Chapter 21

BIOINFORMATICS

Srinivas Aluru
Iowa State University

1 INTRODUCTION

Ever since the structure of DNA was discovered in the early 1950s, biology has been steadily transforming into a discipline that relates essential life processes to underlying biomolecular data. This discovery has stimulated the growth of molecular biology, the study of how biomolecular sequences influence the functioning of organisms. These developments have brought biology closer to computer science. In many ways, the underlying mechanisms are similar to what we employ in building and programming computers. The characteristics of a life form are coded in its DNA (program), which is processed in each cell (executed) to produce the proteins (outputs) that carry out most of the essential life processes. The field holds immense potential for future discoveries that are unrivaled in significance, such as the possibility of treating diseases and engineering improved crops by altering the genetic composition.

The need to discover biomolecular sequences, to relate those sequences to their structure and function, and to understand evolutionary history through sequence homology (similarity) detection has resulted in a number of interesting problems for computer scientists and led to the development of bioinformatics. Broadly defined, bioinformatics or computational biology is the study of computational methods for furthering biological discovery and applying information technology to solving the problems of biological relevance. The field has experienced an explosive growth in the last two decades, and the accumulated knowledge and importance of the field has reached a stage in which successful advanced graduate programs are being developed to train bioinformaticists. While it is impossible to attempt a comprehensive coverage of this field in a short amount of space, this chapter is intended to provide both a sense of the breadth of the field and a focused study of specific applications, particularly in computational genomics.

2 OVERVIEW OF BIOINFORMATICS

In this section, we present some basic concepts in molecular biology that are essential for understanding the remainder of the chapter, and we also outline a number of computational challenges in bioinformatics. The reader should keep in mind that the discussion is purposefully oversimplified and should refer to a standard textbook [2, 65] for full details.

2.1 Basics of Molecular Biology

In bioinformatics, we are typically concerned with two types of biomolecular data: DNA (deoxyribonucleic acid) sequences and protein sequences. A DNA molecule is a sequence made of simpler molecules known as nucleotides. Each nucleotide consists of a deoxyribose sugar molecule, a phosphate group attached to the 5′-carbon of the sugar molecule, and a base attached to the 1′-carbon of the sugar molecule. The different nucleotides are differentiated by the differences in the bases—Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). For computational purposes, a DNA sequence can be represented as a string over the alphabet $\Sigma = \{A, C, G, T\}$, specifying each nucleotide by the first letter of its name. The sequence is formed by phosphodiester bonds between consecutive nucleotides: the 5′-carbon of one nucleotide is linked to the 3′-carbon of the previous nucleotide through the phosphate group. Thus, one end of the sequence has a free 5′ end and the other end has a free 3′ end, giving a directionality to the molecule. It is customary to write a DNA molecule as the sequence of nucleotides from the 5′ end to the 3′ end.

DNA sequences naturally occur as double-stranded molecules, i.e., two sequences of nucleotides attached to each other. The two strands are held together by hydrogen bonds between bases of the corresponding nucleotides. Two types of base pairings are possible – A with T, involving two hydrogen bonds; and G with C, involving three hydrogen bonds. For a given nucleotide in one strand, the corresponding nucleotide in the complementary strand is given by the pairing $A \leftrightarrow T$ and $C \leftrightarrow G$. The two complementary strands also exhibit opposite directionality. Because of these properties, a double-stranded DNA molecule can be accurately described as the sequence of one of its strands from its 5′-end to the 3′-end. Note that this would mean two equivalent strings describing the same DNA molecule. One string (or strand) can be obtained from the other by a *reverse complementation* operation, which refers to reversing the string and replacing A with T, T with A, C with G, and G with C. The length of a DNA sequence is measured in units called *base pairs* (*bp*), where a base pair refers to a pair of corresponding nucleotides on the two strands of a DNA sequence.

DNA is established as the vehicle for passing hereditary genetic information. The complementarity relation between the two strands of a DNA sequence indicates that one strand is sufficient to recover the entire sequence. This mechanism makes DNA self-replicating. Several different terms are used to describe DNA sequences, depending on the role played by particular DNA sequences or the scale at which these sequences are viewed. The term *genome* refers to the entire genetic constitution within the nucleus of a cell of a *eukaryotic* organism (organism whose cells have nuclei) or within a cell of a *prokaryotic* organism (organisms

whose cells do not have nuclei). During cell division, the genome is duplicated using the self-replicating mechanism to provide a copy for each resulting cell. The genome is organized into one or more *chromosomes*, where each chromosome is a continuous strand of DNA. A *gene* is a contiguous stretch of DNA along a chromosome that codes for a protein. A *promoter* is a DNA sequence typically located upstream of a gene to aid in its expression. Significant length scales are exhibited in the sizes of genomes. Viruses have the smallest of the genomes, e.g., the virus Bacteriophage λ has an approximate size of 50,000 *bp*. Bacterial genomes are typically 100 times as large or more. Humans, mice, and maize have genomes about 3×10^9 *bp* in size. Plants are known to have some of the largest genomes. For example, the Lily plant has a genome about 100×10^9 *bp* long.

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 acid residues. An amino acid consists of a central carbon atom known as α-carbon, connected to a hydrogen atom, a carboxyl group, an amino group, and a side chain. It is the side chains that distinguish the twenty different amino acids that constitute protein sequences. Amino acid residues are typically denoted by a three-letter abbreviation of their names, such as *Gly* for *Glycine* and *Val* for *Valine*. For computational purposes, we will use a single letter alphabet of size twenty and depict protein sequences as strings over this alphabet.

The mechanism by which a gene codes for a protein is as follows: First, a copy of the gene (or portions of it intended to code for a protein sequence) is made as an RNA (ribonucleic acid) molecule, called messenger RNA, or mRNA for short. Similar to DNA, mRNA is a sequence of nucleotides except that Uracil (U) is used instead of Thymine (T). A *codon* is a consecutive triplet of nucleotides in the mRNA sequence that codes for an amino acid. The many-to-one mapping between the 64 possible codons and the twenty amino acids has been discovered and is common across species (see Table 21.1). Three codons correspond to a STOP signal. The translation typically starts with the codon *AUG*, which codes for the amino acid methanine, and continues until a STOP signal is encountered. The copying of DNA to mRNA is called *transcription*, and the production of protein from mRNA is called *translation*. Together, this process is popularly known as *the central dogma* in molecular biology.

Proteins are responsible for carrying out most of the essential life processes. For example, they act as tissue building blocks (structural proteins) and as catalysts to speed up biochemical reactions (enzymes); they carry out oxygen transport and conduct antibody defense. The three-dimensional structure of proteins is critical to their function. Complex regulatory mechanisms guide the gene-expression process, which together with other factors will ultimately determine the amount of production of the corresponding protein. Multiple forms of the same gene, known as *alleles*, cause differences in *genotype* (genetic difference) between individuals, which will eventually translate into differences in *phenotype* (observable differences, such as color of eyes). Certain variations in a gene sequences may lead to low or nonfunctional proteins and may cause genetic diseases or increase susceptibility to diseases. These differences often arise due to a change in the nucleotide in a single position, also called a *single nucleotide polymorphism (SNP)*. Developing a database of SNPs along the genome is considered vital to pharmaceutical research. Variations within the genome across different

Table 21.1 The genetic code mapping a consecutive triplet of nucleotides (codon) to the corresponding amino acid. Note that multiple codons code for the same amino acid. With the exception of Ser, the first two positions of the codons that code for the same amino acid are identical.

	Second Position				
First position	U	C	A	G	Third position
	Phe(F)	Ser(S)	Tyr(Y)	Cys(C)	U
U	Phe(F)	Ser(S)	Tyr(Y)	Cys(C)	C
	Leu(L)	Ser(S)	Stop	Stop	A
	Leu(L)	Ser(S)	Stop	Trp(W)	G
	Leu(L)	Pro(P)	His(H)	Arg(R)	U
C	Leu(L)	Pro(P)	His(H)	Arg(R)	C
	Leu(L)	Pro(P)	Gln(Q)	Arg(R)	A
	Leu(L)	Pro(P)	Gln(Q)	Arg(R)	G
	Ile(I)	Thr(T)	Asn(N)	Ser(S)	U
A	Ile(I)	Thr(T)	Asn(N)	Ser(S)	C
	Ile(I)	Thr(T)	Lys(K)	Arg(R)	A
	Met(M)	Thr(T)	Lys(K)	Arg(R)	G
	Val(V)	Ala(A)	Asp(D)	Gly(G)	U
G	Val(V)	Ala(A)	Asp(D)	Gly(G)	C
	Val(V)	Ala(A)	Glu(E)	Gly(G)	A
	Val(V)	Ala(A)	Glu(E)	Gly(G)	G

individuals of the same species are very small compared with the length of the genome. For instance, all humans are expected to show over 99.9% identity at the genome level. A typical high-level organism contains several tens of thousands of genes. Genes are conserved across species, and species that are evolutionarily closer exhibit significant gene homologies.

Several experimental procedures have been designed to complement the molecular biological discoveries summarized above. DNA sequences that are several hundred base pairs long can be read using an experimental procedure known as Sanger's method. A number of recombinant DNA techniques have been developed. These include (1) inserting foreign DNA into bacterial genomes for the purpose of cloning, (2) amplifying DNA sequences using the polymerase chain reaction (PCR) method, which corresponds to exponential growth via doubling, (3) artificially converting mRNA sequences to the corresponding DNA sequences, called *complementary DNA* or *cDNA* sequences, and (4) testing for the presence of a particular DNA sequence by using its complementary strand. These and other experimental techniques have been used to deduce DNA and protein sequence data from a plethora of organisms. Such data are deposited in public databases such as GenBank (http://www.ncbi.nlm.nih.gov) and PDB (http://www.pdb.org). Exponential growth in the size of such databases has necessitated computational methods for accessing and analyzing sequence data.

2.2 Computational Challenges

Computational methods and the use of software have become integral parts of a biologist's toolkit. Their use is pervasive, encompassing the discovery, analysis, and interpretation of biological data, aiding in the discovery of biological knowledge, and helping utilize this knowledge in applications in biotechnology

and medicine. A systematic study of this interdisciplinary research field has led to a number of important subareas within bioinformatics, some of which are described below.

1. **Alignments and Database Search.** Alignment methods are intended to discover homologies (similarities) of interest between DNA or protein sequences. Alignment algorithms are used to query databases to discover homologous sequences, discover homologous genes within or across species, identify common motifs across multiple protein sequences, and identify overlapping sequences.
2. **Genome Sequencing.** While laboratory sequencing techniques can read DNA sequences several hundred base pairs long, genome sizes of higher organisms are more than a millionfold larger in size. The approach used for genome sequencing is to derive an appropriate number (tens of millions for human and mouse genomes) of random fragments of sequenceable size from the genome. Once these fragments are sequenced, computational methods are designed to assemble the fragments to derive the genome sequence.
3. **Gene Identification and Annotation.** An important first step in understanding the genome of an organism is to identify the locations and structures of its genes and to identify the role of the corresponding protein products (annotation). Computational approaches designed for this problem include ab initio gene prediction methods using hidden Markov models, alignment programs using gene transcriptions such as mRNA and cDNA sequences, and close comparison of related species to identify conserved regions.
4. **Comparative Genomics.** Genome comparisons are useful in identifying conserved genes, promoters, and other sequences; validating and annotating of genome assemblies; and understanding evolutionary histories. Comparative genomics also throws light on genome rearrangements and fast-evolving viruses.
5. **Gene Expression Analysis.** DNA microarrays facilitate profiling of the expression levels of tens of thousands of genes in a single experiment. Such information is used in identifying coregulated genes, inferring gene regulatory networks, identifying genes whose abnormality causes specific diseases, studying developmental genetics, etc.
6. **Phylogenetic Analysis.** The study of the evolution of sequences and species and the deciphering of the evolutionary history connecting known and extinct species is called *phylogenetics*.
7. **Protein Structure Prediction.** The three-dimensional structure of a protein is crucial to its function. The ability of a deformed protein molecule to fold back into its native configuration without external assistance led to the hypothesis that the structure is determined by the sequence itself. Computationally determining the structure of a protein from its sequence is considered a "holy-grail" problem in bioinformatics. A corresponding problem is that of inverse protein folding, the problem of finding a protein sequence that will fold into a desired three-dimensional structure.
8. **Structural Homologies and Docking.** While sequence homologies often translate into structure homologies, the converse need not be true. Furthermore, preservation of important structural motifs is sufficient for functional similarity despite differences in other parts of the structure. While sequence alignment

algorithms are used because of their ease and simplicity, detection of structural homologies and structure-based database searches would be the eventual goal for protein sequences. Similarly, an understanding of protein docking is useful in designing proteins for the effective administration of drugs.

The above list is not meant to be exhaustive but is intended to convey the breadth, importance, and interesting nature of the research problems in bioinformatics and computational biology. Computational techniques have already become an integral part of biological discoveries, and this trend will continue in the future. In the remainder of this chapter, we will focus on specific research problems in an attempt to convey the flavor and excitement of this interdisciplinary research field and the challenging applications it provides for computer science research. We will begin with problems in computational genomics. For a high-level view of the role of algorithmic research in computational genomics, the reader is referred to Karp's recent keynote address [56].

3 BASIC TOOLS OF COMPUTATIONAL GENOMICS

3.1 Alignments

Global Sequence Alignments

Consider the problem of determining whether two DNA sequences are evolutionarily related and detecting the extent of homology (similarity) between them. To model this problem computationally, one must understand the evolutionary mechanisms that could change DNA sequences. Two types of events are of primary interest: mutation, a process that results in substitution of one nucleotide with another; and DNA insertions/deletions, which cause insertion/deletion of a contiguous subsequence. Suppose that DNA sequence A is changed to DNA sequence B through some substitution, insertion, and deletion operations. The homology between sequences A and B can be shown by writing one sequence below the other to clearly indicate matching nucleotides and substitutions. An insertion used in transforming A into B (referred to as an insertion in A) is shown by a sequence of gaps in A corresponding to the inserted subsequence in B. Similarly, a deletion in A is shown by a sequence of gaps in B corresponding to the deleted subsequence in A. For example, Figure 21.1 shows an alignment of DNA sequences $ATGTCGA$ and $AGAATCTA$ obtained by deleting the second base, inserting AA after the third base, and substituting T for the sixth base in $ATGTCGA$.

In order to measure the significance of homology shown by an alignment, a scoring scheme is introduced. The idea is to reward matches and penalize substitutions and insertions/deletions, abbreviated *indels*. A higher score indicates a better alignment. Since the same sequences could be represented using different alignments, the scoring mechanism also provides a way to evaluate how good an alignment is. Thus, the alignment problem can be formulated as a problem of finding the highest scoring alignment between two sequences. The highest score, or optimal score, becomes a measure of the homology between the two sequences.

A	T	G	–	–	T	C	G	A
A	–	G	A	A	T	C	T	A
5	–5	5	–6		5	5	–5	5

Figure 21.1. An alignment between DNA sequences *ATGTCGA* and *AGAATCTA* using a score of 5 for a match, −5 for a substitution, 4 for a gap opening penalty, and 1 for a gap extension penalty.

The above ideas can be formalized as follows: Let Σ be the alphabet, and let '–' denote the gap. A score function $f : \Sigma \times \Sigma \rightarrow \Re$ prescribes the score for any column in the alignment that does not contain a gap. Scores of columns involving gaps are determined by an *affine gap penalty function*: for a maximal consecutive sequence of k gaps, a penalty of $h + gk$ is applied. Thus, the first gap in a maximal sequence is charged $h + g$, while the rest of the gaps are charged g each. The term h is called the *gap opening penalty*, and the term g is called *gap extension penalty*. If $h = 0$, the penalty function is called a *constant gap penalty* function. The score of the alignment is the sum of scores over all the columns. The alignment in Figure 21.1 is scored using the simple scoring function, defined as

$$f(c_1, c_2) = \begin{cases} 5, & c_1 = c_2, c_1, c_2 \in \Sigma \\ -5, c_1 \neq c_2, c_1, c_2 \in \Sigma \end{cases}$$

and an affine gap penalty function that penalizes a maximal sequence of gaps of length k with a penalty of $4 + k$. Then the alignment has a total score of 9.

Let $A = a_1\, a_2 \ldots a_m$ and $B = b_1\, b_2 \ldots b_n$ be two sequences. An optimal alignment of A and B is computed using a dynamic programming approach. The algorithm uses three tables T_1, T_2, and T_3, each of size $(m + 1) \times (n + 1)$. An entry $[i, j]$ in each table corresponds to the score for optimally aligning $a_1 a_2 \ldots a_i$ with $b_1\, b_2 \ldots b_j$, but with the following conditions: In T_1, only alignments in which a_i is aligned with b_j are considered. In T_2, b_j must be aligned with "–", and in T_3, a_i must be aligned with "–." Once the tables are computed, the optimal score for aligning A with B is given by the maximum of $T_1[m, n]$, $T_2\,[m, n]$, and $T_3\,[m, n]$.

The top row and leftmost column of each table are initialized to $-\infty$, except in the following cases ($1 \leq i \leq m$; $1 \leq j \leq n$):

$$T_1[0, 0] = 0$$
$$T_2[0, j] = -(h + gj)$$
$$T_3[i, 0] = -(h + gi)$$

A score of $-\infty$ is used to indicate that the alignment is invalid. Consider the task of computing $T_1\,[i, j]$ for some $i \geq 1$ and $j \geq 1$. The last column of the alignment contains a_i aligned with b_j, which gets a score of $f(a_i, b_j)$. The remaining portion of the alignment must be an optimal alignment between $a_1\, a_2 \ldots a_{i-1}$ *and* $\mathrm{b}_1\, \mathrm{b}_2 \ldots \mathrm{b}_{j-1}$. This is given by the maximum of $T_1\,[i-1, j-1]$, $T_2\,[i-1, j-1]$, and $T_3\,[i-1, j-1]$, which can be computed in constant time if these entries are already available. Similar reasoning gives rise to the following recurrence equations:

$$T_1[i, j] = f(a_i, b_j) + \max \begin{cases} T_1[i-1, j-1] \\ T_2[i-1, j-1] \\ T_3[i-1, j-1] \end{cases} \tag{1}$$

$$T_2[i,j] = \max \begin{cases} T_1[i,j-1] - (g+h) \\ T_2[i,j-1] - g \\ T_3[i,j-1] - (g+h) \end{cases} \quad (2)$$

$$T_3[i,j] = \max \begin{cases} T_1[i-1,j] - (g+h) \\ T_2[i-1,j] - (g+h) \\ T_3[i-1,j] - g \end{cases} \quad (3)$$

The tables can be filled row by row or column by column. Either way, when a table entry is computed, the entries needed to compute it are already available. As each table entry can be computed in constant time, the algorithm runs in $O(mn)$ time. The tables can be used not only to find the optimal score but also to retrieve one or all optimal alignments. To do this, for each table entry that is being filled, a pointer is maintained that points to the appropriate table entry that resulted in the highest score among all alternatives being considered. If multiple alternatives result in the same highest score, pointers are maintained linking to each of the corresponding table entries. An optimal alignment is recovered from right to left by following the trail of pointers from a highest entry among $T_1[m, n]$, $T_2[m, n]$, and $T_3[m, n]$ to the top left corner of one of the tables. This procedure is often called *traceback*. As each pointer causes a move to the previous row or column or both, an optimal alignment can be retrieved in $O(m + n)$ time. By enumerating all possible paths, all optimal alignments can be enumerated if desired.

Space and Time Reduction Techniques

The time and space requirements of the just-described algorithm for computing sequence alignments are both $O(mn)$. The space requirement can be reduced to $O(m + n)$ using Hirschberg's technique [43, 76] while increasing the run-time by at most a factor of 2. First, note that the entries required for computing row i of the tables T_1, T_2, and T_3 depend only on row $i - 1$ of the tables. By discarding a row as soon as the next row is computed based on it, the space used per table can be reduced to $O(n)$ ($O(min(m, n))$ by choosing B to be smaller of the two input sequences). The total space required for the algorithm is reduced to $O(m + n)$, including the space for storing the input sequences. It is still possible to determine the optimal score because only the last entry of the last row of each of the tables is required to compute it. The only problem with this approach is that the ability to perform traceback and retrieve an optimal alignment is lost. From the biologist's perspective, alignments are crucial.

Hirschberg's strategy uses divide-and-conquer to find both the optimal score and an optimal alignment using only $O(m + n)$ space. Let $A^r = a_m a_{m-1} \ldots a_2 a_1$ and $B^r = b_n b_{n-1} \ldots b_2, b_1$ denote the reverse of sequences A and B, respectively. Similarly, let T_1^r, T_2^r, and T_3^r denote tables defined similar to T_1, T_2, and T_3 except that entry $[i,j]$ corresponds to an alignment between $a_m a_{m-1} \ldots a_i$ and $b_n, b_{n-1} \ldots b_j$, i.e., $T^r[i, j]$ denotes the score of an optimal alignment between $a_m a_{m-1} \ldots a_i$ and b_n, b_{n-1}, b_j, where a_i is aligned with b_j, etc. Let $k = \lfloor \frac{m}{2} \rfloor$. Compute row k of T_1, T_2, and T_3, and row $k + 1$ of T_1^r, T_2^r, and T_3^r. This is equivalent to filling the top half of the rows of the tables T_1, T_2, and T_3 and the bottom half of the rows of the tables

T_1^r, T_2^r, and T_3^r. Using the space-saving strategy of discarding previously computed rows, the required rows can be computed in $O(mn)$ time and $O(m + n)$ space.

Consider an optimal alignment of A and B. Partition the alignment into two parts by separating the alignment immediately after the column containing a_k. The first part is an alignment between $a_1 a_2 \ldots a_k$ and $b_1 b_2 \ldots b_j$ for some j, and the second part is an alignment between $a_{k+1} a_{k+2} \ldots a_m$ and $b_{j+1} b_{j+2} \ldots b_n$. Because of affine gap penalties, these need not be optimal alignments, but they would be of the type captured by row k of T_1, T_2, and T_3, and row $k + 1$ of T_1^r, T_2^r, and T_3^r. The value of j and the optimal score can be computed by choosing a value of j that maximizes

$$\max \begin{cases} T_1[k,j] + \max(T_1^r[k+1, j+1], T_2^r[k+1, j+1], T_3^r[k+1, j+1]) \\ T_2[k,j] + \max(T_1^r[k+1, j+1], T_2^r[k+1, j+1] + h, T_3^r[k+1, j+1]) \\ T_3[k,j] + \max(T_1^r[k+1, j+1], T_2^r[k+1, j+1], T_3^r[k+1, j+1] + h) \end{cases}$$

The reason for adding h when combining $T_2[k, j]$ and $T_2^r[k+1, j+1]$ or when combining $T_3[k, j]$ and $T_3^r[k+1, j+1]$ is to avoid charging a gap opening penalty twice, once at the beginning and once at the end of a maximal sequence of gaps. Let $T[k, j]$ denote the maximum of $T_1[k,j]$, $T_2[k,j]$, and $T_3[k,j]$, and let $T^r[k + 1, j + 1]$ denote the maximum of $T_1^r[k + 1, j + 1]$, $T_2^r[k + 1, j + 1]$, and $T_3^r[k + 1, j + 1]$. The above equation can be simplified to

$$\max \begin{cases} T[k,j] + T^r[k+1, j+1] \\ T_2[k,j] + T_2^r[k+1, j+1] + h \\ T_3[k,j] + T_3^r[k+1, j+1] + h \end{cases}$$

Once the partitioning of an optimal alignment with respect to the middle of sequence A is found, this procedure is applied recursively to each partition. The recursive decomposition is continued until one of the sequences has a single character, which is then solved directly. This leads to the recurrence

$$t(m,n) = O(mn) + t\left(\lfloor \frac{m}{2} \rfloor, j\right) + t\left(\lceil \frac{m}{2} \rceil, n - j\right)$$

where $t(k, j)$ denotes the run-time for aligning two sequences of lengths k and j, respectively. Solving the recurrence shows that the run-time is $O(mn)$.

Hirshberg's strategy greatly reduces the space required for aligning sequences, making it feasible to perform alignments on very large sequences. Moreover, this strategy confers great practical benefit even for modest-sized sequences because the entire space required may fit in cache memory. While this technique resolves the space problem, the quadratic run-time poses a problem for aligning very large sequences or for carrying out a large number of alignments on short sequences, as is required in many applications. Asymptotically faster sequence alignment algorithms that improve run-time complexity by a log factor have been designed [19, 68]. However, their practical benefits are unclear, and they are rarely used by practitioners.

In most cases, one is interested in an alignment only if a "good" alignment exists, i.e., if the sequences exhibit homology. Consider two sequences of equal length. The ideal score for aligning these sequences occurs when the sequences are identical, yielding a match in every position of the alignment. The quality of an

alignment can be measured by its score as a percentage of the ideal score. For simplicity, assume that each match is rewarded by the same score, and assume also a constant gap penalty function, where each gap position is penalized by the same amount. Consider a band in the dynamic programming table around the main diagonal consisting of k diagonals above and below it. Any solution that crosses this *k-band* must have at least $k + 1$ gaps in either sequence and has to miss at least that many matches, limiting the maximum score possible to no more than $(1 - \frac{3k-1}{n})$ fraction of the ideal score. If we are interested in an optimal alignment only if the score is above a certain threshold percentage, the threshold can be used to compute the value of k. For example, if 90% is desired, k can be chosen to be approximately $\frac{n}{30}$. The search space can then be limited to a band of this size without loss of optimal solution if its score is higher than the threshold. The run-time is reduced to $O(kn)$. Such limits can be established for more elaborate scoring schemes and for aligning sequences of different lengths. For a more clever scheme that operates in $O(kn)$ time without a predetermined threshold by using trial and error on k with exponentially doubling values, see [29].

Local Alignments

Given two sequences A and B, the local alignment problem is to find a subsequence of A and a corresponding subsequence of B that exhibit significant homology. Algorithmically, it makes sense to study global alignments first, but the types of alignments used predominantly by biologists are local alignments. Normally, a homologous sequence that is being sought may be a part of a larger DNA sequence, in which case local alignment must be used. In addition, there are several problems that require local alignments. For instance, conserved motifs in protein sequences often indicate structural similarity and functional similarity. When comparing two versions of a conserved gene across species, it is sufficient for the parts of the gene (*exons*) that code for protein to be homologous.

The local alignment problem is computationally modeled as follows: A local alignment between A and B is a global alignment between a subsequence of A and a subsequence of B, scored as presented before. The local alignment problem between sequences A and B is to find an alignment between a subsequence of A and a subsequence of B that results in the highest possible score over all such possible alignments and subsequences [87, 89]. As in the case of global alignments, this problem can be solved using dynamic programming. As before, tables T_1, T_2, and T_3 are created, but with the following difference: An entry $[i, j]$ in each table is used to store the highest score of an alignment between a suffix of $a_1\, a_2 \ldots a_i$ and a suffix of $b_1\, b_2 \ldots b_j$, with the same restriction on matching a_i and b_j as before. Alignment of empty suffixes is valid in T_1, and is assigned a score of 0. Therefore, Equation (1) is modified in the following way, and Equations (2) and (3) remain the same.

$$T_1[i,j] = f(a_i, b_j) + \max \begin{cases} T_1[i-1, j-1] \\ T_2[i-1, j-1] \\ T_3[i-1, j-1] \\ 0 \end{cases}$$

The maximum score in T_1 is the optimal score. An optimal alignment is retrieved by performing a traceback from a maximum entry in T_1 until a score of 0 is reached. Thus, the local alignment problem can also be solved in $O(mn)$ time.

It is difficult to apply Hirschberg's method to this algorithm to achieve a reduction in space. The following technique was invented by Huang [45] to facilitate space reduction. The algorithm consists of three steps:

1. Compute the optimal local alignment score as before, but only keep track of a largest entry and its position ($[i, j]$) as the tables are filled. Rows are discarded as soon as they are used in computing other rows to save space. This identifies the end of an optimal local alignment.
2. Run a global alignment algorithm on $a_i a_{i-1} \ldots a_1$ and $b_j b_{j-1} \ldots b_1$ to locate a largest entry, which corresponds to the beginning ($[k, l]$) of a subsequence alignment that ends at $[i, j]$. Once again, discard rows as soon as they are no longer needed and remember a largest entry seen so far. This identifies the beginning of an optimal local alignment whose end is discovered in step (1).
3. Run Hirschberg's space-saving global alignment algorithm between subsequences $a_k a_{k+1} \ldots a_i$ and $b_l b_{l+1} \ldots b_j$.

Alignments are one of the most thoroughly studied problem areas in computational biology, dating back to the 1970 introduction of the first global alignment algorithm by Needleman and Wunsch in the context of protein sequence homology [77]. A further study of sequence alignment algorithms can be conducted by referring to the classic text of Sankoff and Kruskal [85]. A good portion of several recent texts in computational biology are devoted to the study of alignment algorithms [22, 37, 71, 81, 87, 96]. Nevertheless, alignments continue to be an active area of research. An important problem area that is not covered here is that of multiple sequence alignments. While pairwise sequence alignments are a fundamental tool used in many computational genomics applications, multiple sequence alignments of a family of related proteins to infer conserved motifs is perhaps the most prevalent direct use of alignments by molecular biologists. Even within pairwise alignments, there are a number of more complex problems, including spliced alignments [66], syntenic alignments [47], and DNA–protein alignments [34, 49, 59, 99]. The sensitivity of an optimal alignment to the particular choice of parameter values used is studied as parametric sequence alignments [27, 38, 78]. To enable fast pairwise alignments for very large sequences, parallel methods have been developed [7, 23, 44, 62, 83].

3.2 Exact Matches

Another standard tool used in computational genomics applications is the identification of exact matching substrings between sequences. Due to evolutionary mechanisms that alter biomolecular sequences and errors introduced by experimental processes, one is rarely interested in exact matches as an end in themselves. Exact matches play a role because they are typically fast—requiring linear time as opposed to the quadratic time of alignment algorithms. As an example, consider the task of finding good local alignments between a query sequence and a database consisting of hundreds of thousands of sequences. It is computationally expensive to do as many pairwise local alignments. If we are interested in a pairwise alignment only if it exhibits significant homology, such an alignment should

also contain regions of exact matches. For instance, if an aligning region of 100 *bp* length contains at most four positions of difference, there should be at least an exact match of length 20 in this region. Exact matches can be used as a filter to eliminate large number of pairs that would not yield a good local alignment by performing alignments only on pairs that have an exact matching region larger than a determined threshold. It is in this spirit that many problems related to exact matches find applications in bioinformatics. Below, we provide a brief introduction to three data structures frequently used in computational genomics.

Lookup Tables

A lookup table is a simple data structure that keeps track of the positions of occurrences of substrings of a prespecified length in one or more strings. Lookup tables are used in a number of important bioinformatic tools, including such popular programs as BLAST [4, 5] for database searches and CAP3 [48] for genome assembly.

Let Σ denote the alphabet, and let w denote a prespecified length. The lookup table is an array LT of size $|\Sigma|^w$, corresponding to the $|\Sigma|^w$ possible substrings of length w. Let $f : \Sigma \rightarrow \{0, 1, \ldots |\Sigma| - 1\}$ be the one-to-one function such that $f(c) = j - 1$ if c is the j^{th} lexicographically smallest character. For the purpose of the lookup table, any arbitrary ordering of the characters can be taken as lexicographic ordering. Using f, a substring of length w can be treated as a w-digit number in a base $|\Sigma|$ system and converted to its decimal equivalent. We use the notation $F(\alpha)$ to denote the decimal number corresponding to a w-long substring α.

Each entry in the lookup table LT points to a linked list of specific locations within the input set of strings where the substring corresponding to the index for the entry occurs. Let s be a string of length n. It is easy to construct the lookup table for s in $O\left(|\Sigma|^w + n\right)$ time. First, create and initialize each entry to a null list in $O\left(|\Sigma|^w\right)$ time. Then insert substrings one at a time. First compute *index* = $F(s[1..w])$ in $O(w)$ time. Insert the position 1 in the linked list corresponding to $LT[index]$. Using the identity

$$F(s[k+1 \ldots k+w+1]) = (F(s[k \ldots k+w] - f(s[k])|\Sigma|^{w-1}) \times |\Sigma| + f(s_{k+w+1})$$

$F(s[k+1..k+w+1])$ can be computed from $F(s[k..k+w])$ in $O(1)$ time. Since each starting position $1...n - w + 1$ occurs in a linked list, the total size of all linked lists is $O(n)$ (typically $n >> w$). Thus, the size of the lookup table data structure is $O\left(|\Sigma|^w + n\right)$. The lookup table can be easily generalized to a set of strings. Let $S = \{s_1, s_2, \ldots, s_k\}$ be a set of k strings of total length N. To create the corresponding lookup table, substrings from each of the strings are inserted in turn. A location in a linked list now consists of a pair giving the string number and the position of the substring within the string. The space and run-time required for constructing the lookup table is $O\left(|\Sigma|^w + N\right)$.

A lookup table is conceptually a very simple data structure to understand and implement. Once the lookup table for a database of strings is available, given a query string of length w, all occurrences of it in the database can be retrieved in

$O(w + k)$ time, where k is the number of occurrences. The main problem with this data structure is its dependence on an arbitrary predefined substring of length w. If the query string is of length $l > w$, the lookup table does not provide an efficient way of retrieving all occurrences of the query string in the database. Nevertheless, lookup tables are widely used in bioinformatics due to their simplicity and ease of use.

Suffix Trees and Suffix Arrays

Suffix trees and suffix arrays are versatile data structures fundamental to string processing applications. Let s' denote a string over the alphabet Σ. Let $\$ \notin \Sigma$ be a unique termination character, and $s = s' \$$ be the string resulting from appending \$ to s'. We use the following notation: $|s|$ denotes the size of s, $s[i]$ denotes the i^{th} character of s, and $s[i..j]$ denotes the substring $s[i]s[i + 1] \ldots s[j]$. Let $suff_i = s[i]s[i + 1] \ldots s[|s|]$ be the suffix of s starting at i^{th} position.

The suffix tree of s, denoted $ST(s)$ or simply ST, is a compacted tree of all suffixes of string s. Let $|s| = n$. It has the following properties:

1. The tree has n leaves, labeled 1 ... n, one corresponding to each suffix of s.
2. Each internal node has at least two children.
3. Each edge in the tree is labeled with a substring of s.
4. The concatenation of edge labels from the root to the leaf labeled i is $suff_i$.
5. The labels of the edges connecting a node with its children start with different characters.

The paths from the root to the leaves corresponding to the suffixes $suff_i$ and $suff_j$ coincide up to their longest common prefix, at which point they bifurcate. If a suffix of the string is a prefix of another, longer suffix, the shorter suffix must end in an internal node instead of a leaf, as desired. It is to avoid this possibility that the unique termination character is added to the end of the string. Keeping this in mind, we use the notation $ST(s')$ to denote the suffix tree of the string obtained by appending \$ to s'. Throughout this chapter, "\$" is taken to be the lexicographically smallest character.

Since each internal node has at least two children, an n-leaf suffix tree has at most $n - 1$ internal nodes. Because of property (5), the maximum number of children per node is bounded by $|\Sigma| + 1$. Except for the edge labels, the size of the tree is $O(n)$. In order to allow a linear space representation of the tree, each edge label is represented by a pair of integers denoting the starting and ending positions, respectively, of the substring describing the edge label. If the edge lable corresponds to a repeat substring, the indices corresponding to any occurrence of the substring may be used. The suffix tree of the string *mississippi* is shown in Figure 21.2. For convenience of understanding, we show the actual edge labels.

Let v be a node in the suffix tree. Let *path-label*(v) denote the concatenation of edge labels along the path from root to node v. Let *string-depth*(v) denote the length of *path-label*(v). To differentiate this with the usual notion of depth, we use the term *tree-depth* of a node to denote the number of edges on the path from root to the node. Note that the length of the longest common prefix between two suffixes is the string depth of the lowest common ancestor of the leaf nodes corresponding to the suffixes. A repeat substring of string S is *right-maximal* if there are two occurrences of the substring that are succeeded by different characters in

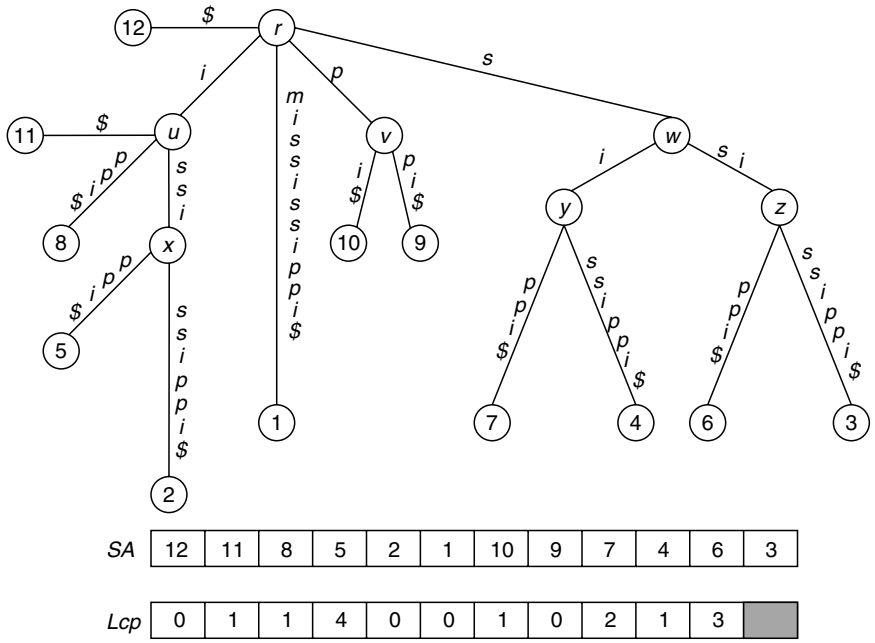

Figure 21.2. Suffix tree, suffix array, and *Lcp* array of the string *mississippi*. The suffix links in the tree are given by $x \rightarrow z \rightarrow y \rightarrow u \rightarrow r$, $v \rightarrow r$, and $w \rightarrow r$.

the string. The path label of each internal node in the suffix tree corresponds to a right-maximal repeat substring and vice versa.

Suffix trees can be generalized to multiple strings. The generalized suffix tree of a set of strings $S = \{s_1, s_2, ..., s_k\}$, denoted $GST(S)$ or simply GST, is a compacted tree of all suffixes of each string in S. We assume that the unique termination character \$ is appended to the end of each string. A leaf label now consists of a pair of integers (i, j), where i denotes the suffix from string s_i and j denotes the starting position of the suffix in s_i. Similarly, an edge label in a GST is a substring of one of the strings. It is represented by a triplet of integers (i, j, l), where i denotes the string number and j and l denote the starting and ending positions of the substring in s_i, respectively. For convenience of understanding, we will continue to show the actual edge labels. Note that two strings may have identical suffixes. This situation is compensated for by allowing leaves in the tree to have multiple labels. If a leaf is multiply labeled, each suffix should come from a different string. If N is the total number of characters of all strings in S, the GST has at most N leaf nodes and takes up $O(N)$ space.

Suffix trees are useful in solving many problems involving exact matching in optimal run-time bounds. Moreover, in many cases, the algorithms are very simple to design and understand. For example, consider the problem of determining if a pattern P occurs in text T over a constant-sized alphabet. Note that if P occurs starting from position i in T, then P is a prefix of $suff_i$ in T. Thus, P occurs in T if and only if P matches an initial part of a path from root to a leaf in $ST(T)$.

Traversing from the root matching characters in P, this can be determined in $O(|P|)$ time, independent of T's length. As another application, consider the problem of finding a longest common substring of a pair of strings. Once the *GST* of the two strings is constructed, the path-label of an internal node with the largest string depth that contains at least one leaf from each string is the answer.

Suffix trees were invented by Weiner [97], who also presented a linear time algorithm to construct them for a constant-sized alphabet. A more space-economical linear-time construction algorithm is given by McCreight [69], and a linear-time online construction algorithm was invented by Ukkonen [94]. A unified view of these three suffix tree construction algorithms can be found in [33]. Farach [25] presented the first linear-time algorithm for strings over integer alphabets. The construction complexity for various types of alphabets is explored in [26].

The space requirement of suffix trees is a cause for concern in many large-scale applications. Manber and Myers [67] introduced suffix arrays as a space-efficient alternative to suffix trees. The suffix array of a string $s = s'\$$, denoted $SA(s)$ or simply SA, is a lexicographically sorted array of all suffixes of s. Each suffix is represented by its starting position in s. $SA[i] = j$ iff $suff_j$ is the i^{th} lexicographically smallest suffix of s. The suffix array is often used in conjunction with an array termed the *Lcp* array, : $Lcp[i]$ contains the length of the longest common prefix between the suffix in $SA[i]$ and $SA[i + 1]$. The suffix and *Lcp* arrays of the string *mississippi* are shown in Figure 21.2. Suffix arrays can also be generalized to multiple strings to contain the sorted order of all suffixes of a set of strings (see Figure 21.3). For linear-time suffix array construction algorithms, see [55, 58]. Techniques for using suffix arrays as a substitute for suffix trees can be found in [1]. Further space reduction can be achieved by the use of compressed suffix trees and suffix arrays and other data structures [28, 36].

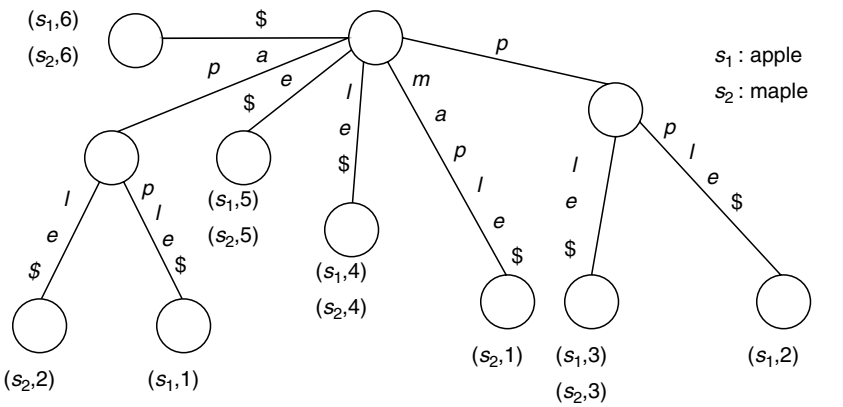

Figure 21.3. Generalized suffix tree and generalized suffix array of strings *apple* and *maple*.

4 APPLICATIONS

4.1 Database Search

Perhaps the most frequently used bioinformatic application is that of searching a database of sequences for homology to a given query sequence. When a new sequence is obtained in the laboratory, the first step in understanding the sequence is often to carry out a database search against as many known sequences as possible. This is important because sequence homology very often translates into functional similarity and immediately provides a way of understanding the newly obtained sequence. To facilitate such searches, large database collections have been developed that include various types of DNA and protein sequences ranging from gene and protein sequences to entire genomes. The most comprehensive such repository in the United States is maintained at the National Center for Biotechnology Information (NCBI; part of National Institutes of Health (NIH)). They have also developed a suite of database search programs commonly known as BLAST, the Basic Local Alignment Search Tool [4, 5]. This collection of programs constitutes the bioinformatic tools most commonly used by molecular biologists. These programs are engineered for speed and employ complex statistics to determine and output the statistical relevance of the generated alignments. The underlying algorithms are not necessarily perfect or optimal. Although we make some BLAST-specific references in some cases, the treatment provided here is more in terms of the issues involved in developing a search program and some computational ways of addressing them.

In principle, a local alignment query can be answered by running a local alignment algorithm on the query sequence and on each of the sequences in the database. The total run-time of such a naive algorithm is proportional to the product of the query sequence length and database size, and is clearly prohibitive and wasteful. Significant savings are realized by first focusing on identifying database sequences that share short exact matching regions with the query, and then processing only such sequences further. We begin with a brief description of the scoring methods employed in practice.

Scoring Schemes

The alignment scoring schemes presented so far relied on simple reward or penalty based on match or mismatch, respectively. In practice, elaborate scoring schemes are constructed to reflect the realities of evolutionary manipulations or functional similarities which the generated alignments are expected to capture. These are developed in the context of aligning protein sequences [3] and are applied to DNA sequence comparison in [90]. Recall that there are twenty different amino acid residues constituting protein sequences. Rather than use a simple match score and mismatch penalty, a symmetric 20×20 matrix is defined to capture the appropriate scores for every possible pair of amino acid residues that may be part of an alignment. The rationale for individualized scores come from preferential substitution of certain types of amino acids. For instance, six of the twenty amino acids are hydrophobic, and substituting one for another is likely to still preserve the protein

function. Thus, a high score is awarded for such a substitution, in contrast to when a hydrophilic amino acid is substituted for a hydrophobic one. The most commonly used scoring matrices are the PAM [20] and BLOSUM [42] matrices.

Dayhoff introduced the notion of Percent Accepted Mutation (PAM) to quantify evolutionary changes within protein sequences. A PAM unit is the amount of evolution that will, on average, change 1% of the amino acids within a protein sequence. A 20×20 transition probability matrix M is defined such that $M[i, j]$ captures the probability of amino acid i changing to amino acid j within 1 PAM evolutionary distance. Longer evolutionary distance probabilities can be determined by computing an appropriate exponent of the matrix—M^{100} gives the matrix for 100 units etc. The score for a PAMk matrix is defined by PAM$k[i, j]$ $= 10 \log \frac{M^k[i,j]}{p_j}$, where p_j is the probability of random occurrence of amino acid j. Smaller evolutionary distances are used for finding short, strong local alignments, while longer evolutionary distances are used for detecting weak, long spanning alignments.

The BLOSUM matrices, short for Block Substitution Matrices, are constructed based on local multiple sequence alignments of protein sequences from a related family. BLOSUM matrices are based on the minimum percentage identity of the aligned protein sequences used in deriving them – for instance, the standard BLOSUM62 matrix corresponds to alignments exhibiting at least 62% identity. Thus, larger numbered matrices are used to align closely related sequences and smaller numbered matrices are used for more distantly related ones. Scores in a BLOSUM matrix are log-odds scores measuring the logarithm of the ratio of the likelihood of two amino acids appearing in an alignment on purpose and the likelihood of the two amino acids appearing by chance. A score in a BLOSUM matrix B is defined as

$$B[i,j] = \frac{1}{\lambda} \log \frac{p_{ij}}{f_i f_j}$$

where p_{ij} is the probability of finding amino acids i and j aligned in a homologous alignment, and f_i and f_j denote the probability of occurrence of i and j, respectively, in protein sequences. The scaling factor λ is used for convenience to generate scores that can be conveniently rounded off to integers. The BLOSUM62 matrix is the default matrix used by the BLAST suite of programs, and is shown in Table 21.2.

As for gaps, affine gap penalty functions are used because insertion or deletion of a subsequence is evolutionarily more likely than several consecutive individual base mutations. By using matrices and gap penalty functions derived either directly, based on evolutionary processes, or indirectly, based on knowledge of what types of alignments are biologically satisfactory, it is expected that the alignments generated using computer algorithms reflect biological reality.

Finding exact matches

The first step in query processing is to find database sequences that share an exact matching subsequence with the query sequence. For convenience, the programs designed often look for matches of a fixed length. For instance, BLAST uses a length of 11 for DNA alignments and a length of 3 for protein alignments. When

Table 21.2. The BLOSUM62 matrix.

	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

such an exact length is used (say, w), a lookup table provides a natural and convenient way to find the matches. The lookup table is constructed to index all substrings of length w occurring in database sequences and stored a priori. When a query is issued, all w-length substrings of it are extracted. For each substring, the lookup table entry indexed by it immediately points to the database sequences that contain this substring. Note that in a purely random sequence, the chance of finding a DNA sequence of length 11 is 1 in $4^{11} > 4 \times 10^6$, and the chance of finding an amino acid sequence of length 3 is 1 in $20^3 = 8{,}000$. In the protein-to-protein version of BLAST, the program does not look for exact matches but rather identifies substrings that may be different than the query substring as long as their nongapped alignment has a score above a specified threshold, as per the amino acid substitution matrix used. In addition, BLAST avoids certain substrings that are known to occur commonly, but the user has the option to request that this filter be turned off.

Generating Alignments

Once a subset of database sequences is identified using exact matches as above, an alignment of each sequence with the query sequence is generated and displayed. This can be done by using full-scale dynamic programming, as explained earlier. In practice, the alignment is obtained by anchoring it at the matching region found and extending it in either direction. As a practical heuristic, the alignment is not continued so as to explore all options, but is stopped once the score falls off a certain amount from the peak score seen.

4.2 Genome Sequencing

Genome sequencing refers to the deciphering of the exact order of nucleotides that make up the genome of an organism. Since the genome contains all the genes

of the organism along with promoter and enhancer sequences that play critical roles in the expression and amount of expression of genes, the genome of an organism servers as a blueprint for what constitutes the species itself. Knowledge of the genome sequence serves as a starting point for many exciting research challenges – determining genes and their locations and structures, understanding genes involved in complex traits and genetically inherited diseases, gene regulation studies, genome organization and chromosomal structure and organization studies, finding evolutionary conservation and genome evolution mechanisms, etc. Such fundamental understanding can lead to high-impact applied research in genetically engineering plants to produce desirable traits and in designing drugs targeting genes whose malfunction causes diseases. Over the past one and a half decades, concerted research efforts and significant financial resources directed towards genome sequencing have led to the sequencing of many genomes, starting from the *Haemophilus influenzae* genome sequenced in 1995 [30] to the more recent sequencing of the complex human and mouse genomes [16, 17, 95]. At present, the complete genomes of over 1,000 viruses and over 100 microbes are known. *Arabidopsis thaliana* is the first plant genome to be sequenced, rice genome sequencing is at an advanced stage, and sequencing of the maize genome is currently under way.

The basic underlying technology facilitating genome sequencing is the DNA sequencing methodology developed by Sanger et al. [84]. This method allows laboratory sequencing of a DNA molecule of length about 500 *bp*. However, even bacterial genomes are about 3 to 4 order of magnitudes larger, and the genomes of higher organisms such as human and mouse are about 7 orders of magnitude larger. To sequence a large target DNA molecule, a sufficient number of smaller overlapping fragments of sequenceable size are derived from it and independently sequenced. Once the fragments are sequenced, overlaps are used to computationally assemble the target DNA sequence. This process is called the *shotgun sequencing* approach, and the corresponding computational problem is called *fragment assembly*. In the whole-genome shotgun approach, the target DNA is the entire genome itself. Another alternative is to partition the genome into large DNA sequences of a size on the order of 100, 000 *bp* whose locations along the genome are known from techniques such as physical mapping. Each of these large DNA sequences is then deciphered using the shotgun sequencing approach. While whole-genome shotgun sequencing is quicker, it is computationally more challenging to perform whole-genome assembly than to assemble a few hundred thousand base-pair-long target sequences. However, such whole-genome shotgun assemblies have been carried out for the human and the mouse. For the remainder of this section, we focus on the fragment assembly problem.

FRAGMENT ASSEMBLY

Consider a target DNA sequence to be assembled using the shotgun sequencing approach. We assume that a large number of copies of the same target sequence are available (either as samples or via cloning methods using bacterial artificial chromosomes). Copies of the target sequence are sheared in segments of a defined length and cloned into a plasmid vector for sequencing. Plasmids are

circular DNA molecules that contain genes conferring antibiotic resistance and a pair of promoter sequences flanking a site where a DNA sequence can be inserted for replication. The sheared segments are inserted into plasmids and injected into bacteria, a process that allows them to be replicated along with bacteria. The bacteria can be killed using an antibiotic to extract copies of the inserts for sequencing. The inserts are typically a few thousand base pairs long, and about 500 *bp* from each end can be sequenced using Sanger's method. This not only gives two fragments from random locations of the target DNA sequence but also gives the approximate distance between their locations, since the size of the insert can be determined. These distances, known as *forward-reverse constraints* because the two fragments will be on different strands of the genome [48], are crucial in ensuring correct assembly.

Early work on developing the foundations of fragments assembly was carried out by Lander, Myers, Waterman, and others [57, 63, 73, 74]. A number of fragment assembly programs were developed [15, 35, 46, 48, 49, 79, 91]. Based on the experiences gained from these efforts, a new generation of assembly programs have recently been developed for handling whole-genome shotgun sequencing [12, 41, 47, 50, 72, 75]. The discussion provided here is not meant to represent any one particular program but rather is intended to give highlights of the issues involved in fragment assembly and some algorithmic means of handling them.

The primary information available to assemble fragments is the overlap between fragments that span intersecting intervals of the target sequence. DNA sequencing is not error free, but the error rates are quite tolerable, with high-quality sequencing averaging under 1%. Also, the error rates tend to be higher at either end of the sequenced fragment. If the target DNA sequence is not unique—for example, if genomes of several individuals are sampled for diversity—then there are naturally occurring variations that show up in fragments as well. Due to the presence of experimental errors and other differences, potential overlaps between fragments must be investigated using alignment algorithms. It is computationally infeasible to run them on every pair of fragments in a reasonable time frame. On the other hand, the differences are small enough that a good alignment should have exact matching regions. Thus, pairs of fragments that have sufficiently long exact matches are identified, and alignments are carried out only on such pairs. We term these pairs as *promising pairs*. Genomes are known to contain repeats – these range from a large number of copies of short repeating sequences to repeats or tandem repeats of genes that are present in multiple copies to boost the production of the corresponding protein. Repeats mislead assembly software since fragments coming from different parts of the genome may overlap. This is where forward-reverse constraints are useful. In the following, we describe in more detail the computational aspects of shotgun assembly.

Determining the number of fragments

Let $|G|$ denote the length of the target DNA sequence G, and let l denote the average size of a fragment. Let n denote the number of fragments to be derived. The coverage ratio x implied by this sampling is defined as $x = \frac{nl}{|G|}$. Intuitively, the coverage should be sufficient so that overlaps between fragments provide enough

information for assembly. It is not possible to guarantee that the fragments will provide complete coverage of G. In that case, the fragment assembly program is expected to generate a number of *contigs* (contiguous subsequences) corresponding to the disjoint regions of G that can be deciphered from the fragments. Hence, *fragment assembly* is also known as *contig assembly*. Under the assumption that the starting position in G corresponding to a fragment is uniformly distributed over the length of G, the expected number of contigs and the fraction of G covered by the contigs can be estimated [63]. A coverage of 4.6 is enough to cover 99% of G, and a coverage of 6.9 is sufficient to cover 99.9% of G. As an example, the mouse genome shotgun sequence data consists of 33 million fragments. Assuming an average fragment length of 500 and a 3 billion *bp* genome, the coverage ratio works out to be 5.5. To quickly sequence such a large number of fragments, several high-throughput sequencing machines are typically used.

Finding promising pairs

In a random shotgun sequencing approach using a constant coverage factor (approximately 5–7 in practice), it is easy to see that the number of overlapping pairs of sequences is linear in n, provided that repeats do not have an overwhelming presence in the genome. Thus, identifying promising pairs based on exact matches potentially reduces the number of pairwise alignments from $O(n^2)$ to $O(n)$. Most assembly software programs use the lookup table data structure to identify pairs. First, a lookup table is constructed for all the input fragments and their complementary strands using a fixed substring length w. Each entry in the lookup table points to a list of fragments that contain a fixed w-long substring indexing the entry. Thus, every pair of fragments drawn from this list shares a w-long substring. Once a pair is identified, the detected w-long match is extended in either direction to uncover a maximal common substring. Some programs further extend the matching region by allowing a small number of errors. One problem with the lookup table is that a pair of fragments having a maximal common substring of $l > w$ bases will have $(l - w + 1)$ common substrings of length w within that region. This may cause multiple considerations of the same pair based on the same region, and it is important to find ways of avoiding this possibility. A more elegant strategy using suffix trees will be outlined later.

Aligning promising pairs

Each promising pair is aligned using a pairwise alignment algorithm. Two types of alignments – containments and suffix–prefix overlaps – are of interest, as shown in Figure 21.4. It is typical to reduce alignment time by first anchoring the alignment based on the already found matching region, and extending the alignment at both ends using banded dynamic programming. If one fragment is contained in the other, the shorter fragment need not be used for further overlap computation or in determining which other fragments will be in the same contig. It is, however, used in determining the contig sequence. Based on the alignment score, a measure of overlap strength can be associated with each aligned pair.

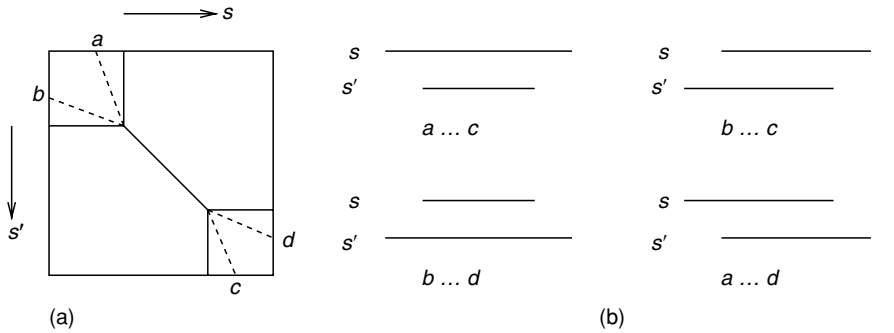

Figure 21.4. The figure shows the pairwise alignment strategy of extending a common substring match at both ends. Also shown are the four types of alignments of interest and their corresponding optimal paths in the dynamic programming table.

Creating contig layouts

This step consists of identifying all the fragments that determine a contig and the layout of these fragments with respect to the contig. One way to do this is to employ a greedy heuristic and consider all good overlaps in decreasing order of overlap strength. The next pair is added to the layout if it does not conflict with the layout determined so far. Forward-reverse constraints can be used to resolve conflicts as well. Another way to address contig layout is to use a graph model, with nodes representing fragments and edges representing good overlaps. Each connected component can then be a contig.

Assembly of contigs

Once the layout of each contig is determined, the exact sequence of the contig is computed by using alignments on the layout. Ideally, one would want a multiple sequence alignment of all the overlapping sequences, but this is time consuming. The pairwise alignments computed earlier can be used to draw the layout, and a simple scheme such as majority voting can be used to determine the base at each location of the contig.

Generating Scaffolds

A scaffold is an ordered collection of one or more contigs. Such an order between contigs can be determined with the help of any available forward-reverse constraints between a pair of fragments, one from each contig. This process allows ordering of some contigs, although there are no overlapping fragments connecting them, and also allows the determination of the approximate distance between the contigs. It is possible to use targeted techniques to fill the gaps later.

The above description is meant to be a generic description of the various phases in genome assembly and the computational challenges in each phase. Clearly, the diverse available genome assembly programs employ different strategies. A modular open-source assembler is currently being developed by the AMOS consortium (http://www.cs.jhu.edu/~genomics/AMOS).

4.3 Expressed Sequence Tag Clustering

Expressed Sequence Tags (ESTs) are DNA sequences experimentally derived from expressed portions of genes. They are obtained as follows: A cell mechanism makes a copy of a gene as an RNA molecule, called the *premessenger RNA*, or pre-mRNA for short. Genes are composed of alternating segments called *exons* and *introns*. The introns are spliced out from the pre-mRNA, and the resulting molecule is called *mRNA*. The mRNA essentially contains the coded recipe for manufacturing the corresponding protein. Molecular biologists collect mRNA samples and, using them as templates, synthetically manufacture DNA molecules. These are known as *complementary DNA molecules*, or *cDNA*s for short. Due to the limitations of the experimental processes involved and due to breakage of sequences in chemical reactions, several cDNAs of various lengths are obtained instead of just full-length cDNAs. Part of the cDNA fragments, of average length about 500–600 *bp*, can be sequenced with Sanger's method. The sequencing can be done from either end. The resulting sequences are called *ESTs* (Expressed Sequence Tags). For a simplified diagrammatic illustration, see Figure 21.5.

It is important to note that the genes sampled by ESTs and the frequency of sampling depend on the expression levels of the various genes. The EST clustering problem is to partition the ESTs according to the (unknown) gene source they come from. This process is useful in several ways, some of which are outlined below:

- *Gene Identification:* Genome sequencing is only a step towards the goal of identifying genes and finding the functions of the corresponding proteins. ESTs provide the necessary clues for gene identification.
- *Gene Expression Studies:* In EST sequencing, genes that are expressed more will result in more ESTs. Thus, the number of ESTs in a cluster indicates the level of expression of the corresponding gene.
- *Differential Gene Expression:* ESTs collected from various organelles of an organism (such as the leaf, root, and shoot of a plant) reveal the expression levels of genes in the respective organelles and provide clues to their possible function.
- *SNP Identification:* The same gene is present in slight variations, known as *alleles*, among different members of the same species. Many of these alleles differ in a single nucleotide, and some of these differences are the cause of genetic diseases. ESTs from multiple members of a species help identify such disease-causing single nucleotide polymorphisms, or SNPs.

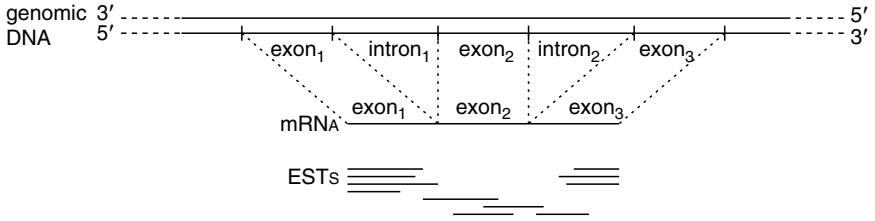

Figure 21.5. A simplified diagrammatic illustration of genomic DNA, mRNA, and ESTs.

- *Design of Microarrays:* Microarrays, also called DNA chips, are a recent discovery allowing gene expression studies of thousands of genes simultaneously. ESTs can be used in designing microarrays to detect the level of expression of the corresponding genes.

EST clustering is an actively pursued problem of current interest [18, 32, 39, 53, 60, 61, 64, 70, 82, 98]. ESTs are fairly inexpensive to collect and represent a major source of DNA sequence information currently available. A repository of ESTs collected from various organisms is maintained at the National Center for Biotechnology Information (http://www.nicb.nlm.nih.gov/dbEST).

If the genome of the organism is available and small, the individuals ESTs can be directly aligned with the genome to determine clustering. However, this situation is rarely the case. As with fragment assembly, the potential overlaps between ESTs from the same gene provide the primary information available for EST clustering. For this reason, fragment assembly software is often used for EST clustering, though there are some subtle differences between the two problems that need to be carefully addressed. One important difference is that ESTs do not sample the gene space uniformly at random, but rather the sampling rate is proportional to the gene expression. It is quite common to have a few very large clusters containing as many as 10% of the input ESTs and to have thousands of single EST clusters. Because of this nonuniform sampling, the number of overlapping pairs can be as high as $\Omega(n^2)$ and are observed to be such in practice. This overlap considerably slows down standard lookup table-based fragment assembly software when applied to EST clustering problems. Furthermore, the space required to store potential overlapping pairs or promising pairs is quadratic, limiting the effectiveness of the software to much smaller data sets.

A suffix tree-based solution can be designed to address these problems [52], and such a solution could be important even for genome assembly when the sampling is purposefully nonuniform [24]. The basic idea is to build a GST of all ESTs and their complementary strands. Common substrings between ESTs can be identified by shared paths in the suffix tree. However, the power of this method lies in directly identifying maximal common substrings and avoiding the generation of pairs based on parts of maximal common substrings. Moreover, the pairs can be generated in nonascending order of the maximal common substring length on an as-needed basis, without having to store any pairs generated so far. This approach will reduce the memory required from quadratic to linear. Below we outline the various steps in EST clustering based on this strategy.

ON-DEMAND PAIR GENERATION

Let the term *promising pair* refer to a pair of strings that have a maximal common substring of length at least equal to a threshold value ψ. The goal of the on-demand pair generation algorithm is to report promising pairs on-the-fly, in the nonincreasing order of maximal common substring length. A pair is generated as many times as the number of maximal substrings common to the pair. The algorithm operates on the following idea: If two strings share a maximal common substring α, then the leaves corresponding to the suffixes of the strings starting

with α will be present in the subtree of the node with path-label α. Thus the algorithm can generate the pair at that node.

A substring α of a string is said to be *left-extensible* (alternatively, *right-extensible*) by character c if c is the character to the left (alternatively, right) of α in the string. If the substring is a prefix of the string, then it is said to be left-extensible by λ, the null character. Let *leaf-set*(v) denote the suffixes in the subtree under v. Based on the characters immediately preceding these suffixes, they are partition into five sets, $l_A(v)$, $l_C(v)$, $l_G(v)$, $l_T(v)$ and $l_\lambda(v)$, collectively referred to as *lsets*(v). The algorithm for generation of pairs is given in Figure 21.6. The nodes in GST with string-depth $\geq \psi$ are sorted in nonincreasing order of string depth and are processed in that order. The *lsets* at leaf nodes are computed directly from the leaf labels. The set of pairs generated at node v is denoted by P_v. If v is a leaf, a cartesian product of each of the *lsets* at v corresponding to A, C, G, T, λ with every other *lset* of v corresponding to a different character is computed. In addition, a cartesian product of $l_\lambda(v)$ with itself is computed. The union of these cartesian products is taken to be P_v. If v is an internal node, a cartesian product of each *lset* corresponding to A, C, G, T, λ of each child of v with every other *lset* corresponding to a different character in every other child node is computed. In addition, a cartesian product of the *lset* corresponding to λ of each child node with each of the *lsets* corresponding to λ of every other child node is computed.

Algorithm 1 *Pair Generation*

GeneratePairs

1. Compute the string-depth of all nodes in the GST.
2. Sort nodes with string-depth $\geq \psi$ in non-increasing order of string-depth.
3. For each node v in that order
   IF v is a leaf THEN
   ProcessLeaf (v)
   ELSE
   ProcessInternalNode(v)

ProcessLeaf(Leaf: v**)**

1. Compute

$$P_v = \cup_{(c_i, c_j)} l_{c_i}(v) \times l_{c_j}(v), \forall\, (c_i, c_j) \text{ s.t., } c_i < c_j \text{ or } c_i = c_j = \lambda$$

ProcessInternalNode(Internal Node: v**)**

1. Compute

$$P_v = \cup_{(u_k, u_l)} \cup_{(c_i, c_j)} l_{c_i}(u_k) \times l_{c_j}(u_l), \forall\, (u_k, u_l), \forall\, (c_i, c_j) \text{ s.t., } 1 \leq k\ l \leq m, c_i \neq c_j \text{ or } c_i = c_j = \lambda$$

2. Create all *lsets* at v by computing:
   For each $c_i \in \sum \cup \{\lambda\}$ do

$$l_{c_i}(v) = \cup_{u_k} l_{c_i}(u_k), 1 \leq k \leq m$$

Figure 21.6. Algorithm for generation of promising pairs.

The union of these cartesian products is taken to be P_v. The *lset* for a particular character at v is obtaining by taking a union of the *lsets* for the same character at the children of v.

A pair generated at a node v is discarded if the string corresponding to the smaller EST *id* number is in complemented form. This avoids duplicates such as generating both (e_i, e_j) and ($ \bar e_i, \bar e_j $) or generating both ($e_i$, $\bar e_j$) and ($\bar e_i, e_j$) for some $1 \le i, j \le n$. Thus, without loss of generality, we denote a pair by (s, s'), where $s = e_i$ and s' is either e_j or $\bar e_j$ for some $i < j$. The relative orderings of the characters in $\sum \cup \{\lambda\}$ and the child nodes avoid generation of both (s, s') and (s', s) at the same node.

In summary, if v is a leaf,

$$P_v = \{(s, s') \mid s \in l_{c_i}(v), s' \in lc_j(v), c_i, c_j \in \sum \cup \{\lambda\}, ((c_i < c_j) \vee (c_i = c_j = \lambda))\}$$

and if v is an internal node,

$$P_v = (s, s') \Big| s \in l_{c_i}(u_k), s' \in l_{c_j}(u_l), c_i, c_j \in \Sigma \cup \lambda, u_k < u_l, ((c_i \neq c_j) \vee (c_i = c_j = \lambda))\}$$

CLUSTERING STRATEGY

Consider a partition of the input ESTs into subsets (also called clusters) on which the following two standard operations are supported: *Find*(e_i) returns the cluster containing e_i, and *Union*(A, B) creates a new cluster combining the clusters A and B. These operations can be supported efficiently using a standard Union-Find algorithm [92]. To begin with, each EST is in a separate subset of the partition. At some point during the clustering, let (e_i, e_j) be the next EST pair generated by the on-demand pair generation algorithm. If *Find*(e_i) = *Find*(e_j), the two ESTs are already in the same cluster and they need not be aligned. Otherwise, an alignment test is performed. If the test succeeds, the clusters containing e_i and e_j are merged. Otherwise, they are left as they were. This process is continued until all promising pairs are exhausted. Note that the number of union operations that can be performed is $O(n)$, while the number of pairs can be $\Omega(n^2)$. That means there are $O(n)$ pairs that can lead to the right answer, though one does not know a priori what these pairs would be. The order in which promising pairs are processed does not affect the outcome but does impact the alignment work performed. The least amount of work is performed when each alignment test is a success until the final set of clusters is formed. At this point, no new promising pairs generated will need to be aligned. Thus, based on the intuition that longer exact matches more likely lead to successful alignments, the particular order in which promising pairs are generated should bring enormous savings in execution time.

4.4 Comparative Genomics

Comparative genomics is the comparison and analysis of two or more genomes or of very large genomic fragments to gain insights into molecular biology and evolution. Comparative genomics is valuable because there is significant

commonality between genomes of species that may appear very different on the surface. Moreover, coding sequences tend to be conserved in evolution, making comparative genomics a viable tool to discover coding sequences in a genome by merely comparing it with a genome of a related species and identifying the common parts. For instance, approximately 99% of human genes have a counterpart in mice. Based on a study of nearly 13,000 such genes, it has been found that the encoded proteins have a median amino-acid identity of 78.5% [14]. Learning about the subtle genomic differences between humans and mice is a starting point for understanding how these differences contribute to the vast differences between the two species (brain size, for example), and ultimately help us understand how genomes confer the distinctive properties of each species through these subtle differences.

Genome comparisons can be classified into three broad categories, depending on the relationships between the genomes being compared. In each case, a wealth of information can be gained about identifying key functional elements of the genome or subtle differences that have significant implications, some of which are highlighted below:

Individuals from the same species

A single nucleotide difference in a gene can cause a nonfunctional gene (i.e., no protein product) or give rise to a malfunctioning protein that could have serious consequences to health and tissue functioning. Such differences, known as *single nucleotide polymorphisms (SNPs)*, are the cause of genetically inherited diseases such as sickle cell anemia, cystic fibrosis, and breast cancer. They are also known to be responsible for striking hereditarily passed-on differences, including height, brain development and facial structure. Comparative genomics is a valuable tool to reveal SNPs, to help us understand the genetic basis for important diseases, and to serve as the foundation for developing treatments.

Closely related species or conserved regions across species

Closely related species such as different types of microbes can have remarkably different properties, and comparative genomics can help provide the clues that differentiate them. The United States Department of Energy has been supporting research into microbial genomes because of the ability of microbes to survive under extreme conditions of temperature, pressure, darkness, and even radiation. Understanding microbial genomes could enable far-reaching applications such as the cleanup of toxic waste and the development of biosensors. In the category of closely related species, we include species whose genomes are so closely related that a traditional alignment algorithm is capable of elucidating the genomic alignment, even though it may not be computationally advantageous to do so because of the sizes of the sequences. Certain genomic segments across more distantly related species may also have this property, where not only the genes are conserved but also the gene order is also conserved. Comparative genomics study of such regions, known as *syntenic regions*, are carried out between humans and mice [8, 13, 21, 47].

Relatively more distant species

Comparing the genomes of two species is not just a question of applying standard alignment algorithms to sequences much larger in scale. Although most genes may have counterparts across the two genomes, the order is not preserved on a global scale, which is a requirement for classical alignment algorithms to be useful. For instance, it is established that by breaking the human genome into approximately 150 pieces and rearranging the pieces, one can build a good approximation to the mouse genome. Genomic comparisons require taking into account such genomic rearrangements, which include differences in the number of repeats, reversals, translocations, fusions, and fissions of chromosomes [80]. Once such global alignment algorithms are developed and applied, alignments between syntenic blocks can be carried out.

Our discussion of computational methods for comparative genomics will follow the above-mentioned classification. We first describe how alignment algorithms can be extended to solve this problem when it is known that both genes and gene order are preserved. We then describe faster techniques for whole-genome alignments of closely related species.

Alignment-Based Methods

Consider the alignment of two large genomic sequences. If these are from different species, we expect that we are at least dealing with syntenic regions that contain orthologous genes in the same gene order. The order is important for an alignment algorithm to be useful. While the order is not necessary for a gene to play its role, evolution does tend to preserve gene order across these large blocks called syntenic regions. It is only when such an assumption is valid that a direct-alignment algorithm can be designed for genome comparisons.

Even when gene order is preserved, there are differences between a standard alignment and what is required for genomic comparisons. While the genes may be preserved, large intergenic regions may be different. Even when genes are conserved, this may largely apply only to coding regions. Since introns do not participate in the translation of a gene to its corresponding protein product, conservation of exons alone is sufficient to create a highly similar protein product. When aligning two large genomic sequences, it is important to focus on the alignment of regions of similarity and not to penalize for mismatching regions that should not be aligned in the first place. This process can be modeled as the problem of finding an ordered list of subsequences of one sequence that is highly similar to a corresponding ordered list of subsequences from the other sequence. We refer to this problem as the *syntenic alignment* problem.

The syntenic alignment problem can be formalized as follows: Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be two sequences. A subsequence A' of A is said to *precede* another subsequence A'' of A, written $A' \prec A''$, if the last character of A' occurs strictly before the first character of A'' in A. An ordered list of subsequences of A, $(A_1, A_2, \ldots, A_k)$ is called a *chain* if $A_1 \prec A_2 \prec \ldots A_k$. The syntenic alignment problem for sequences A and B is to find a chain $(A_1, A_2, \ldots, A_k)$ of subsequences in A and a chain $(B_1, B_2, \ldots, B_k)$ of subsequences in B such that the score

$$\left\{\sum_{i=1}^{k} score(A_i, B_i)\right\} - (k-1)^d$$

is maximized (see Figure 21.7).

The function $score(A_i, B_i)$ corresponds to the optimal score for the global alignment of A_i and B_i using scores for matches, penalties for mismatches, and an affine gap penalty function, as described earlier. The parameter d is a large penalty aimed at preventing alignment of short subsequences that occur by chance and not because of any biological significance. Intuitively, we are interested in finding an ordered list of matching subsequence pairs that correspond to conserved exons. One can think of the subsequence between A_i and A_{i+1} and the subsequence between B_i and B_{i+1} as an unmatched subsequence pair. The penalty d can be viewed as corresponding to an unmatched subsequence pair. For a small alphabet size, given a character in an unmatched subsequence, there is a high probability of finding the same character in the corresponding unmatched subsequence. In the absence of the penalty d, using these two characters as another matched subsequence pair would increase the score of the syntenic alignment. The penalty d serves to avoid declaring such irrelevant matching subsequences as part of the syntenic alignment, and its value should be chosen carefully by considering the length of the shortest exons that we expect. Setting d too low increases the chance of substrings, which are too short to be exons, to be considered as matching subsequences. Setting d too high prevents short matching exons from being recognized as matching subsequences.

Based on the problem definition, the syntenic alignment of two sequences $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$ can be computed by dynamic programming. Basically, we compute the syntenic alignment between every prefix of A and every prefix of B. We compute four tables C, D, I, and H of size $(m+1) \times (n+1)$. Entry $[i,j]$ in each table corresponds to the optimal score of a syntenic alignment between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$, subject to the following conditions:

- In C, a_i is matched with b_j.
- In D, a_i is matched with a gap.
- In I, a gap is matched with b_j.
- In H, either a_i or b_j is part of an unmatched subsequence.

It follows from these definitions that the tables can be computed using the following recurrence equations:

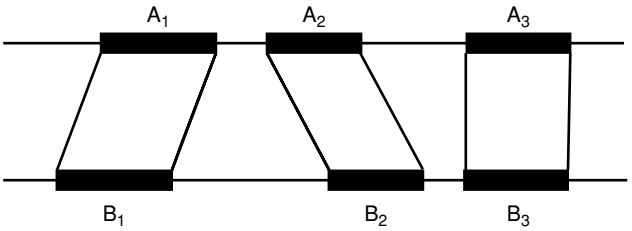

Figure 21.7. An illustration of the Syntenic alignment problem.

$$C[i,j] = f(a_i, b_j) + \max \begin{cases} C[i-1,j-1] \\ D[i-1,j-1] \\ I[i-1,j-1] \\ H[i-1,j-1] \end{cases}$$

$$D[i,j] = \max \begin{cases} C[i-1,j] - (g+h) \\ D[i-1,j] - g \\ I[i-1,j] - (g+h) \\ H[i-1,j] - (g+h) \end{cases}$$

$$I[i,j] = \max \begin{cases} C[i,j-1] - (g+h) \\ D[i,j-1] - (g+h) \\ I[i,j-1] - g \\ H[i,j-1] - (g+h) \end{cases}$$

$$H[i,j] = \max \begin{cases} C[i-1,j] - d \\ I[i-1,j] - d \\ C[i,j-1] - d \\ D[i,j-1] - d \\ H[i-1,j] \\ H[i,j-1] \end{cases}$$

Prior to computation, the top row and left column of each table should be initialized. In table H, the top row is initialized using the penalty d. The top row of I and the leftmost column of D are initialized using the affine gap penalty function. The rest of the initialization entries are set to $-\infty$. After the computation, the maximum value in $C[m, n]$, $D[m, n]$, $I[m, n]$, or $H[m, n]$ gives the optimal score. Using traceback, an optimal syntenic alignment can be reproduced.

Before computing the syntenic alignment, it is important to screen the input sequences for interspersed repeats and low-complexity subsequences using a program such as RepeatMasker [88]. Similar regions consisting of such subsequences are functionally less important than other matches but show strong sequence similarity. In the presence of repeats and low complexity subsequences, the alignment algorithm gives priority to aligning such sequences and may miss aligning the more important subsequences.

The main advantage of the alignment method is that it guarantees finding an optimal solution and is capable of detecting weak similarities. However, its quadratic run-time makes it difficult to apply this method for very large sequences. To alleviate this difficulty, Futamura et al. [31] parallelized the syntenic alignment algorithm. The program produced a syntenic alignment of a gene-rich region on human chromosome 12 (12p13; length 222, 930 bp; GenBank Accession U47924) with the corresponding syntenic region on mouse chromosome 6 (length 227, 538 bp; GenBank Accession AC002397) in about 24 minutes on a 64-processor Myrinet cluster with Pentium 1.26GHZ processors. This region contains 17 genes

[8], and most of the coding regions are identified by the program as 154 matching ordered subsequence pairs spanning 43, 445*bp* with an average identity of 79%.

Fast Genome Comparison Methods

A number of fast comparison algorithms have been developed for comparing very large-scale DNA sequences from closely related species or syntenic regions from more distantly related species [13, 21, 51, 86]. Generally, these methods perform fast identification of significant local similarities using exact matches. Then alignments are used to extend these similarities or to close gaps in alignments made up of only exact matches. Our presentation here closely follows the work of Delcher et al. [21], who developed the MUMmer program for aligning whole genomes.

MUMmer works on the assumption that long exact matching regions are part of the genomic alignment. This assumption is exploited by identifying unique maximal common substrings between the two sequences called MUMs (Maximum Unique Matches), using them to anchor the alignment, and then further exploring the regions between every pair of consecutive MUMs. The algorithm is composed of the following steps:

1. Fast identification of large MUMs using a suffix tree
2. Computing an ordered sequence of pairs of MUMs to anchor the alignment
3. Aligning regions between consecutive MUMs recursively by using shorter MUMs or dynamic programming-based tools

The first step is the identification of maximal unique matches. To this end, a generalized suffix tree of the two input sequences A and B is constructed in linear time. A maximal common substring that occurs only once in each input sequence corresponds to the path-label of an internal node that has exactly two leaf children, one from each of A and B. However, it is possible that a suffix of such a maximal common substring is also unique and corresponds to a node with similar properties. Recall that the path-label of an internal node in a suffix tree is already right maximal. To ensure that it is left maximal, one only needs to check to make sure that each suffix in the subtree of the node has a different previous character. Thus, a scan of the suffix tree to identify internal nodes with two leaf children and eliminate those that are not left maximal yields the MUMs in linear time. Only MUMs that are larger than a user-specified threshold length are identified to avoid anchoring the alignment on relatively short MUMs that may occur by coincidence.

It may not be possible to use all the MUMs so discovered in anchoring the alignment. A pair of MUMs may cross, allowing the inclusion of only one of them in any viable alignment. To identify a large set of MUMs that does not contain any such pairwise conflicts, the following method is used: the MUMs are first sorted according to their position in genomic sequence A and are labeled 1, 2, ... k in that order, where k is the number of MUMs. The same labeling is applied to the corresponding MUMs on genomic sequence B. A longest increasing subsequence of MUM along B is then sought. This can be solved by a variation of the Longest Increasing Subsequence (LIS) problem [37] that takes into account the lengths of the MUMs so that what is maximized is the total lengths of all the selected MUMs and not the number of MUMs. This can be done in $O(k \log k)$ time.

Once the alignment is anchored using the selected MUMs, one is left with the task of aligning the regions between every consecutive pair of MUMs. The chief advantage of this strategy is the quick decomposition of the original problem into several smaller subproblems. While this should considerably reduce the run-time even if each of the subproblems is addressed using an alignment algorithm such as syntenic alignment, further computational savings can be obtained by recursing on this strategy using shorter MUM threshold length. Also, certain special cases can be readily identified that can be treated separately. Three such special cases are identified here:

- Two MUM pairs separated by a single differing nucleotide – This is classified as a SNP.
- Two MUMs that are consecutive along one genome but are separated in the other – The subsequence separating the two MUMs is treated as an insert.
- Overlapping MUMs – If the intervals spanned by two MUMs along both genomes overlap, this is an indication of a tandem repeat (repeats that occur consecutively with no intervening sequences) separating the two MUMs. The genome that has fewer occurrences of the tandem repeats will restrict the lengths of the MUMs and causes the two flanking matching regions to be identified as two different MUMs.

Once the special cases are identified, the remaining gaps in alignment can be filled by using quadratic time alignment algorithms. If these regions are long enough, a recursive application of MUMs strategy can be applied while eventually closing the remaining gaps using alignment. This method should produce fairly accurate alignments in very short amounts of time. For example, Delcher et al. report alignment of the same human chromosome 12p13 and mouse chromosome 6 syntenic regions described earlier in just 30 seconds [21].

Genomic Rearrangements

The previous algorithms are useful in aligning very closely related genomes where gene order is largely preserved or syntenic regions across genomes within which gene order is preserved. To extend comparative genomics beyond that, computational modeling of genomic rearrangements is needed. Such rearrangements are useful in hypothesizing the relative evolutionary distance between two species and also in identifying syntenic regions that can be aligned using previously described methods. A simple way to model this problem is to consider each genome as consisting of essentially the same set of genes. Each gene is also given an orientation depending on the genomic strand on which it appears. This can be modeled by placing a plus sign in front of each gene that appears on one of the strands and choosing a minus sign for the genes appearing on the complementary strand. Genomic rearrangements are modeled as *inversions*, where inversion of a stretch of the genome is represented by reversing the sequence of genes contained in it and reversing the sign of each gene. The distance between the genomes is then measured as the minimum number of inversions required to convert one genome into the other. These important methods are not described here due to space limitations, largely because the underlying algorithms are different from the

exact match and alignment-based methodologies that this chapter is focused on. The interested reader is referred to [9–11, 40, 54, 80, 93].

5 CONCLUSIONS AND FUTURE CHALLENGES

This chapter has aimed at providing a brief introduction to bioinformatics and conveying the flavor of research in computational genomics to readers with little or no familiarity with bioinformatics. The volume of research results that have accumulated so far in bioinformatics and the large number of researchers engaged in research in this area indicate that this field is no longer in its infancy. In fact, it is difficult to provide complete coverage of this area in a medium-sized textbook. The approach taken in this chapter has been to provide a bird's-eye view of computational genomics by looking at a number of challenging research problems in a holistic and integrated manner by focusing on the underlying fundamentals. It is hoped that this approach enabled the coverage of the material in reasonable depth within the scope of a chapter and provided enough understanding to spark the reader's curiosity. The large number of references provided should serve as a starting point for further investigation. For a fairly comprehensive treatment, including recent research directions in bioinformatics and computational biology, the reader is referred to [6].

Many computational challenges remain in bioinformatics, promising decades of interesting work for researchers in this area. Within computational genomics, some of the main challenges can be summarized as (1) sequencing the genomes of many more organisms, (2) understanding the genes and their structure and function within each organism, (3) developing capabilities to compare and analyze a large number of genomes collectively, (4) constructing evolutionary relationships between all known species, known as the *tree-of-life project*, (5) using gene expression studies to understand gene interactions, (6) inferring complex protein interaction pathways and networks, and (7) understanding gene regulatory behaviour as related to its impact on developmental genetics. Computational structure biology is another exciting area that is not covered in this chapter. Computational determination of protein structure from its amino acid sequence is often mentioned as the "holy grail" problem in bioinformatics. Finding sequences that fold into desired structures and understanding the mechanisms of protein–protein docking and protein–drug docking are considered vital to pharmaceutical research. Developing structure databases to enable the search for structural homologies is a difficult problem to address, but may well end up showing that current sequence-based alignment strategies are pursued for computational convenience. Remarkable discoveries await research in computational medicine. One promising area of research is personalized medicine, where an understanding of the complex relationships between the genetic composition of an individual and his or her tendency to develop diseases and response to drugs is expected to lead to the design of targeted treatments with tremendous health care benefits. Progress in solving many such challenges facing modern biology and medicine can only be made by interdisciplinary teams of researchers. Computation will remain an integral part of most potential discoveries and should provide exciting applied problems for computing researchers to work on for decades to come.

REFERENCES

[1] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch (2004): Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, *2*.

[2] B. Alberts, A. Hohnson, J. Lewis, M. Raff, K. Roberts, and P. Walter (2002): *Molecular Biology of the Cell*. Garland Science, New York, NY.

[3] S.F. Altschul (1991): Amino acid substitution matrices from an information theory perspective. *Journal of Molecular Biology*, *219*:555–565.

[4] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman (1990): Basic local alignment search tool. *Journal of Molecular Biology*, *215*(3), 403–410.

[5] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman (1997): Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, *25*, 3389–3402.

[6] S. Aluru, (ed) (2005): *Handbook of Computational Molecular Biology*. CRC Press, Boca Raton, FL.

[7] S. Aluru, N. Futamura, and K. Mehrotra (2003): Parallel biological sequence comparison using prefix computations. *Journal of Parallel and Distributed Computing*, *63*(3), 264–272.

[8] M.A. Ansari-Lari, J.C. Oeltjen, S. Schwartz, Z. Zhang, D.M. Muzny, J. Lu, J.H. Gorrell, A.C. Chinault, J.W. Belmont, W. Miller, and R.A. Gibbs (1998): Comparative sequence analysis of a gene-rich cluster at human chromosome 12p13 and its syntenic region in mouse chromosome 6. *Genome Research*, *8*, 29–40.

[9] D.A. Bader, B. M.E. Moret, and M. Yan (2001): A linear-time algorithm for computing inversion distance between two signed permutations with an experimental study. *Journal of Computational Biology*, *8*(5), 483–491.

[10] V. Bafna and P.A. Pevzner (1995): Sorting by reversals: genome rearrangements in plant organelles and evolutionary history of X chromosome. *Molecular Biology and Evolution*, *12*, 239–246.

[11] V. Bafna and P.A. Pevzner (1996): Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, *25*(2), 272–289.

[12] S. Batzoglou, D. Jaffe, K. Stanley, J. Butler, et al. (2002): ARACHNE: A wholegenome shotgun assembler. *Genome Research*, *12*, 177–189.

[13] S. Batzoglou, L. Pachter, J.P. Mesirov, B. Berger, and E.S. Lander (2000): Human and mouse gene structure: comparative analysis and application to exon prediction. *Genome Research*, *10*, 950–958.

[14] M.S. Boguski (2002): Comparative genomics: the mouse that roared. *Nature*, *420*, 515–516.

[15] J.K. Bonfield, K. Smith, and R. Staden. (1995): A new DNA sequence assembly program. *Nucleic Acids Research*, *24*, 4992–2999.

[16] International Human Genome Sequencing Consortium (2001): Initial sequencing and analysis of the human genome. *Nature*, *409*, 860–921.

[17] Mouse Genome Sequencing Consortium (2002): Initial sequencing and comparative analysis of the mouse genome. *Nature*, *420*, 520–562.

[18] E. Coward, S. A. Haas, and M. Vingron. (2002): SpliceNest: visualizing gene structure and alternative splicing based on EST clusters. *Trends in Genetics*, *18*(1), 53–55.

[19] M. Crochemore, G.M. Landau, and Z. Ziv-Ukelson (2002): A subquadratic sequence alignment algorithm for unrestricted cost metrics. In *Proc. Symposium on Discrete Algorithms*, pp. 679–688.

[20] M.O. Dayhoff, R. Schwartz, and B.C. Orcutt (1978): *Atlas of Protein Sequence and Structure*, volume 5. A model of evolutionary change in proteins: matrices for detecting distant relationships, pp. 345–358. National Biomedical Research Foundation.

[21] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. While, and S.L. Salzberg (1999): Alignment of whole genomes. *Nucleic Acids Research*, *27*, 228–233.

[22] R. Durbin, S.R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*.

[23] E.W. Edmiston, N.G. Core, J.H. Saltz, and R.M. Smith (1988): Parallel processing of biological sequence comparison algorithms. *International Journal of Parallel Programming*, *17*(3), 259–275.

[24] S. Emrich, S. Aluru, Y. Fu, T. Wen, et al. (2004): A strategy for assembling the maize (*zea mays* L.) genome. *Bioinformatics*, *20*, 140–147.

[25] M. Farach (1997): Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science*, pp. 137–143. IEEE.

[26] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan (2000): On the sorting-complexity of suffix tree construction. *Journal of the Association of Computing Machinery*, *47*.

[27] D. Fernández-Baca, T. Seppalainen, and G. Slutzki (2002): Bounds for parametric sequence comparison. *Discrete Applied Mathematics*, *118*, 181–198.

[28] P. Ferragina and G. Manzini (2000): Opportunistic data structures with applications. In *41th Annual Symposium on Foundations of Computer Science*, pp. 390–398. IEEE.

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

[30] R.D. Fleischmann, M.D. Adams, O. White, R.A. Clayton, et al. (1995): Whole-genome random sequencing and assembly of *haemophilus influenzae* rd. *Science*, *269*(5223), 496–512.

[31] N. Futamura, S. Aluru, and X. Huang (2003): Parallel syntenic alignments. *Parallel Processing Letters*, *13*, 689–703.

[32] C. Gemund, C. Ramu, B. A. Greulich, and T. J. Gibson (2001): Gene2EST: a BLAST2 server for searching expressed sequence tag (EST) databases with eukaryotic gene-sized queries. *Nucleic Acids Research 29*, 1272–1277.

[33] R. Giegerich and S. Kurtz (1997): From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. 19:331–353.

[34] O. Gotoh (2000): Homology-based gene structure prediction: simplified matching algorithm using a translated codon (tron) and improved accuracy by allowing for long gaps. *Bioinformatics*, *16*(3), 190–202.

[35] P. Green (1996): *http://www.mbt.washington.edu/phrap.docs/phrap.html*.

[36] R. Grossi and J.S. Vitter (2000): Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Symposium on the Theory of Computing*, pp. 397–406. ACM.

[37] D. Gusfield (1997): *Algorithms on Strings Trees and Sequences*. New York.

[38] D. Gusfield, K. Balasubramaniam, and D. Naor (1994): Parametric optimization of sequence alignment. *Algorithmica*, *12*, 312–326.
[39] S. A. Haas, T. Beissbarth, E. Rivals, A. Krause, and M. Vingron (2000): GeneNest: automated generation and visualization of gene indices. *Trends in Genetics*, *16*(11), 521–523.
[40] S. Hannenhalli and P.A. Pevzner (1999): Transorming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the Association for Computing Machinery*, *46*(1), 1–27.
[41] P. Havlak, R. Chen, K.J. Durbin, A. Egan, Y.R. Ren, and X.Z. Song (2004): The Atlas genome assembly system. *Genome Research*, *14*(4):721–732.
[42] S. Henikoff and J.G. Henikoff (1992): Amino acid substitution matrices from protein blocks. *Proc. National Academy of Sciences*, *89*, 10915–10919.
[43] D.S. Hirschberg (1975): A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, *18*(6), 341–343.
[44] X. Huang (1989): A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, *18*(3), 223–239.
[45] X. Huang (1990): A space-efficient algorithm for local similarities. *Computer Applications in the Biosciences*, *6*(4), 373–381.
[46] X. Huang (1992): A contig assembly program based on sensitive detection of fragment overlaps. *Genomics*, *14*, 18–25.
[47] X. Huang and K. Chao (2003): A generalized global alignment algorithm. *Bioinformatics*, *19*(2), 228–233.
[48] X. Huang and A. Madan (1999): CAP3: A DNA sequence assembly program. *Genome Research*, *9*(9), 868–877.
[49] X. Huang and J. Zhang (1996): Methods for comparing a DNA sequence with a protein sequence. *Computer Applications in Biosciences*, *12*(6), 497–506.
[50] D.B. Jaffe, J. Butler, S. Gnerre, and E. Mauceli, et al. (2003): Whole-genome sequence assembly for mammalian genomes: ARACHNE2. *Genome Research*, *13*, 91–96.
[51] N. Jareborg, E. Birney, and R. Durbin (1999): Comparative analysis of noncoding regions of 77 orthologous mouse and human gene pairs. *Genome Research*, *9*, 815–824.
[52] A. Kalyanaraman, S. Aluru, V. Brendel, and S. Kothari (2003): Space and time efficient parallel algorithms and software for EST clustering. *IEEE Transactions on Parallel and Distributed Systems*, 14.
[53] Z. Kan, E. C. Rouchka, W. R. Gish, and D. J. States (2001): Gene structure prediction and alternative splicing analysis using genomically aligned ESTs. *Genome Research*, *11*, 889–900.
[54] H. Kaplan, R. Shamir, and R.E. Tarjan (2000): A faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal on Computing*, *29*(3), 880–892.
[55] J. Kärkkäinen and P. Sanders (2003): Simpler linear work suffix array construction. In *International Colloquium on Automata, Languages and Programming*, to appear.
[56] R.M. Karp (2003): The role of algorithmic research in computational genomics. In *Proc. IEEE Computational Systems Bioinformatics*, pp. 10–11. IEEE.

[57] J. Kececioglu and E. Myers (1995): Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, *13*(1-2), 7–51.
[58] P. Ko and S. Aluru (2003): Space-efficient linear-time construction of suffix arrays. In *14th Annual Symposium, Combinatorial Pattern Matching*.
[59] P. Ko, M. Narayanan, A. Kalyanaraman, and S. Aluru (2004): Space conserving optimal DNA-protein alignment. In *Proc. IEEE Computational Systems Bioinformatics*, pp. 80–88.
[60] A. Krause, S. A. Haas, E. Coward, and M. Vingron (2002): SYSTERS, GeneNest, SpliceNest: Exploring sequence space from genome to protein. *Nucleic Acids Research*, 30.
[61] A. Krause, J. Stoye, and M. Vingron (2000): The SYSTERS protein sequence cluster set. *Nucleic Acids Research*, *28*, 270–272.
[62] E. Lander, J.P. Mesirov, and W. Taylor (1988): Protein sequence comparison on a data parallel computer. In *Proc. International Conference on Parallel Processing*, pp. 257–263.
[63] E.S. Lander and M.S. Waterman (1988): Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics*, *2*, 231–239.
[64] F. Liang, I. Holt, G. Pertea, S. Karamycheva, S. Salzberg, and J. Quackenbush (2000): An optimized protocol for analysis of EST sequences. *Nucleic Acids Research*, *28*(18), 3657–3665.
[65] H.F. Lodish, A. Berk, P. Matsudaira, C.A. Kaiser, M. Krieger, M.P. Scott, S.L. Zipursky, and J. Darnell (2003): *Molecular Cell Biology*. W.H. Freeman and Company, New York, NY.
[66] P. A. Pevzner M. S. Gelfand, and A. Mironov (1996): Gene recognition via spliced alignment. *Proc. National Academy of Sciences*, *93*, 9061–9066.
[67] U. Manber and G. Myers (1993): Suffix arrays: a new method for on-line search. *SIAM Journal on Computing*, *22*, 935–48.
[68] W.J. Masek and M.S. Paterson (1980): A faster algorithm for computing string edit distances. *Journal of Computer and System Sciences*, *20*, 18–31.
[69] E. M. McCreight (1976): A space-economical suffix tree construction algorithm. *Journal of the ACM*, *23*, 262–72.
[70] B. Modrek and C. Lee (2002): A genomic view of alternative splicing. *Nature Genetics*, *30*, 13–19.
[71] D.W. Mount (2001): *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory.
[72] J.C. Mullikin and Z. Ning (2003): The phusion assembler. *Genome Research*, *13*, 81–90.
[73] E. Myers (1994): *Advances in Sequence Assembly*, chapter in Automated DNA Sequencing and Analysis Techniques (C. Ventner, ed), pp. 231–238. Academic Press Limited.
[74] E.W. Myers (1995): Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, *2*(2), 275–290.
[75] E.W. Myers, G.G. Sutton, A.L. Delcher, I.M. Dew, et al. (2000): A whole genome assembly of *drosophila. Science*, *287*(5461), 2196–2204.
[76] E.W. Myers and W. Miller (1988): Optimal alignments in linear space. *Computer Applications in the Biosciences*, *4*(1), 11–17.

[77] S.B. Needleman and C.D. Wunsch (1970): A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, *48*, 443–453.
[78] L. Patcher and B. Strumfels (2004): Parametric inference for biological sequence analysis. *Proc. National Academy of Sciences*, to appear.
[79] H. Peltola, H. Soderlund, and E. Ukkonen (1984): SEQAID: a DNA sequence assembly program based on a mathematical model. *Nucleic Acids Research*, *12*, 307–321.
[80] P. Pevzner and G. Tesler (2003): Transforming men into mice: the Nadeau-Taylor chromosomal breakage model revisted. In *Proc. International Conference on Research in Computational Molecular Biology (RECOMB)*, pp. 247–256. ACM.
[81] P.A. Pevzner (2000): *Computational Molecular Biology: An Algorithmic Approach*. MIT Press.
[82] J. Quackenbush, J. Cho, D. Lee, F. Liang, I. Holt, S. Karamycheva, B. Parvizi, G. Pertea, R. Sultana, and J. White (2001): The TIGR gene indices: analysis of gene transcript sequences in highly sampled eukaryotic species. *Nucleic Acids Research*, *29*, 159–164.
[83] S. Rajko and S. Aluru (2004): Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, *15*(11).
[84] F. Sanger, S. Nicklen, and A.R. Coulson (1977): DNA sequencing with char-interminating inhibitors. *Proc. National Academy of Sciences*, *74*, 5463–5467.
[85] D. Sankoff and J.B. Kruskal (1983): *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Reading, MA.
[86] S. Schwartz, Z. Zhang, K. Frazer, A. Smit, C. Riemer, J. Bouck, R. Gibbs, R. Hardison, and W. Miller (2000): PipMaker-a web server for aligning two genomic DNA sequences. *Genome Research*, *10*, 577–586.
[87] J. Setubal and J. Meidanis (1997): *Introduction to Computational Molecular Biology*. PWS Publishing Company, Boston, MA.
[88] A. Smit and P. Green (1999): http://ftp.genome.washington.edu/RM/RepeatMasker.html, 1999.
[89] T.F. Smith and M.S. Waterman (1981): Identification of common molecular subsequences. *Journal of Molecular Biology*, *147*, 195–197.
[90] D.J. States, W. Gish, and S.F. Altschul (1991): Improved sensitivity of nucleic acid database searches using application-specific scoring matrices. *Methods*, *3*, 66–70.
[91] G. Sutton, O. White, M. Adams, and A. Kerlavage (1995): TIGR assembler: A new tool for asembling large shotgun sequencing projects. *Genome Science and Technology*, *1*, 9–19.
[92] R.E. Tarjan (1975): Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, *22*(2), 215–225.
[93] G. Tesler (2002): Efficient algorithms for multichromosomal genome rearrangements. *Journal of Computer and System Sciences*, *65*, 587–609.
[94] E. Ukkonen (1995): On-line construction of suffix-trees. *14*, 249–60.
[95] J.C. Venter, M.D. Adams, E.W. Myers, P.W. Li, et al. (2001): The sequence of the human genome. *Science*, *291*(5507), 1304–1351.

[96] M.S. Waterman (1995): *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman and Hall, London.
[97] P. Weiner (1973): Linear pattern matching algorithms. In *14th Symposium on Switching and Automata Theory*, pp. 1–11.
[98] R. Yeh, L. P. Lim, and C. B. Burge (2001): Computational inference of homologous gene structures in the human genome. *Genome Research*, *11*, 803–816.
[99] Z. Zhang, W. R. Pearson, and W. Miller (1997): Aligning a DNA sequence with a protein sequence. *Journal of Computational Biology*, pp. 339–49.