can choose k such that $best_{k+1} \leq T$ and never have to run the k-band algorithm more than one iteration; if $s_k < best_{k+1}$, then $s_k < T$ and we can report no good alignment. Solving k in terms of T, we have

$$k \geq \frac{T - (n - 1)\alpha}{2\gamma - \alpha}$$

The k-band as described for two strings of exactly the same length is very limiting, so we will briefly look at the implications of $|A| = m \neq |B| = n$. For ease of discussion, assume that $m > n$. Now the k-band is redefined.

$$S[i, j] \in k\text{-}band \Leftrightarrow n - m - k \leq i - j \leq k$$

This can be thought of as inserting a parallelogram of width $m - n$ at the center of the k-band defined for $n = m$, as shown in Figure 1.3.

In addition the score $best_{k+1}$ — used as the termination decision and in calculating the k based on the threshold parameter T — is calculated using a more general form of the equation presented for $n = m$.

$$best_{k+1} = (2(k + 1) + m - n)\gamma + (n - (k + 1))\alpha$$

1.7 Other Gap Penalty Functions

In the algorithms presented thus far, the penalty for aligning a character with a gap has been γ. However, in most biological applications, it does not make sense to penalize gaps in this manner. For example, a single insertion (or deletion) in a DNA sequence typically results in inserting (or deleting, respectively) a string of nucleotides, making multiple consecutive gap characters much more likely than isolated gap characters. The penalty function chosen should reflect this reality.

For this reason, researchers use gap penalties that do not increase linearly with the number of gap characters. In particular, researchers have studied general gap penalty functions [30], affine gap penalty functions [14, 17], and concave and convex gap penalty functions [31, 13, 15].

In terms of computational cost, alignments based on general gap functions cost the most to compute, and in practice they are hardly ever used. We shall discuss them here as motivation for choosing from among simpler functions. Alignments based on convex and concave gap penalty functions can be calculated in $O(nm\log(n+m))$ time. However, affine gap penalty functions offer enough flexibility and can be calculated almost as quickly as linear gap penalty functions. Therefore, these gap penalty functions are almost always used in practice.

1.7.1 General Gap Penalties

Envision a general gap penalty function $\omega(i)$ which is the penalty for inserting a gap of length i. If we allow for this arbitrary gap penalty function, then the runtime increases considerably. An $O(n^2m + nm^2)$ algorithm can be constructed by modifying the recursive definition presented for a linear gap penalty function.

Now, instead of considering a constant number of cells when calculating $S[i, j]$, we must consider $O(i + j)$ cells. This is because, when aligning two suffixes A_i and B_j, one must consider the possibility of aligning a_i with any character b_k $1 < k \leq j$. One must also consider aligning all of A_i with the empty string. Extending this idea to both strings we end up with four possibilities.

1. Align a_i with b_k, $1 < k \leq j$ and A_{i-1} with B_{k-1}.
2. Align b_j with a_k, $1 < k \leq i$ and A_{k-1} with B_{j-1}.
3. Align A_i with a gap
4. Align B_i with a gap

The equation for $S[i, j]$ becomes

$$S[i,j] = \max \begin{cases} \max_{k=1}^{j} S[i-1,k-1] + \delta(a_i, b_k) + \omega(j-k) \\ \max_{k=1}^{i} S[k-1,j-1] + \delta(a_k, b_j) + \omega(i-k) \\ \omega(i) \\ \omega(j) \end{cases}$$

Finally, for global alignment, we initialize the first row and column of the table based on the gap penalty function.

$$S[i,0] = \omega(i)$$
$$S[0,j] = \omega(j)$$

As $O(n+m)$ possibilities are considered for each cell, and there are $O(nm)$ cells, the total runtime of the algorithm is $O(n^2m + nm^2)$.

1.7.2 Affine Gap Penalties

The runtime penalty to allow the flexibility of a general gap penalty function is harsh. It would be nice to find a function with a more complex shape that incurs less of a runtime cost. Using *affine gap penalty* functions allows us to maintain an $O(nm)$ runtime.

An affine gap penalty function has two values. A gap opening penalty g is the cost of starting a new gap. A gap extension penalty h is the cost of extending a gap. The first gap character in an affine gap has a score of $g + h$. Each subsequent gap character in the affine gap has a score of h. Affine gap penalties are widely preferred by biologists because consecutive gap characters likely correspond to a single insertion/deletion event, while an equal number of scattered gaps correspond to as many insertion/deletion events, which is much less probable. For this reason h is often much smaller than g.

To solve the problem in quadratic time, we augment table S with two additional tables, G_A and G_B. $G_A[i, j]$ is the best score of aligning A_i with B_j under the restriction that a_i is aligned with a gap. $G_B[i, j]$ is the best score of aligning A_i and B_j under the restriction that b_j is aligned with a gap. As before, $S[i, j]$ holds the optimal score of aligning A_i and B_j under no restrictions.

There are three possible ways in which the maximum score can arise, with two of the cases consisting of two parts each.

1. a_i is aligned with b_j and A_i is aligned with B_j.
2. a_i is aligned with a gap. In this case, we must consider two sub-cases.
   (a) A_{i-1} is aligned with B_j such that a_{i-1} is not aligned with a gap, and we start a gap.
   (b) A_{i-1} is aligned with B_j such that a_{i-1} is aligned with a gap, and we extend the gap.
3. b_i is aligned with a gap. In this case we must consider two sub-cases.
   (a) A_i is aligned with B_{j-1} such that b_{j-1} is not aligned with a gap, and we start a gap.
   (b) A_i is aligned with B_{j-1} such that b_{j-1} is aligned with a gap, and we extend the gap.

These possibilities are captured and scored in the following equations:

$$S[i,j] = \max \begin{cases} S[i-1, j-1] + \delta(a_i, b_j) \\ G_A[i,j] \\ G_B[i,j] \end{cases}$$

$$G_A[i,j] = \max \begin{cases} S[i-1, j] + g + h \\ G_A[i-1, j] + h \end{cases}$$

$$G_B[i,j] = \max \begin{cases} S[i, j-1] + g + h \\ G_B[i, j-1] + h \end{cases}$$

The number of cells considered in each cell calculation is constant for all tables. Because there are $O(nm)$ number of cells per table, the total runtime of the algorithm is $O(nm)$.

Consider global alignment. We initialize the first row and column of each table such that $S[i, 0] = g + hi$, $S[0, j] = g + hj$, and $S[0, 0] = 0$. $S[n, m]$ contains the optimal alignment score after the algorithm finishes. We can construct the alignment by tracing back the path, starting at position $S[n, m]$. With an extension of the ideas presented for global alignment with linear gap penalties, the traceback can be accomplished without storing any pointers. The details are omitted.

Semiglobal alignment can be handled in a straightforward way. Initialize the first row and column of each table to 0. The maximum value in the last row and column of S is the optimal alignment score. Starting at this position, trace the alignment path back through the table until reaching the first row or column.

Assume that matches are scored positive, while mismatches and gaps are scored negative. Now local alignment with affine gaps is easy, as seen by the observation that every local alignment starts with some character a_k aligned with some b_l and ends with some $a_{k'}$ aligned with some $b_{l'}$. The proof is by contradiction. Assume that an optimal local alignment L starts with a character aligned with a gap. Then there exists a new alignment with higher score constructed removing the first character from L. Therefore, L is not optimal, a contradiction. The same reasoning holds for a gap character at the end of an alignment. As a result of this observation, we can handle local alignment by modifying the equation for $S[i, j]$

$$S[i,j] = \max \begin{cases} S[i-1, j-1] + \delta(a_i, b_j) \\ G_A[i,j] \\ G_B[i,j] \\ 0 \end{cases}$$

The alignment is found by searching for the maximum score in S, corresponding to aligning $a_{k'}$ and $b_{l'}$ by the observation above. The traceback continues until reaching some 0 in S, corresponding to the initial alignment of a_k and b_l.

1.8 Substitution Matrices

In this section we will consider the specific problem of aligning two amino acid sequences and the additional considerations needed in order to produce a biologically meaningful result.

Proteins are sequences of amino acids that fold into an energetically stable shape. The surface of a protein interacts with other proteins and molecules through its shape and chemical properties. It can be the case that proteins with rather different sequences can fold into molecules with similar shapes and properties — and consequently perform the same function. Moreover, a mutation occurring within the DNA sequence corresponding to the protein can result in an amino acid substitution, insertion, or deletion, having varying affects on the protein by affecting the protein's properties.

Some amino acid substitutions might be more acceptable than others. For example, six of the twenty amino acids are hydrophobic, which prefer to face the interior to avoid interacting with water. A substitution within this class of amino acids is more acceptable than a substitution with an amino outside of the class. For this reason, matching a hydrophobic amino acid with another hydrophobic amino acid should be scored higher than matching a hydrophobic amino acid with a hydrophilic one (an amino acid attracted to water).

In section 1.2, we presented a delta function for scoring the alignment of two characters:

$$\delta(x,y) = \begin{cases} \alpha & x = y \\ \beta & x \neq y \end{cases}$$

Now we wish to use a more complex function $\delta : D \times D \rightarrow \Re$, where D is the set of 20 amino acids. In practice this function is stored in a 20×20 matrix for use during the execution of an alignment algorithm. Two classes of such matrices called PAM and BLOSUM matrices are used, and we shall look at the origins of both.

1.8.1 PAM Matrices

A biologically valid scoring function δ arises from a complex process that is hard to model analytically. For this reason experimental data has been used to discover appropriate values. Dayhoff *et al.* [10, 11] described an evolutionary model used to interpret experimental data and derive the scoring function δ.

As an organism evolves, mutations will cause changes in the proteins. Those changes that are allowed to remain by an organism are said to be accepted or retained mutations. When comparing two proteins from divergent organisms, one would expect to observe some of these mutations as differences in the amino acid chains for the proteins of the two organisms.

Dayhoff *et al.* built a phylogenetic tree of closely related proteins in an attempt to discover accepted mutations. They accumulated the number of times amino acid i was substituted by amino acid j as one traveled up the phylogenetic tree. This data was stored in a 20×20 matrix, symmetric along the main diagonal, as a transition from i to j was considered a substitution from both i to j and j to i.

Using this data, they calculated the conditional probability of seeing an amino acid j given an amino acid i for all amino acid pairs (i, j). From this the PAM matrix was born. The PAM matrix stands for *Point Accepted Mutation*. From the experimental data, they calculated the probability matrix M. $M[i, j]$ contains the probability that amino acid i will be substituted for amino acid j in one evolutionary unit of time. The PAM evolutionary unit of time is the amount of time it takes for one amino acid in every hundred to undergo an accepted mutation.

Given the M matrix, the matrix M^k is the probability of substituting amino acids in k units of time. The PAMk scoring matrices are derived from the M^k probability matrices using the following equation, where p_j is the probability of a random occurrence of amino acid j.

$$\text{PAM}k[i, j] = 10\, log \frac{M^k[i, j]}{p_j}$$

1.8.2 BLOSUM Matrices

When considering protein sequences that are highly diverged, PAM matrices are not well suited as they were constructed based on closely related proteins with less than 15% difference. The BLOSUM matrices, introduced by Heinikoff and Heinikoff [20] are constructed using an approach that allows comparison of more highly diverged proteins. They are constructed using conserved regions of proteins. These regions, called blocks, give rise to the name BLOSUM, which stands for *BLock SUbstitution Matrix*.

The blocks are found by aligning multiple proteins in protein families (see Chapter 3 for a discussion of multiple alignments). As mentioned before, blocks are regions with a high degree of similarity. Within these regions, it could be the case that certain proteins are nearly identical. For this reason, in calculating the BLOSUMX matrix, multiple proteins that are X percent identical are weighted as one protein. Varying X gives rise to different scoring matrices, labeled BLOSUM30, BLOSUM50, BLOSOM62 and so forth.

The substitution frequencies are calculated based on the enumeration of all pairs of amino acids appearing in each column of the multiple alignment blocks. A value p_{ij} is calculated for amino acids i and j based on this multiple alignment. The details of this score are not easily described without a greater understanding of multiple alignment. If p_i is the probability of seeing amino acid i at random and p_j the probability of seeing amino acid j at random, then the BLOSUM matrix is calculated using the following equation:

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

TABLE 1.2 The BLOSUM62 matrix.

$$\text{BLOSUM}[i,j] = \frac{1}{\lambda} \log \frac{p_{ij}}{p_i p_j}$$

λ is a scaling factor used to generate scores that can be converted into integers. The BLOSUM62 matrix, the default matrix used for BLAST [3], is shown as Table 1.2.

1.9 Local Alignment Database Search

The dynamic programming algorithm finds the highest scoring alignment between two strings. However, performing a full alignment is often prohibitively slow. For example, if we were to compile a database of protein sequences, we could represent the database as a string D constructed by concatenating each string in the database. If we then attempted to find the optimal local alignment between some query Q and D, the runtime would likely be prohibitive, as the total cost would be the total length of all sequences in the database times the length of the query sequence.

Various approximations for local alignment have been proposed to speed up this basic problem of database search. The first of these was Fasta [25, 26], which we will not discuss here. In 1990, Altschul *et al.* presented the basic local alignment search tool [3] as a method to search protein databases quickly. We will present the second version of their algorithm, published in [27].

The basic idea behind BLAST is that good local alignments contain good ungapped alignments. An ungapped alignment is an alignment not allowing gaps. We wish to find these good ungapped alignments quickly and then extend them to find good local alignments.

Consider a window of size ω. As we move this window along the string, we can see ω characters of the string at a time. The number of unique strings, called ω-mers, of length ω is Σ^ω. If the strings are protein sequences, then the alphabet size is 20 and the number of strings we can make of length 3 is $20^3 = 8000$. We will create an index into the database showing all the locations of each ω-mer.

Additionally, we can calculate the score of aligning any ω-mer with any other ω-mer without gaps. Now, for a given word a, there is a set of words S for which $s \in S$ if and only if the score of aligning a with s is above some threshold T.

Now given a query string Q and a database D, we wish to find good local alignments

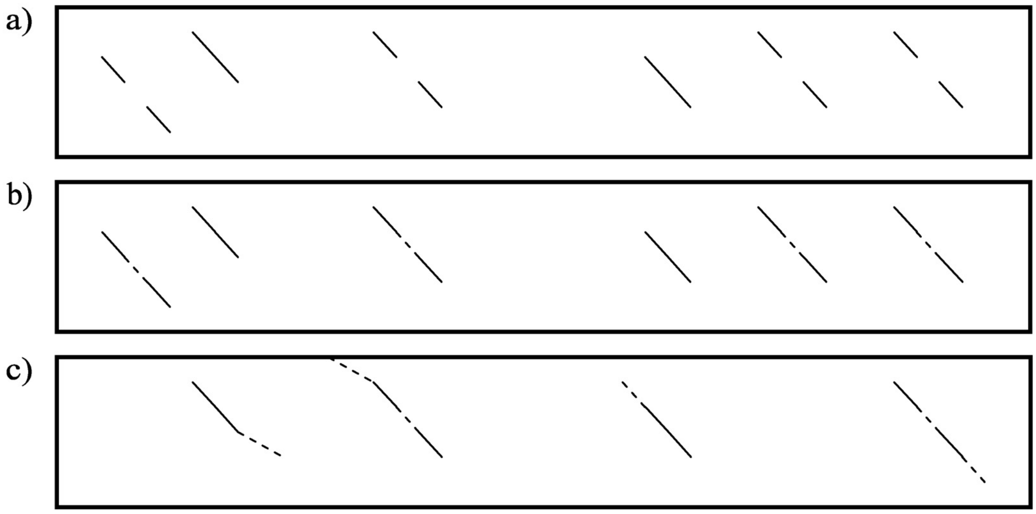


FIGURE 1.4: The BLAST program runs in phases. (a) In one phase pairs of hits are found that lie on the same diagonal of a conceptual dynamic programming table. (b) The region between these hits is aligned without gaps. (c) Finally, the alignment is completed by extending the ends of the alignment for those seed alignments scoring above some threshold.

between Q and D. The BLAST algorithm performs the following steps (see Figure 1.4).

1. The first step is to find hits between Q and the database. Each hit corresponds to some word a in Q matching to some word d in the database such that the score of aligning a and d is above the threshold.
2. From the hits discovered in step one, find those pairs of hits (h_i, h_j) that can be part of the same gapless alignment. That is, they would lie on one diagonal on the dynamic programming table.
3. Perform a gapless extension between these two hits by aligning each character in the query string with the corresponding character in the database. This will produce some alignment score.
4. For those gapless alignments with a score deemed significant, perform a gapped alignment extension from each end of the gapless alignment, such that the total score of the alignment does not drop below some threshold.

BLAST is a popular program for protein database searches, but recently researchers have revisited the problem. An algorithm called DASH (for Diagonal Aggregating Search Heuristic) reports runtimes ten times faster than BLAST with similar sensitivity [18]. Their heuristic extends the idea of BLAST. First they find all words occurring in the same diagonal region. Next, global alignments connect these gapless regions. Finally, they extend the end of the alignments using a global alignment on some part of the dynamic programming table. In addition to the diagonal region heuristic, a key technique they use to improve runtime is to mask those words that occur with high frequency in each sector of the database. This reduces the number of initial database hits.

1.10 Similarity and Distance Measures

This section details some alternate characterizations of the problem found in the bioinformatics field. We have described the solution to the alignment problem as finding the maximum score of an alignment between two strings. This score was the summation of the pairwise scores of each pair of characters involved in the alignment.

$$score(L) = \sum_{i=1}^{l} score(a_i, b_i)$$

The score is considered a measure of the similarity of the two strings, and it easily allows for the extensions into semiglobal and local alignment. However, one interesting result of these extensions is that the scoring system can fail to follow the triangle inequality.

$$score(A, B) + score(B, C) \leq score(A, C)$$

An alternate way of looking at the problem is to define a sequence of elementary operations on a string — insertions, deletions, and substitutions. One can transform A into B through a sequence of these operations $T = \langle t_1, t_2, ...t_n \rangle$. We now assign a cost function $cost(t)$ to the set of operations such that three conditions hold for any strings A, B, C.

1. $dist(A, A) = 0$
2. $dist(A, B) = dist(B, A)$ (symmetry)
3. $dist(A, B) + dist(B, C) \leq dist(A, C)$ (triangle inequality)

Where the $dist(A, B)$, called the *edit distance*, is the minimum sum of the cost of a sequence of operations that transforms A into B.

$$dist = \min_{T} \sum_{i} cost(t_i)$$

These requirements impose restrictions on our cost function. For symmetry, we require

1. $cost(insertion) = cost(deletion)$
2. $cost(substitute(a, b)) = cost(substitute(b, a))$

To satisfy the triangle inequality, we require

1. $cost(insertion), cost(deletion), cost(substitute(a, b)) \geq 0$

Distance measures have their uses, as the triangle inequality allows for certain reasoning and analysis that would otherwise be impossible. For example, performance guarantee proofs on approximation algorithms for the computationally expensive problem of multiple alignment are only valid using distance metrics. Multiple alignments are covered in Chapter 3.

An important result in the study of distance and similarity is that for any distance metric used in the alignment problem, one can construct a corresponding similarity metric. That is, finding the minimum distance for some cost function will simultaneously find the maximum score for some similarity function. Smith and Waterman [40] presented a theorem and proof of this assertion, and the idea is fully developed in [38]. The key observation is that for each alignment between two strings A and B, containing a matches, b mismatches, and g gaps, the following equation, known as the *alignment invariant*, is true:

$$n + m = 2(a + b) + g$$

In practice, we construct some scoring scheme based on the cost scheme using an arbitrary constant P.

$$\delta(a, b) = P - cost(sub(a, b))$$

$$\gamma = \frac{P}{2} - cost(insertion)$$

The maximum score and the distance under these valuation schemes are related by the equation:

$$score(A, B) + dist(A, B) = \frac{P(m + n)}{2}$$

Therefore, distances can be quickly calculated from similarity scores.

1.11 Normalized Local Alignment

Assume that we have two DNA sequences that we wish to compare using an alignment algorithm. Importantly, we wish to find regions of high similarity. Local alignment is somewhat suitable for this task, as it will return an alignment between substrings A' and B' that gives the highest scoring alignment.

However, there is a basic problem in the presentation of local alignment, in that the lengths of A' and B' are not taken into account when calculating the score. Therefore an alignment of length 100 and score 51 is considered better than an alignment of length 50 and score 50, although the second alignment has a much higher average score per base. Alexander and Solovyev [2] argued that the local alignment algorithm did not always find the most biologically relevant alignment because it did not consider alignment length.

One can think of post processing local alignments, but the highest scoring local alignment might mask some lower scoring alignment with higher normalized score. Instead, one could individually look at each pair of cells in the global alignment score matrix and compute the normalized alignment score.

$$\max_{i,j,k,l} \frac{S[i, j] - S[k, l]}{(i - k - 1) + (j - l - 1)}, 0 \leq i \leq k \leq n, 0 \leq j \leq l \leq m$$

However, the number of such combinations is $\Theta(n^2m^2)$, which is expensive.

The problem was explored in [41, 35, 4]. Pevzner *et al.* [5] were the first to provide an $O(nm(\log n))$ algorithm to compute the normalized alignment of some minimum length, and we will present their ideas here. For the sake of brevity, we will discuss the algorithm within the context of linear gap penalties, but the ideas extend easily to affine gap penalties, as shown by Pevzner *et al.*

The score of a best local alignment between two substrings A' and B' is $a\alpha + b\beta + g\gamma$, where a, b, and g are the number of matches, mismatches, and gaps. According to the *alignment invariant* first presented in Section 1.10, we have $n + m = 2a + 2b + g$, where n and m are the lengths of A' and B'. Pevzner proposed to measure the length of the alignment as $n + m + L$ where L is some positive constant. Then the length of some alignment with the score $a\alpha + b\beta + g\gamma$ is $2a + 2b + g + L$.

The best local alignment is found as:

$$LA(A, B) = \max_{(A', B')} a\alpha + b\beta + g\gamma$$

The best normalized local alignment is:

$$NLA(A,B) = \max_{(A',B')} \frac{a\alpha + b\beta + g\gamma}{2a + 2b + g + L}$$

The ideas used in solving the normalized local alignment quickly were introduced by Dinkelbach [12], who developed a general scheme for maximization problems which displayed the following properties:

1. The optimization involves a ratio $\frac{g}{h}$, where g and h are functions
2. The domain of g is equal to the domain of h
3. h is always positive

For our normalized local alignment, we have,

$$\max_{domain} \frac{g(a,b,g)}{h(a,b,g)}$$

Without going into details, we will illustrate some of the main ideas underlying this approach. First, we introduce an alignment called a *parametric local alignment* for some parameter λ.

$$PA(A,B,\lambda) = \max_{domain} g(a,b,g) - \lambda h(a,b,g)$$

$$PA(A,B,\lambda) = \max_{(A',B')} a\alpha + b\beta + g\gamma - \lambda(2a + 2b + g + L)$$

Dinkelbach's interesting result is that the following equation holds:

$$\lambda = NLA(A,B) \Leftrightarrow PA(A,B,\lambda) = 0$$

That is, λ is the score of the best normalized local alignment if and only if the parametric local alignment for λ has a score of zero.

Dinkelbach proposed an iterative search method to find the zero of $PA(A,B,\lambda)$ that has no provable run time but runs well in practice. His ideas are used in the following algorithm. First, initialize lambda by finding the local alignment $LA(A,B)$ and selecting $\lambda = \frac{a\alpha+b\beta+g\gamma}{2a+2b+g+L}$. Next, repeat two steps until lambda stops changing.

1. Find the parametric local alignment $PA(A,B,\lambda)$.
2. Set λ' to $\frac{a\alpha+b\beta+g\gamma}{2a+2b+g+L}$, and then set λ to λ'.

This method is faster in practice than the provably optimal alternative. However, if one restricts the values α, β, and γ to rational numbers, one can find the proper λ in $O(\log n)$ time using Megiddo's technique [29], the details of which are omitted here but can be found in [4].

The key to completing the algorithm is to effectively find $PA(A,B,\lambda)$. With some manipulation we see that parametric local alignment can be rewritten in terms of local alignment.

$$PA(A,B,\lambda) = \left(\max_{(A',B')} a(\alpha - 2\lambda) + b(\beta - 2\lambda) + g(\gamma - \lambda) \right) - L\lambda$$

To solve the parametric local alignment, we can first solve local alignment with $\alpha' = \alpha - 2\lambda$, $\beta' = \beta - 2\lambda$, and $\gamma' = \gamma - \lambda$, and then subtract a constant to find the score of the parametric local alignment.

Using this method of solving the parametric local alignment takes the same time and space as local alignment, $O(nm)$ time and $O(n+m)$ space. Therefore, if the scoring scheme is restricted to rational numbers then the time required to complete the normalized local alignment is $O(nm \log n)$. In practice, the Dinkelbach search is known to work equally well.

1.12 Asymptotic Improvements

Normalized local alignment is used to produce a more valid biological result. In the final section of this chapter, we explore some interesting techniques that can be used to reduce the asymptotic runtime complexity of the algorithm. For ease of presentation, assume $\Theta(m) = \Theta(n)$. It might seem at first glance that an $O(n^2)$ solution is as fast as the problem can be solved; however, this is not true. Masek and Paterson [28] were the first to introduce an $O\left(\frac{n^2}{\log n}\right)$ solution. However, their solution was limited in that it did not provide an answer to local alignment problem and required that the scoring method consist of rational numbers only.

Crochemore and Landau presented an algorithm [8] that answered these limitations. Their algorithm makes use of the periodic nature of strings to achieve a runtime of $O\left(\frac{hn^2}{\log n}\right)$, where h is the entropy [9] measure of the strings, varying between 0 and 1. Obviously even when the strings are random, with an entropy of 1, the algorithm shows an asymptotic improvement over $O(nm)$, but strings that are highly repetitive gain a larger improvement.

1.12.1 LZ Parsing of Strings

The algorithm uses a version of Lempel-Ziv parsing [24, 43, 44], which compresses a string by exploiting its repeat structure. The basic idea behind LZ compression is that one can divide a string S into a set of blocks. The blocks are formed in a greedy way, from left to right, using the following formulation. Suppose that we have divided the string into blocks up to position j and block i. In the Lempel-Zip parsing scheme, we will define block $i+1$ using a substring of $S[1...j]$ and a character c. More specifically, we look for the maximal substring $M = S[s...e]$ $(s \leq e \leq j)$ that matches $S[j+1...k]$. The new block is represented by the triple, $\langle s, l, c\rangle$, where s is the starting index of the substring M, l is the length of the substring M, and c is the character $S[k+1], S[k+1] \neq S[e+1]$.

It has been shown that the number of such blocks for a string of length n is $O\left(\frac{hn}{\log n}\right)$ [23], where again h is the entropy of the string. An example is given in Figure 1.5.

This is the most general version of LZ parsing. The alignment algorithm uses a slightly more restricted version known as LZ78. LZ78 parsing only allows the reuse of complete

```
LZ Parsing
a|g|gg|ga|ac|aacc|
(0,0,a) (0,0,g) (1,1,g) (1,1,a) (0,1,c) (4,3,c)
------------------------------------------------
LZ78 Parsing
a|g|gg|ga|ac|aa|c|c|
(0,a) (0,g) (2,g) (2,a) (1,c) (1,a) (0,c) (1,$)
```

FIGURE 1.5: Example LZ parsings of the string "agggaacacc".

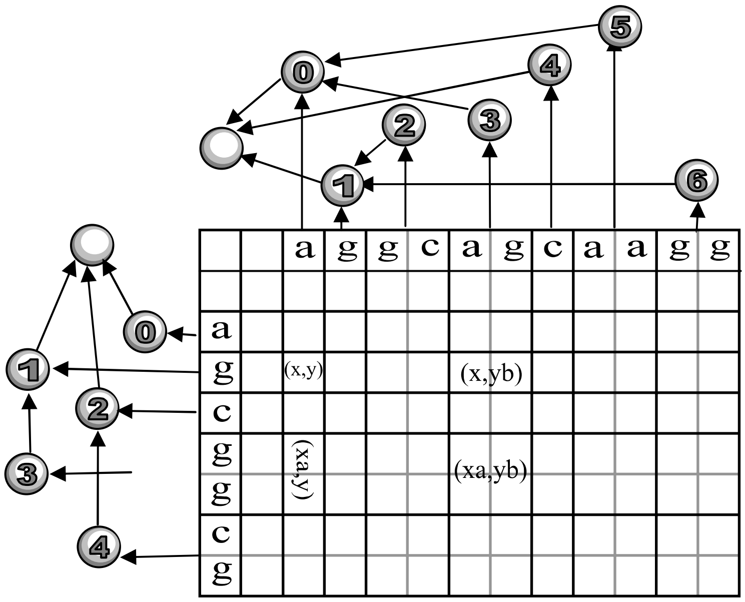


FIGURE 1.6: The dynamic programming table is decomposed into blocks based on each string's LZ78 block decomposition. In addition, two trie indexes are created that capture the structure of this decomposition. For each block (xa, yb), the blocks (x, yb), (xa, y), and (x, y) exist in the submatrix to the left and above block (xa, yb); the block indexes for substrings x and y can be found using the tries.

blocks rather than some arbitrary substring. One nice implication is that each block can be encoded using only two values $\langle i, c\rangle$, where i is the block index and c is the next character. Obviously, this method produces more blocks than the general scheme, but the total number of blocks is the same asymptotically, and the storage per block is less.

1.12.2 Decomposing the Problem

We will create a trie representation of the LZ78 parsing. For more information on tries and specifically suffix trees, refer to Chapter 5. The nodes of the trie correspond to each block in the parsing, and a node's parent corresponds to the block used as the prefix block. Edges point from children to the parents, and correspond to the extending character. The LZ78 parsing of the strings and the construction of the tries takes $O(n)$ time using suffix trees.

Using the block boundaries, as Figure 1.6 shows, we can conceptually divide the table into subtables, which we will also call blocks, as confusion can be avoided through context. Each block G defined for substrings xa of A and yb of B, written as (xa, yb), where x and y are strings and a and b are characters.

The intuition behind the algorithm is that the path information for all cells except for the bottom right cell should have been previously calculated. This is because blocks corresponding to $(xa, y), (x, yb)$, and (x, y) exist in the submatrix above and to the left of the current block. We want to use this observation to do work proportional to the number of cells on block boundaries, which is $O\left(\frac{hn^2}{\log n}\right)$, our goal.

First, consider viewing the alignment problem at the block level, as shown in Figure 1.7.

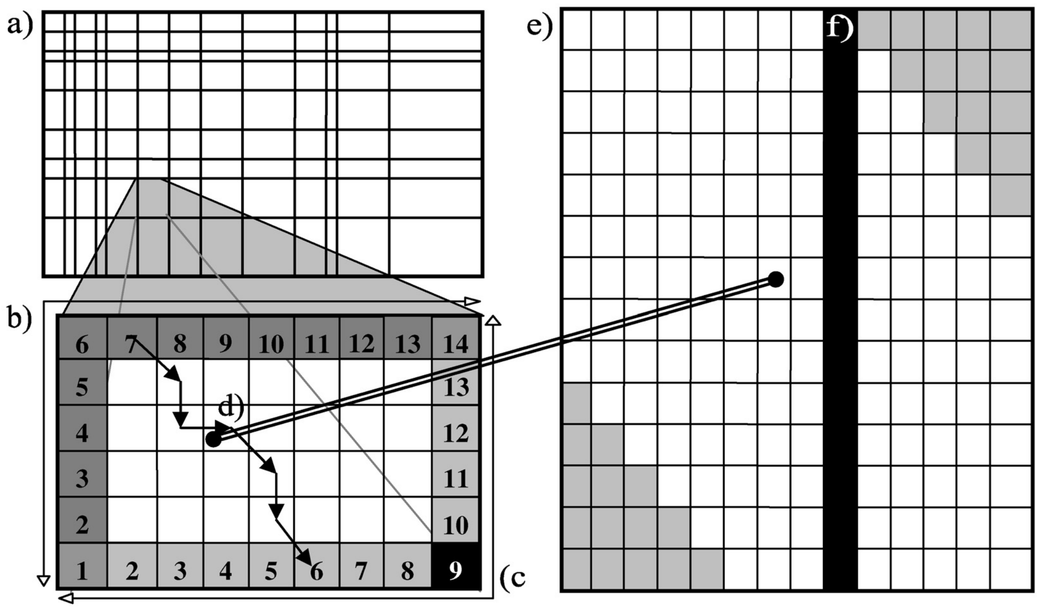


FIGURE 1.7: This figure shows an expanded view of each block in the dynamic programming table (a). The left column and top row are considered input cells for the block (b). The right column and bottom row are considered output cells (c). The total number of input cells or output cells is called p. One can consider every highest scoring path connecting each input cell with each output cell (d), and think of a $p \times p$ square matrix representing the score of each path through the block (e). This path matrix is incomplete as some input and output cells have no paths connecting them (gray). For this block, we only calculate and store one row of this matrix (f), which takes $O(p)$ time and space.

Let G be a block of width w and height h. We define the perimeter size of G as $p = w+h-1$. Note that this is not the same as the classical perimeter of G $(2(w+h))$. We call the left column and the top row the input cells of G and the right column and bottom row the output cells of G. The optimal path between some input cell *in* and some output cell *out* has a score $path[in, out]$. We create a $p \times p$ matrix called the path matrix, with the rows corresponding to the input cells and the columns the output cells. This matrix stores all optimal path scores for pairs of input cells and output cells.

For a block G defined as $G = B(xa, yb)$ where x and y are substrings and a and b are characters, we have the following information:

1. The score of all paths from input to output cells except for the bottom right cell, br. There are two reasons why this is the case. First, all paths must move down and to the right. Second, as stated previously, the blocks $B(x, yb), B(xa, y)$, and $B(x, y)$ have already been calculated.
2. The score of a best alignment path from the origin cell to each input cell, $input_i$.

We wish to calculate two things:

1. Scores of optimal paths from each input cell to the bottom right cell, br. This corresponds to one column in the path matrix as shown in figure 1.7. For each input cell *in*, the score is the maximum of three values:

$$path[in_i, br] = \max \begin{cases} path[in_i, northwest[br]] + \delta(a_i, b_j) \\ path[in_i, west[br]] + \gamma \\ path[in_i, north[br]] + \gamma \end{cases}$$

   Assuming that the three previously calculated path scores can be accessed in constant time, calculating the new path takes constant time for each input cell.
2. The input cell scores for the blocks neighboring our output blocks. This is accomplished by first calculating the output scores for each output cell, $output_j$, the score of the optimal path from the origin cell to the output cell.

$$output_j = \max_i (path[in_i, out_j] + input_i)$$

   The input cell scores can be calculated as the maximum of three values, using the bordering output cell scores.

It appears as if we have not reached our runtime goal. We wish to spend time proportional to the number of perimeter cells p in G. Certainly this is the case in step one, as we use a constant number of operations per input cell. However in step two it appears as if we break this requirement by searching for a max over all input cells for each output cell, which naively appears to take $O(p^2)$ time.

1.12.3 The SMAWK algorithm

It has been shown [1, 37] that the path matrix is *Monge* [32] by showing that $score[a, c] + score[b, d] \leq score[a, b]$ for all $a < b, c < d$, which is the concave requirement. In turn, the matrix is *totally monotone* because any Monge matrix is also totally monotone. Again our matrix meets the concave condition of total monotonicity: $score[a, c] \leq score[b, c] \Rightarrow score[a, d] \leq score[b, d]$.

Aggarwal and Park [1] gave a recursive algorithm, called SMAWK, that solves the problem of finding all row and column maxima in an $n \times n$ totally monotone, full, rectangular matrix in $O(n)$ time. The idea it uses is very simple: as one travels down the matrix from top to bottom, the row maxima must move left to right. Equivalently, the column maxima move top to bottom as one moves between columns left to right.

With this in mind, assume that we know the column maxima for all even columns. We can find the column maxima for all odd columns in $O(n)$ time by searching only the rows between column maxima in adjacent even columns. This alone does not produce the desired runtime; a simple recursion would produce an $O(n \log n)$ bound.

However, given any irregular matrix with more rows R than columns C, it is obvious that only C rows can actually produce column maxima. Using the total monotonicity property, we can find the set of rows producing maxima in $O(R)$ time by eliminating all rows not producing maxima.

We construct a stack s of size $|s|$ that contains the set of rows producing maxima. The top row on the stack will be represented as s_t and the next row down s_{t-1}, and row at position n is denoted as s_n. The stack is initially empty. We will consider the rows from top to bottom. We will place a row r on s only if $r[|s|] < s_t[|s|]$. If this is not the case then we will pop s_t off the stack, because it can contain no maxima. The test is repeatedly applied to r and the top of the stack until the condition is met or the stack becomes empty.

Why can we pop s_t off the stack when $r[|s|] \geq s_t[|s|]$? By total monotonicity, $r[c] \geq s_t[c]$ for all columns $c \geq |s|$. It is also the case that $s_{t-1}[c] \geq s_t[c]$ for all columns $c < |s|$. This can be proved as follows: Assume, for a contradiction, that $s_{t-1}[c] < s_t[c]$ for some $c < |s|$. Then, by total monotonicity, $s_{t-1}[c'] < s_t[c']$ for all $c' > c$. Therefore, $s_{t-1}[|s-1|] < s_t[|s-1|]$. However, by construction, $s_{t-1}[s-1] \geq s_t[s-1]$, a contradiction. Therefore, it must be the case that $s_{t-1}[c] \geq s_t[c]$ for all columns $c < |s|$. Therefore, unless s_t meets the condition, it can be discarded as containing no maxima.

It follows from the proof that row s_n may only contain column maxima for columns $c >= n$. This property is desirable because it bounds the stack size to C, as any rows placed on a stack of size C would not be able to contain any column maxima and can be thrown away. Therefore, when the algorithm is complete the stack contains at most C rows that will contain all column maxima. This set of rows will be fed into the recursion. The runtime of this algorithm is $O(R)$, as each row is pushed and popped off the stack at most once.

Thus we can halve both the row and column size for each recursive step in linear time, and the runtime is given by $O(n + \frac{n}{2} + \frac{n}{4} + ...) = O(n)$. Therefore, the total runtime of the algorithm is linear.

1.12.4 String Alignment in Subquadratic Time

Returning to our problem, we wish to use the SMAWK algorithm to find all column maxima in our path matrix. The algorithm can be adapted for this purpose only after we complete the matrix. The matrix is incomplete because paths must move down and to the right. Therefore some input and output cells cannot be connected by paths. Consequently, the highest scoring path between these cells is undefined. However, we can choose values for the corresponding positions in the matrix that will not result in row or column maximum while maintaining the totally monotone property.

1. For each cell in the upper right triangle, assign the value $-\infty$.
2. For each cell in the lower left triangle, assign the value $-(n + i + j)k$, where k is the maximum possible theoretical score of some path through the block, or $k = |w - |w - h||\alpha + |w - h|\gamma$.

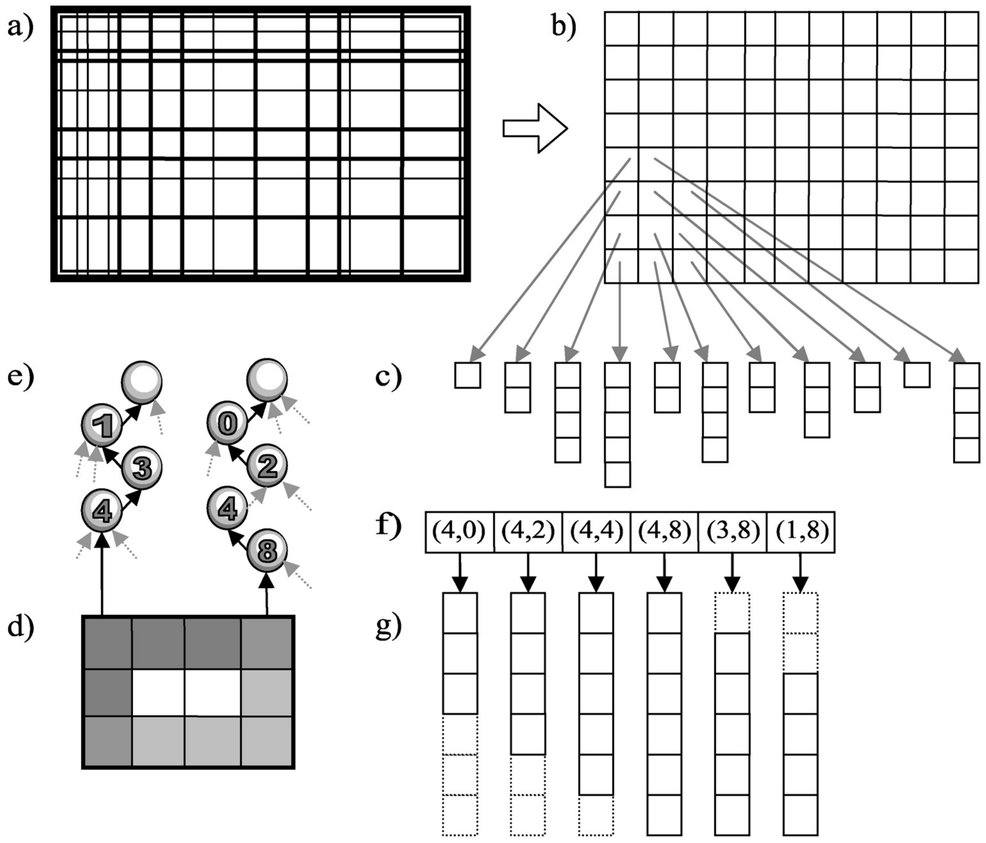


FIGURE 1.8: From the block decomposition of the dynamic programming table (a), we create an index (b) of size $O((\frac{hn}{\log n})^2)$ that points to the score column stored for each block (c). Each block (d) points to its corresponding node in the two trie indexes (e). Travelling up the path of these two trees, we can create a temporary array (f) that collects the rows needed to access the path matrix for this block (g) in constant time. Refer to Figure 1.7 for a description of the path matrix.

Now, let's say that for each row i corresponding to some input cell, we conceptually add $input_i$ to each cell in that row. After this operation, the totally monotone property is maintained as we change all values in each row by the same amount. After the addition, the result for each output cell is found by searching for each column maxima.

While we cannot spend the time to do the addition before running the SMAWK algorithm, we can do the addition for only those cells encountered during the run of the SMAWK algorithm and achieve the same affect. Therefore, we can find the needed maxima for all columns in time proportional to the number of rows and columns, which is the desired result.

Finally, we need to be able to find the path matrix values in constant time. However, we only have direct access to one column of the path matrix, the one that we constructed for this block. We need an indexing scheme that allows us to find the rows for all other output cells in constant time per cell. As shown in Figure 1.8, we will create a two dimensional array of size $O\left(\left(\frac{hn}{\log n}\right)^2\right)$ of pointers to the column stored for each block. This matrix in indexed by the tries we constructed for each string corresponding to the LZ decomposition of the string.

On the trie of each string, there is a path from the node for our current block to the root. Using the block IDs stored on the trie along this path as indexes into the two dimensional array, we can find the columns for the blocks used as prefixes of the current block in $O(1)$ time per block. There are p blocks of interest, each pointing to one column of our path matrix. To allow access to the path matrix values in constant time during the execution of the SMAWK algorithm, we create a temporary array of pointers to each column of the path matrix.

This is the last detail needed to finish the algorithm. In summary, the following steps are done for each block G with perimeter size p:

1. The column of the path matrix corresponding to connecting all input cells to the bottom right cell is constructed in $O(p)$ time using the columns for the blocks (x, yb), (x, y), and (xa, y), which can be found in constant time using the tries.
2. The path matrix for the block is constructed by constructing a temporary array of size p pointing to the columns of the matrix. This can be done in $O(p)$ time.
3. The output scores for this matrix are compiled using the SMAWK algorithm to find the column maxima. This also takes $O(p)$ time.
4. The input scores for the next block are calculated using the output scores from surrounding blocks, taking $O(p)$ time.

Therefore the total time for the algorithm is proportional to the number of perimeter cells, which as stated previously is $O\left(\frac{hn^2}{\log n}\right)$.

1.12.5 Space Requirements

Using Hirschberg's technique, we used $O(n+m)$ space and $O(nm)$ time to produce the alignment. In this section, we have described how to reduce the time required by the algorithm, but there is some expense. There is no known way in which to achieve subquadratic time and linear space in the same algorithm.

There is no published way to reduce the space bound without sacrificing flexibility, and we can find the space bound through a direct list of those data structures needed to solve the problem.

1. Two trie indexes corresponding to the block decomposition of our strings and two linear indexes into these trees, one for each row and column, as shown in Figure 1.6. This takes $O(\frac{hn}{\log n})$ space.
2. The block index structure corresponding to the block decomposition of S, as shown in Figure 1.8 (b). This takes $O\left(\left(\frac{hn}{\log n}\right)^2\right)$ space.
3. Input and output scores for each block, as shown in Figure 1.7 (b) and (c), taking p space per block, for a total of $O\left(\frac{hn^2}{\log n}\right)$ space.
4. One path matrix column for each block, as shown in Figure 1.8 (c), which takes p space per block, for a total of $O\left(\frac{hn^2}{\log n}\right)$ space.

Therefore, the total space complexity is the same as the runtime complexity, $O\left(\frac{hn^2}{\log n}\right)$.

1.13 Summary

In this chapter, we have provided a thorough presentation of fundamental techniques used to find the homology between a pair of DNA or protein sequences. While the basic alignment technique is simple to understand, the diversity of related problems quickly leads to new problem formulations and the resulting semiglobal, local, and banded alignments. We covered many advanced topics, including space saving techniques, normalized alignment, and subquadratic time alignment. Still, this chapter only represents an introduction to the field of alignments. The upcoming chapters in this part will expand on the ideas presented here and introduce a breadth of new formulations and solutions.

References

[1] A. Aggarwal and J. Park. Notes on searching in multidimensional monotone arrays. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 497–512, 1988.

[2] N.N. Alexander and V.V. Solovyev. Statistical significance of ungapped alignments. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 463–472, 1998.

[3] S.F. Altschul, W. Gish, W. Miller, and M. Myers *et al.* Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[4] A.N. Arslan and O. Egecioglu. Efficient algorithms for normalized edit distance. *Journal of Discrete Algorithms*, 1(1):3–20, 2000.

[5] A.N. Arslan, O. Egecioglu, and P.A. Pevzner. A new approach to sequence comparison: normalized sequence alignment. *Bioinformatics*, 17(4):327–337, 2001.

[6] C. Branden and J. Tooze. *Introduction to Protein Structure.* Garland Publishing, 1991.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[8] M. Crochemore, G.M. Landau, and Z. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 679–688, 2002.

[9] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[10] M.O. Dayhoff and R.M. Schwartz. Matrices for detecting distant relationships. In M.O. Dayhoff, editor, *Atlas of Protein Structure*, pages 353–358. National Biomedical Reasearch Foundataion, 1979.

[11] M.O. Dayhoff, R.M. Schwartz, and B.C. Orcutt. A model of evolutionary change in proteins. In M.O. Dayhoff, editor, *Atlas of Protein Structure*, pages 345–352. National Biomedical Research Foundation, 1979.

[12] W. Dinkelbach. On nonlinear fractional programming. *Management Science*, 13:492–498, 1967.

[13] D. Eppstein. Sequence comparison with mixed convex and concave costs. *Journal of Algorithms*, 11(1):85–101, 1990.

[14] D. Eppstein, Z. Galil, R. Giancarlo, and I. Italiano. Sparse dynamic programming I: linear cost functions. *Journal of the ACM*, 39(3):519–545, 1992.

[15] D. Eppstein, Z. Galil, R. Giancarlo, and I. Italiano. Sparse dynamic programming II: convex and concave cost functions. *Journal of the ACM*, 39(3):546–567, 1992.

[16] J. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12(1):175–179, 1984.

[17] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to

molecular biology. Technical Report 110–87, Columbia University Department of Computer Science, 1987.

[18] P. Gardner-Stephen and G. Knowles. DASH: localising dynamic programming for order of magnitude faster, accurate sequence alignment. In *Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference*, pages 732–733, 2004.

[19] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[20] S. Henikoff and J.G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Acadamy of Sciences of the U.S.A*, 89(22):10915–10919, 1992.

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

[22] X. Huang, R.C. Hardison, and W. Miller. A space-efficient algorithm for local similarities. *Computer Applications in Biosciences*, 6(4):373–381, 1990.

[23] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proceedings of the 3rd South American Workshop on String Processing*, pages 141–155, 1996.

[24] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):783–795, 1976.

[25] D.J. Lipman and W.R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[26] D.J. Lipman and W.J. Wilbur. Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences of the U.S.A.*, 80(3):726–730, 1983.

[27] D.J. Lipman, J. Zhang, R.A. Schffer, and S.F. Altschul *et al.* Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[28] W.J. Masek and M.S. Paterson. A faster algorithm for computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[29] N. Megiddo. Combinatorial optimization with rational objective functions. *Mathematics of Operations Research*, 4(4):414–424, 1979.

[30] J. Meidanis. Distance and similarity in the presence of nonincreasing gap-weighting functions. In *Proceedings of the 2nd South American Workshop on String Processing*, pages 27–37, 1995.

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

[32] G. Monge, Deblai, and Rembai. *Memoires del l'Academie des Sciences*, 1781.

[33] E.W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[34] 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.

[35] B.J. Oommen and K. Zhang. The normalized string editing problem revisited. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(6):669–672, 1996.

[36] D. Sankoff and J.B. Kruskal. *Time Warps, String Edits and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison Wesley, 1983.

[37] J.P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.

[38] C. Setubal and J Meidanis. *Introduction to Computational Biology*, chapter 3, pages

47–104. PWS Publishing, 1997.

[39] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[40] T.F. Smith, M.S. Waterman, and W.M. Fitch. Comparative biosequence metrics. *Journal of Molecular Evolution*, 18:38–46, 1981.

[41] E. Vidal, A. Marzal, and P. Aibar. Fast computation of normalized edit distances. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(9):899–902, 1995.

[42] M.S. Waterman. *Mathematical methods for DNA sequences.* CRC Press, 1991.

[43] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[44] J. Ziv and A. Lempel. Compression of individual sequences via variable rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.