Algorithms for String Matching with Applications in Molecular Biology

By J. L. Holloway

A Thesis submitted to

Oregon State University

in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

Completed 7 August 1992
Commencement June 1993.

AN ABSTRACT OF THE THESIS OF

J. L. Holloway for the degree of Doctor of Philosophy in
Computer Science presented on 7 August 1992.
Title: Algorithms for String Matching with Applications in Molecular Biology

Abstract approved:

Paul Cull.

Abstract approved:

Walter Rudd.

As the volume of genetic sequence data increases due to improved sequencing techniques and increased interest, the computational tools available to analyze the data are becoming inadequate. This thesis seeks to improve a few of the computational methods available to access and analyze data in the genetic sequence databases. The first two results are parallel algorithms based on previously known sequential algorithms. The third result is a new approach, based on assumptions that we believe make sense in the biological context of the problem, to approximating an $\mathcal{NP}$–complete problem. The final result is a fundamentally new approach to approximate string matching using the divide and conquer paradigm instead of the dynamic programming approach that has been used almost exclusively in the past.

Dynamic programming algorithms to measure the distance between sequences have been known since at least 1972. Recently there has been interest in developing parallel algorithms to measure the distance between two sequences. We have developed an optimal parallel algorithm to find the edit distance, a metric frequently used to measure distance, between two sequences.

It is often interesting to find the substrings of length k that appear most frequently in a given string. We give a simple sequential algorithm to solve this problem and an efficient parallel version of the algorithm. The parallel algorithm uses an efficient novel parallel bucket sort.

When sequencing a large segment of DNA, the original DNA sequence is reconstructed using the results of sequencing fragments, that may or may not contain errors, of many copies of the original DNA. New algorithms are given to solve the problem of reconstructing the original DNA sequence with and without errors introduced into the fragments. A program based on this algorithm is used to reconstruct the human beta globin region (HUMHBB) when given a set of 300 to 500 mers drawn randomly from the HUMHBB region.

Approximate string matching is used in a biological context to model the steps of evolution. While such evolution may proceed base by base using the change, insert, or delete operators, there is also evidence that whole genes may be moved or inverted. We introduce a new problem, the string to string rearrangement problem, that allows movement and inversion of substrings. We give a divide and conquer algorithm for finding a rearrangement of one string within another.

Approved:

Professor of Computer Science in charge of major

Professor of Computer Science in charge of major

Head of Department of Computer Science

Dean of Graduate School

Date thesis presented 7 August 1992

Text entered by J. L. Holloway

Approved by Committee:

Major Professor (Paul Cull)

Major professor (Walter Rudd)

Committee Member (Bella Bose)

Committee Member (Bruce D'Ambrosio)

Graduate School Representative (Christopher Bell)

Date thesis presented 7 August 1992

ACKNOWLEDGEMENTS

Thanks to Paul Cull, Walter Rudd, Bruce D'Ambrosio, Bella Bose, and Christopher Bell for serving on my committee, guiding the direction of this research, and suggesting improvements on early drafts of my dissertation.

Thanks to Paul Cull, Rick Hangartner, and Shawn Larson for many interesting discussions on numerous topics over coffee. Several ideas were born, and many more died, at the Beanery.

Thanks to Charlotte, David, Judy, and Mildred for creating wonderful places to come home to.

Special thanks to my mother, Judy Kelble, for providing me with every possible opportunity to succeed.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the 7 May 1992 issue of *Nature*, S. Oliver et. al. [213] reported that the complete sequence of the first eukaryotic chromosome, chromosome III of *Saccharomyces cerevisiae* (bakers' yeast), had been determined. This and many similar projects have contributed to the rapid growth of the nucleic acid databases during the past decade. Figures 1.1 and 1.2 show how the number of nucleotides in the GenBank database has been growing during the past ten years. How will this information be analyzed? How will it be accessed in the future?

1.1 Motivation

The goals of the human genome effort as stated in the report "The U.S. human genome project: The first five years, FY 1991–1995" [275] include the following:

- Develop algorithms and analytical tools to interpret genomic information.
- Create database tools that provide easy access to up–to–date physical mapping, genetic mapping, chromosome mapping, and sequencing information and allow ready comparison of the data in these several data sets.
- Develop effective software and database designs to support large–scale mapping and sequencing projects.

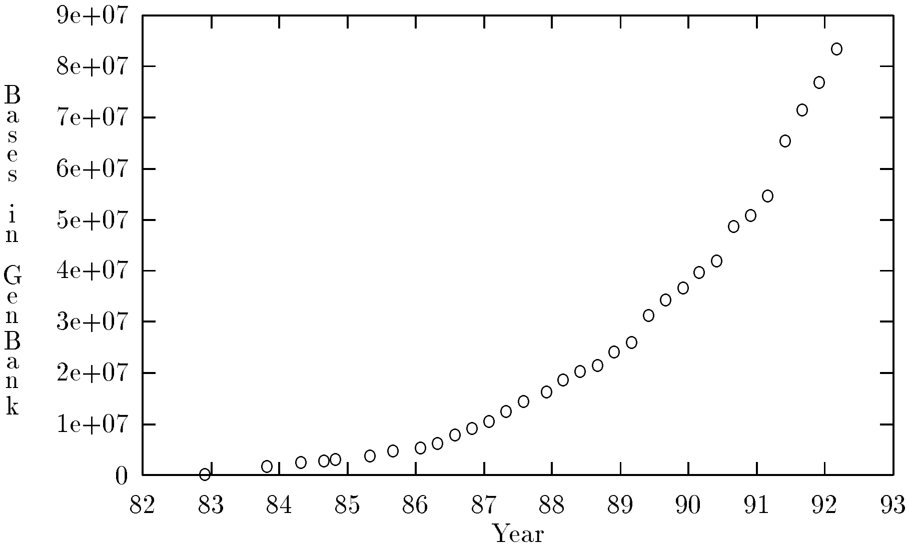


Figure 1.1: The size of the GenBank nucleic acid database measured in nucleotides during the past ten years.

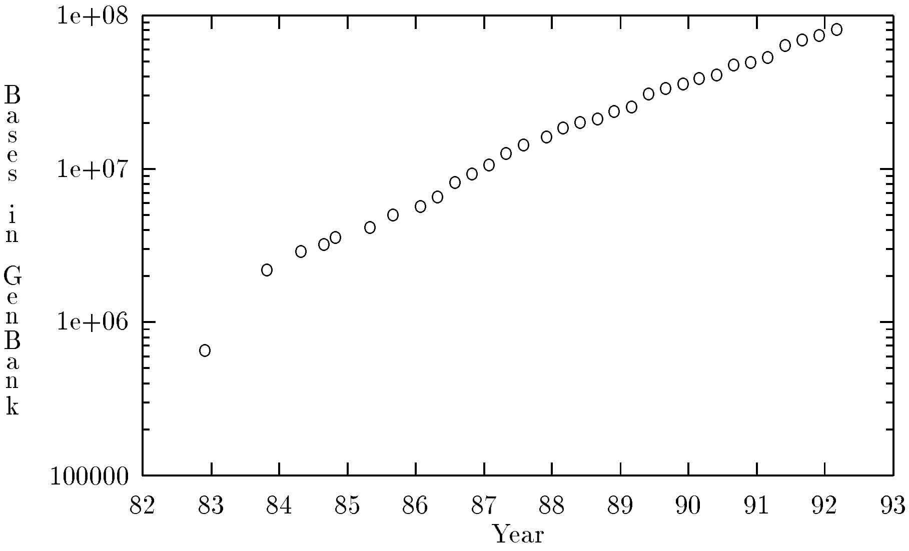


Figure 1.2: The size of the GenBank nucleic acid database measured in nucleotides on a log scale during the past ten years.

The human genome project has created and supports a community of researchers investigating algorithms to improve the analysis and access to the genetic databases. My interest is in the applications of approximate string matching to the problems of accessing and analyzing data in the DNA, RNA, and protein sequence databases. A number of other reasons have been given by others for studying approximate string matching including, spelling correction, tree ring dating, geological strata recognition, and bird song recognition.

1.1.1 Shotgun sequencing

Current technology limits the number of bases of a DNA sequence that can be determined in one operation to several hundred. To determine the sequence of longer pieces of DNA, the DNA must first be broken into appropriate sized pieces. The two methods frequently used to determine the order of bases in a DNA sequence are ordered sequencing and shotgun sequencing.

Ordered sequencing creates a collection of DNA fragments that all have one end in common by progressively removing bases from the other end of the sequence with an enzymatic reaction. As the bases are being removed, samples of the DNA are removed from the enzymatic reaction and the degradation of the DNA in the sample is stopped. Each successive sample of DNA will be shorter than the previous sample since the longer the enzymatic reaction is allowed to proceed, the shorter the remaining DNA will be. If the DNA samples are removed at appropriate intervals, we will have a collection of different length DNA fragments, each with a common suffix. Each of the DNA fragments is cloned to produce a large quantity of the DNA that is then sequenced. Since we know the order of each clone in the complete sequence, once the clones have been sequenced, reconstructing the original sequence is simple.

In shotgun sequencing, instead of producing an ordered set of sequence fragments, a random set of fragments is produced. Both the starting and ending position of the sequence fragment vary over the length of the sequence. The sequence fragments are sorted by size and appropriate sized fragments are cloned so that enough DNA is produced to be sequenced. The DNA may be fragmented by sonicating the DNA, exposing the DNA to DNase I, or exposing the DNA to restriction enzymes. The overlaps that occur between the sequenced random DNA fragments are used to reconstruct the original sequence. Since errors occur in the process of sequencing, reading the sequencing gels, and entering the data, approximate string matching methods must be used to reconstruct the original sequence from the sequence fragments.

1.1.2 Sequence similarity and alignments

Once a DNA sequence that codes for some protein with unknown function has been sequenced, the databases of DNA sequences are searched for similar DNA sequences. The function of the unknown piece of DNA may be inferred from DNA sequences with similar structure that are found in the databases. In many situations, knowing that two sequences are similar is enough, but in other situations such as constructing a phylogenetic tree, a one to one alignment between the bases of the sequences is required. The processes of evaluating the similarity of sequences and constructing an alignment between two sequences both make heavy use of algorithms to do approximate string matching. The size of these problems varies greatly, from aligning two short sequences to doing a complete n–wise comparison of all sequences in the GenBank database.

1.1.3 Motifs

A motif (also called a template or a fingerprint) is a short pattern in an amino acid or nucleotide sequence that is frequently associated with some structure or function of the molecule. Motifs involve computation in two situations, discovery of the motif, and searching sequences for previously discovered motifs.

The problem of systematically discovering a motif from a collection of sequences that are believed to have a similar structure or function is challenging.

There are several aspects to discovering a motif that make it difficult. A motif is not always continuous, there may be gaps between positions that specify certain characters. For example, the leucine zipper motif specifies the amino acid leucine appear in every seventh position of the amino acid sequence. A motif is not always perfectly conserved, there will be positions in the motif that are only partially specified. The motif is not always located at the same position in each sequence in which it appears.

Once a motif has been specified it is used to search for other occurrences of the motif. Several hundred motifs have been collected by A. Bairoch [28] in the PROSITE database. A single motif may be used to select sequences from a database in an attempt to find all sequences with a specific structure or function. A single sequence may be compared with each motif in the database to find one or a few motifs that match parts of the sequence. The motifs that match a sequence may be used to infer the structure or function of the molecule.

1.1.4 RNA secondary structure

Transfer RNA (tRNA) and ribosomal RNA (rRNA) form a stem and loop structure that is often shown in two dimensions. Figure 1.3 shows the secondary structure of the 5s rRNA of *Bacillus licheniformis* as computed by the MFOLD program of M. Zuker & P. Stiegler [298]. The resulting stem and loop structure is the arrangement of the bases in the rRNA that minimizes the free energy in the two dimensional structure. The stems generally follow the Watson–Crick base pairing so that G's usually align with C's and A's with U's. Occasionally G's will pair with U's.

The two subsequences that compose a stem in the rRNA secondary structure are approximately inverted complements of one another. Algorithms that can quickly and accurately search for approximate substring matches in an RNA sequence might be able to improve the speed of programs used to compute the secondary structure of RNA molecules.

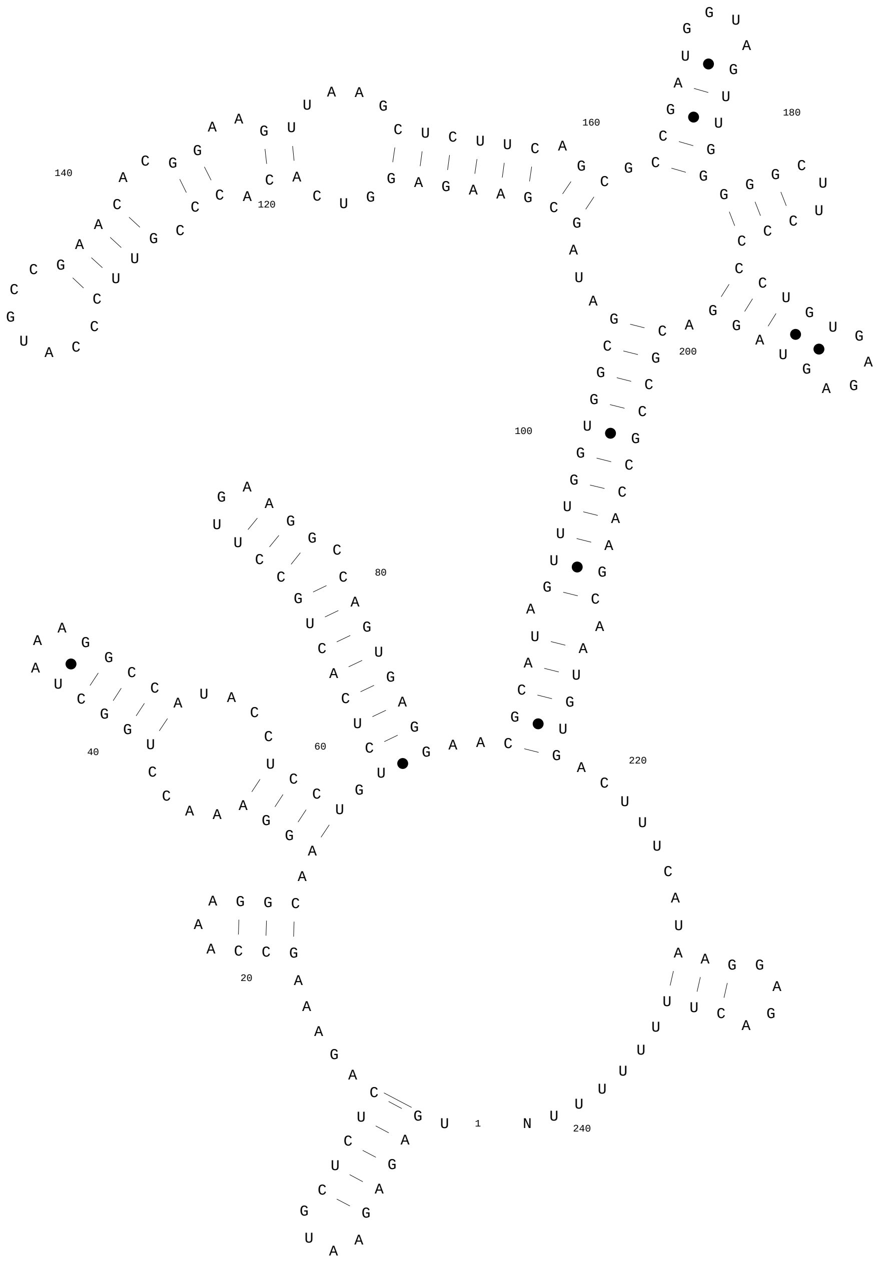


Figure 1.3: One of three possible secondary structures for the 5s–rRNA of *Bacillus licheniformis* as computed by the MFOLD program of Zuker & Stiegler. The lines connecting adjacent base pairs in the stems represent strong bonds and the dots represent weaker bonds.

1.2 Overview of thesis

The remainder of the thesis is organized as follows. Chapter two is a review of the literature in exact string matching, approximate string matching and related areas. Since string matching and pattern matching are fundamental questions in both computer science and computational biology, the literature is extensive. The final few pages of chapter two discuss problems that are not directly addressed in the rest of the thesis, but that may benefit from the algorithms that are developed in this thesis or may contribute to the areas that are discussed in this thesis.

Chapters three and four present efficient parallel solutions to two independent problems. The dynamic programming approach to aligning sequences has been investigated extensively in both the computer science and computational biology literature. In chapter three, I present an efficient parallel algorithm to compute the edit distance between two sequences, a method of measuring the similarity between sequences, that runs in $O(mn/p)$ time using $O(p),\ p \leq \min(m, n)$ EREW PRAM processors when given two sequences of length m and n. Chapter four presents an efficient parallel solution to the problem of finding the most frequent substring of length m given a string of length n over a fixed, finite alphabet that uses $O(mn/p)$ time with $O(p),\ p \leq n/\log n$ EREW PRAM processors. The solution to the most frequent substring problem required the development of an efficient parallel bucket sort algorithm.

Chapters five and six examine the problem of reconstructing a sequence when given the fragments of several copies of the sequence. The original sequence is reconstructed by examining matches between suffixes and prefixes of the fragments. It is assumed that given a set of sequence fragments, the original sequence is the shortest common superstring of the fragment strings. A common superstring of a set of strings is some string that contains, as a substring, each string in the set of strings. The problem of finding the minimal length reconstruction has been shown to be $\mathcal{NP}$–complete. Previous approaches to this problem have been to develop algorithms that approximate the correct arrangement of the fragments.

The approach used in chapters five and six is to make assumptions about the set of fragments and then compute the correct arrangement. Chapter five discusses two new algorithms to solve the problem when the matches between suffixes and prefixes are assumed to be exact matches. In chapter six, two algorithms are developed that will solve the problem when the suffixes and prefixes do not exactly match but are allowed to have a number of mismatches that is related to the length of the suffix/prefix overlap.

Chapter seven presents a fundamentally new approach to solving the general problem of approximate string matching. Dynamic programming solutions for the approximate string matching problem have been used almost exclusively to solve the problem. A restriction that most dynamic programming solutions impose is that every operator used to transform one string into the other must have the non–intersection property. Given the strings A and B, the non–intersection property states that if the characters A_i and B_j are aligned with one another, and A_k and B_l are aligned with one another, then if $i < k$, it must also be the case that $j \leq l$. The non–intersection property prohibits the consideration of A matching B^{-1}, although this situation occurs in the DNA databases. I developed a divide and conquer algorithm that is not constrained by the non–intersection property to solve the approximate string matching problem. The divide and conquer algorithm is simple, uses a very small amount of work space, runs quickly, and is easily parallelized. Each processing node in the parallel algorithm uses a constant amount of space, communicates with at most four other processors and can complete the computations for each input character in a small constant amount of time. Several variations of the algorithm are given and the algorithm is applied to a number of DNA sequences to demonstrate its utility.

Chapter eight, the final chapter, briefly discusses several problems along with ideas for their solution that I believe are interesting questions for future research.

- Dynamic programming algorithms have been used to align DNA sequences with amino acid sequences. Since every amino acid corresponds to three adjacent DNA bases, the deletion or insertion of a DNA base changes the "reading frame" of all positions following in the string and so must be treated differently than an ordinary insertion or deletion.

- The divide and conquer approach to approximate string matching can be extended to multidimensional approximate matching. The divide and conquer algorithm allows simultaneous searches for a number of transformations of the pattern so that, when searching a bitmap for a pattern bitmap, a match can be found even if parts of the bitmap must be transformed.

- Heavy use of the suffix array data structure is made in some of the approximate pattern matching algorithms. Since the genetic databases are large and growing, algorithms to construct a suffix array in parallel and incrementally update the suffix array would be useful.

- Finally, I give an idea that might allow us to find all substrings of a given text string that have a hamming distance within k of a given pattern string quickly for small pattern strings.

Chapter 2

Literature Review

2.1 History, review, and overview

In 1972, Ulam [274] wrote a 'preview' article with the title "Some ideas and prospects in biomathematics" that contained several of the ideas that have been extensively studied in the past twenty years. He advocated a distance metric between DNA sequences as a measure of sequence similarity. He considered the operations of individual insertions, deletions, and mutations and suggested that insertions and deletions of blocks of characters might be important to consider. In that paper, Ulam also discussed the construction of phylogenies using both distance methods and character based methods.

There are a number of books for geneticists and molecular biologists that address the problems of sequence analysis. In the book "Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison", editors David Sankoff and Joseph Kruskal [235] gathered together an excellent collection of introductory papers on the topics of sequence searching, sequence alignment, pattern recognition and speech processing and recognition. In 1986, R. Doolittle [73] wrote the book "Of URFs and ORFs"[1] that addressed the problems of identifying genetic sequences and testing the significance of matches found between the query sequence and sequences in the databases that appear to be similar. Two books appeared in 1987, the first by G. von Heijne [278] and the

[1] Unidentified Reading Frame (URF) and Open Reading Frame (ORF)

second by Bishop & Rawlings [40], and another book in 1988 by A. Lesk [179] covered many of the same problems and procedures as the two books mentioned previously. Recently M. Gribskov and M. Devereux [124] edited a book that addresses the steps of sequence entry into the computer, overlap detection, editing, analyzing, and reporting of sequence data. Even though the intended audience for these books is geneticists and molecular biologists, it is important for a computer scientist working in the area of sequence alignment and sequence searching to be aware of the material in these books.

There have been several review papers in the past ten years about problems, algorithm, and software applicable to problems of sequence analysis in genetics and molecular biology. In 1983, Gingeras [115] surveyed the available software packages that performed statistical analysis of DNA sequence data. In the 1986 volume of *Annual Review of biophysics and biophysical chemistry*, W. Goad [116] reviewed algorithms for sequence alignment and sequence searching. The paper included sections on text processing methods in sequence analysis, the Needleman–Wunsch–Sellers alignment algorithms, local similarity algorithms, database searching algorithms, and the statistical significance of similar sequences. R. Nussinov [211] published the paper "Theoretical molecular biology: Prospectives and perspectives" in 1987, in which she discussed searching databases for similar sequences and searching for patterns common to many sequences. In a review paper that appeared in *Science* in 1988, C. Delisi [71] discussed the application of computers in molecular biology. Although only a few paragraphs of the paper deal directly with sequence alignment, there are many interesting ideas in the paper. In the forward to the 1989 special issue of the *Bulletin of Mathematical Biology* on molecular sequence analysis, M. Waterman [285] gives an introduction to the many papers that appear in that issue of the *Bulletin of Mathematical Biology*.

There are several databases that are frequently used when searching for similar sequences of nucleic acids or amino acids. In the paper "The EMBL data library", G. Cameron [49] described the data bases and services that EMBL provides. In similar papers, H. Bilofsky and C. Burks [38] and J. Moore & C. Burks [199] describe the databases that are maintained at GenBank, the genetic sequence databank.

In the 1987 papers by W. Gilbert [114] and L. Hood & L. Smith [135] arguments are presented in favor of completely sequencing the human genome. The paper by Hood and Smith advocates waiting for five to ten years while the sequencing technology develops before sequencing the human genome. Gilbert advocates starting the project now and believes that it could be completed by the year 2000. Another interesting project with plans to sequence an entire genome is the worm project. The 1989 paper by L. Roberts [228] describes the current status of the *Caenorhabditis elegans* project and the approach being taken to sequence the genome.

Jungck and Friedman [151] in 1984 compiled the first annotated bibliography that I have found in the area of computational biology [151]. In 1991 Barron, Witten, Harkness, and Driver [31] published "A bibliography on computational algorithms in molecular biology and genetics" and in the same year Bairoch [29] published "SEQANALREF: a sequence analysis bibliographic reference databank". Both Barron's and Bairoch's bibliography contain well over 1000 references, although neither contain abstracts or extensive comments. Both are available electronically to facilitate computer searches. I [134] have recently made available an annotated bibliography that contains about 350 papers, each with the bibliography or a summary of the paper that is available as a technical report or electronically.

In 1985, in an effort to introduce some of the computational problems in genetics to computer scientists, Friedland and Kedes [96] presented a paper in the *Communications of the ACM* that covered several computational questions that a geneticist might be faced with. They discuss the program MOLGEN and AI techniques that might be useful in solving some these questions.

2.2 Exact string matching

Exact string matching is a fundamental problem in computer science that has been addressed in many of the algorithm texts including [6, 239, 113, 57]. Since exact string matching algorithms are frequently used as important pieces of approximate string matching algorithm, I will review the exact string matching literature here.

A typical string matching problem is: given some text T of length n and a pattern P of length m find the each occurrence of P in T. The obvious brute force algorithm to solve this problem takes $O(n \cdot m)$ time in the worst case, but frequently takes time proportional $n + m$.

2.2.1 Sequential algorithms

In a paper in 1975, A. Aho and M. Corasick [4] describe an algorithm to find all occurrences of a set of strings, the pattern strings, in a text using a pattern matching machine. The pattern matching machine can be constructed in time proportional to the sum of the length of the pattern strings. The number of state transitions that occur when the pattern matching machine is given the text is shown to be less than $(2 \cdot |\text{text}|)$. The state transitions can be stored in a two dimensional array of size $((\text{numberofstates}) \times |\text{alphabet}|)$ so it is possible to simulate the machine in time proportional to the length of the text. A modification of this algorithm was given by J. Aoe [20].

In 1977, D. Knuth, J. Morris & V. Pratt [162] published an algorithm that finds all matches of a pattern in a text without backing up in the text using $O(|\text{pattern}|+|\text{text}|)$ operations. A finite state machine is built from the pattern so that when a mismatch is found the pattern can be moved ahead as far as possible in the text. In 1990, C. Consel and O. Danvy [55], gave a program refinement derivation of the Knuth Morris Pratt pattern matching algorithm.

The string matching algorithm that is frequently used in practice today was introduced by R. Boyer and J. Moore [45] in 1977. This algorithm, like the Knuth

Morris Pratt algorithm, uses time proportional to the length of the pattern and the length of the text. The Boyer Moore algorithm constructs a finite automata that compares the pattern with the text starting at the right end of the pattern and proceeding left until a mismatch is found. When a mismatch is found, the pattern is moved ahead in the text. Although in practice this algorithm is faster that the Knuth Morris Pratt algorithm it does require that the text be read backwards in some situations. Variations of the Boyer Moore algorithm have been given by R. Baeza–Yates [26] and T. Lecroq [178]. The papers by R. Schaback [236] and R. Baeza–Yates [27] show the expected time of the Boyer Moore algorithm is sublinear. Recently, a number of refinements to the Boyer Moore algorithm have been published [137, 261, 251, 142].

In 1987 R. Karp and M. Rabin [158] gave a randomized pattern matching algorithm. The basic idea used by the algorithm is that a fingerprint of a string can be computed in real time and will be much smaller than the original string. The fingerprint of a string is the string interpreted as a number modulo a large prime. The fingerprint of the pattern string can be compared with the fingerprints of substrings of the text string. In the worst case this algorithm takes $O(|\text{pattern}| \cdot |\text{text}|)$ time, although G. Gonnet and R. Baeza–Yates [118] show that the algorithm almost always runs in linear time.

Zvi Galil has published a series of papers on exact string matching that have resulted in optimal, measured in space and time complexity, string matching algorithms. In 1983, Z. Galil and J. Seiferas [106] developed an algorithm that simultaneously used $O(n)$ time and $O(1)$ space. This algorithm does have the drawback that in some situations it will need to re–read some of the input. M. Crochemore [60] constructed an algorithm that uses linear time and constant space during both the preprocessing phase and the scanning phase of the algorithm. In two papers, L. Colussi, Z. Galil, & R. Giancarlo [54] and Z. Galil [99], give tight upper and lower bounds on the number of comparisons needed to do string matching.

There have been a number of interesting generalized and hybrid string matching algorithms published. In 1987, K. Abrahamson [2] defined and gave an algorithm to solve the generalized string matching problem where each position of the pattern can match some set of characters from the alphabet. T. Eilam–Tzoreff & U. Vishkin [75] consider the problem of matching transformations of the pattern in the text instead of matching the pattern itself. A linear time, constant space algorithm by M. Crochemore and D. Perrin [62] used the ideas of the Knuth Morris Pratt and the Boyer Moore algorithms to process the pattern string automaton from both the left and right ends simultaneously. By sampling the text, R. Quong [223] developed an algorithm with fast average case behavior. If the alphabet is ordered, M. Crochemore [61] has developed an algorithm that finds all occurrences of a pattern in a text using linear time, constant space, and that does not need to preprocess the pattern.

2.2.2 Data structures

A suffix tree is a data structure that can be created in time proportional to the length of a string, A, that allows the question "Is B a substring of A?" to be answered in $O(|B| + \log |A|)$ time. The basic ideas of a suffix tree were introduced in 1973 by Weiner [290] in a paper that gave an algorithm to construct bi–trees, a data strucutre closely related to suffix trees, in linear time and space. That algorithm is presented in the Aho, Hopcroft, & Ullman text book [6] and the data structure is referred to as a position tree. In 1976 E. McCreight [193] introduced an algorithm to construct a suffix tree, a data structure closely related to the bi–tree and position tree that uses about 25 percent less space than previous algorithms. In 1986, A. Apostolico [21] devised an $O(\log n)$ time parallel algorithm to construct a suffix tree for a string of length n on a CRCW PRAM with $O(n)$ processors. The algorithm has the disadvantage of using $O(n^2)$ space although only $O(n \log n)$ of the cells need to be initialized.

In 1990, Manber and Myers [189] introduced the suffix array, a data structure that is three to five times more space efficient than the suffix tree. Using suffix arrays, the question "Is B a substring of A?" can still be answered in $O(|B| + \log |A|)$ time. In some cases a suffix array for a string of length n may take $O(n \log n)$ time to construct when a suffix tree for the same string could be constructed in $O(n)$ time. The expected time for the construction of a suffix array is $O(n)$.

2.2.3 Parallel algorithms

Galil and Vishkin have written a series of papers leading to optimal parallel algorithms for exact string matching. Each of the four following papers gives an optimal parallel algorithm, but the time required decreases and the number of processors increases in each successive paper. The final paper gives a constant time optimal parallel algorithm for exact string matching.

In 1984, Z. Galil [98] gave an algorithm that matches strings in $O(\log n)$ time using $O(n/\log n)$ CRCW PRAM processors. This time includes the preprocessing of the pattern string. One limitation to Galil's algorithm is that it requires the size of the alphabet to be fixed. In a 1985 paper, U. Vishkin [277] constructs a parallel algorithm for string matching that runs in $O(\log n)$ time using $O(n/\log n)$ CRCW PRAM processors, but does not require the size of the alphabet to be fixed. In 1990, Breslauer & Galil [47] reduces the time required by a CRCW PRAM for string matching to $O(\log \log n)$ using $O(n/\log \log n)$ processors. In his 1992 paper, Galil [100] gave an algorithm that reduced the time required by a CRCW PRAM computer with $O(n)$ processors to $O(1)$ to solve the exact string matching problem.

2.3 Approximate string matching and sequence alignment

Algorithms to do approximate string matching and sequence alignment using similar solutions have been developed independently by a number of different researchers in both computer science and biology. In this section I will try to follow the development of the approximate string matching in the computer science literature separately from the development of sequence alignment in the biology literature. Many of the algorithms discussed in this section are variations on a basic dynamic programming algorithm to do approximate string matching. The paper by W. Miller & E. Myers [196] gives an excellent presentation of the basic algorithm.

2.3.1 Computer science literature

There are several problems in computer science that are considered approximate string matching, including the string to string correction problem, the edit distance problem, the longest common subsequence problem and the k–mismatch problem. Several of these problems have similar dynamic programming solutions.

The string to string correction problem

In 1974, R. Wagner & M. Fischer [279] defined the string to string correction problem and gave a dynamic programming algorithm to solve it. The string to string correction problem is: given two strings, A and B, $m = |A|$, $n = |B|$, find the minimum number of operations needed to change A into B. The operations available are to change one character of one of the strings, to delete a character from one of the strings, and to insert a character into one of the strings. This problem is also called the edit distance problem. The algorithm Wagner & Fischer gave solves the string to string correction problem in $O(mn)$ time. In the following year, R. Lowrance & R. Wagner [184] extended the algorithm to include the operation of interchanging two adjacent characters in one of the strings. With this added operation, the problem can be solved using a similar dynamic programming algorithm that uses $O(mn)$ time. Both of the previous algorithm require $O(mn)$ space. In 1975, D. Hirschberg gave a linear space algorithm to solve the longest common subsequence problem that runs in $O(|A| \cdot |B|)$ time. The longest common subsequence problem is very similar to the string to string correction problem and the space saving ideas of Hirschberg can be applied to the algorithms of Wagner & Fischer and Lowrance & Wagner.

There have been several interesting results related to the string to string correction problem. W. Tichy [267], in 1984, defined another set of operations that allowed the string to string correction problem with block moves to be solved in time and space linear in the size of the input. The only operation available is to move a block of characters in one of the strings and the goal of the problem is to cover one of the strings with blocks from the other string. The algorithm given by W. Tichy runs in time and space that is linear in the size of the input. M. Maes [185] considered cyclic strings, strings with no ends, and defined the cyclic string to string correction problem. An algorithm was developed to find the minimum number of edit operations needed to transform one cyclic string into the other that used $O(nm \log m)$ time where n and m are the size of the cyclic strings. W. Masek & M. Paterson [191] in a 1980 paper gave an algorithm to compute the minimum cost of a sequence of edit operations to convert one string into another that used $O(n \cdot \max(1, m/\log n))$ time, $n > m$. To achieve this time bound the cost of each edit operation must be an integral multiple of a single positive real number and the alphabet must be finite.

In a 1976 paper C. Wong & A. Chandra [294] and later in a 1988 paper X. Huang [139] showed an $\Omega(nm)$ lower bound to find the minimum cost sequence of edit operations to convert one string of length n to another string of length m. The model of computation assumed that the operations on the characters in the strings were limited to tests of equality.

The longest common subsequence problem

A subsequence of the string $A = a_1 a_2 \cdots a_n$ is any string of characters $a_{i_1} a_{i_2} \cdots a_{i_k}$ where $i_1 < i_2 < \cdots < i_k$. The longest common subsequence between two strings A and B is the longest string that is a subsequence of both A and B. The problem of finding the minimum cost edit sequence to transform one string into another is the dual of the problem of finding the longest common subsequence of two strings. The solution to the longest common subsequence problem seems to be folklore in computer science [57]. The dynamic programming algorithms that were used to solve the longest common subsequence problem took $O(nm)$ time and space, where n and m were the lengths of the strings. In 1975, Hirschberg [132] gave an algorithm to solve the longest common subsequence problem in $O(nm)$ time and $O(n + m)$ space. In the following year, A. Aho, D. Hirschberg & J. Ullman [5] show that any algorithm to solve the longest common subsequence problem using a decision tree model of computation where each vertex represents a test of equality between two characters must use $\Omega(nm)$ comparisons if the size of the alphabet is not bounded.

A number of algorithms have been presented that improve the running time by restricting the problem in various ways. In 1977, J. Hunt & T. Szymanski [143] added the parameter r that is the number of ordered pairs of position in the strings that match. They give an algorithm that runs in $O((r+n)\log n)$ time. A modification to this algorithm was presented by S. Kuo & G. Cross [164] that improved the running time to $O(r+ns+\log n)$ where s is the length of the longest common subsequence. W. Hsu and M. Du [138], in 1984, added the parameters ρ, the length of the longest common subsequence, and s, the number of distinct symbols appearing in the two strings. They gave an algorithm to find the longest common subsequence of two sequences that uses $O(n \log s)$ preprocessing time and $O(\rho m \log(n/m) + \rho m)$ time. By showing the equivalence of the longest common subsequence and the edit distance problem, E. Myers [204] was able to construct an $O(nd)$ time algorithm to find the longest common subsequence where d is the length of the edit sequence to transform one string into the other. A variant of the algorithm that takes $O(n \log n + d^2)$ time was also presented. In 1992, A. Apostolico, S. Browne, & C. Guerra [22] discussed several algorithm to compute the longest common subsequence using linear space and, in favorable circumstances, sub–quadratic time. If the matrix used in the dynamic programming algorithm can be made sparse, D. Eppstein, R. Giancarlo, Z. Galil, & F. Italiano [82] have developed an algorithm using a linear, concave, or convex gap penalty function (defined in the following section) to take advantage of the sparsity of the matrix to reduce the running time.

Algorithms have been developed that limit the maximum edit distance to be considered between the pattern and substring in the text. This limit allows the algorithm to consider a much smaller number of possible alignments. In 1985, E. Ukkonen [272] gave an algorithm that, given k, the maximum edit distance to consider, constructs a finite state automaton to find the positions in the text that were at most an edit distance of k from the pattern. In the following year, G. Landau, U. Vishkin, & R. Nussinov [169] developed an algorithm to solve the above restricted edit distance problem in $O(m^2 + k^2 n)$ time. Z. Galil & K. Park [103] gave a new algorithm to solve this problem using $O(m^2 + kn)$ time.

Convex and concave gap penalty functions

Many of the algorithms that compute the minimum cost edit script required to convert one string into another penalize a sequence of k delete operations as k independent events. There are situations where it is desirable to treat a series of consecutive deletes as one event and penalize it by some function of the number of consecutive deletes. This function, the gap penalty function, may be any function, but constant, linear, affine, convex, or concave functions are usually used. Of particular interest to people aligning sequences are convex gap penalty functions, functions that eventually grow more slowly than linearly.

In a series of papers and technical reports, Z. Galil and his co–workers have developed algorithms that make clever use of the assumption that the gap penalty function is convex or concave [101, 79, 81, 83, 78, 76, 102, 80, 104, 105]. W. Miller and E. Myers [197] gave algorithms to compute the edit distance between two sequences using a convex or a concave gap penalty function. The algorithms run in $O(n^2 \log n)$ time and use $O(n)$ space. They also gave a variant of the algorithm that uses a piece–wise affine gap penalty functions.

The k–mismatch problem

Given two strings, A and B, $m = |A|$, $n = |B|$, $|A| < |B|$, the k–mismatch problem it to find all positions in B where A aligns with k or fewer mismatches. In a series of papers, G. Landau, U. Vishkin, & R. Nussinov [165, 166, 170] gave algorithms to solve the k–mismatch problem using $O(k(m \log m + n))$ time and $O(nk)$ space. In 1989, R. Grossi and R. Luccio [126] developed an algorithm using suffix trees that solved the k–mismatch problem. A. Bertossi, E. Lodi, F. Luccio, & L. Pagli [37] considered the k–mismatch problem when the length of the text is much larger than the length of the pattern. In this paper, they also consider mismatches that are dependent on the context of the pattern or the text. Algorithms are developed to solve these problems in $O(kn)$ time.

Other interesting approximate string matching algorithms

E. Myers & W. Miller [207] considered approximate matching of regular expressions. The problem is "given a sequence A and a regular expression R, find a sequence matching R whose optimal alignment with A is the highest scoring of all such sequences". The algorithm they give to solve this problem runs in $O(mn)$ time where $m = |A|$ and $n = |R|$. They give a number of related algorithms including one that uses an arbitrary, increasing gap penalty function and runs in $O(mn(m + n) + n^2 \log n)$ time.

String matching with a sequence of don't cares is a weak form of approximate string matching. U. Manber & R. Baeza–Yates [188] gave an algorithm to solve this problem using suffix arrays. Once the suffix array has been constructed the algorithm can find the number of occurrences of the pattern in $O(\log n)$ time where n is the length of the text. A list of all occurrences can be produced in $O(n^{1/4} + R)$ where R is the number of occurrences of the pattern in the text. J. Bradford [46] introduced an algorithm that encodes strings as binary codewords such that the hamming distance between the codewords is equal to the edit distance between a pair original strings. The cost of computing the codewords is high, but the cost of computing the hamming distance between two of the codewords is linear in the length of the codeword. In 1990, W. Chang and E. Lawler [51] improved the algorithm of Landau & Vishkin [167] and used it to construct algorithms that perform exact and inexact matching in sublinear expected time. In a technical report, P. Cull and J. Holloway [63] gave a divide and conquer algorithm to do approximate string matching with inversions and swaps that runs in $O(mn)$ using $O(m)$ work space where m and n, $m < n$, are the lengths of the strings. This algorithm is discussed in detail in Chapter 7.

E. Myers [205] introduced the consecutive alignments problem to allow the video screen to be updated on–line as the optimal alignment between strings is being computed. The problem is, given two strings A and B, find the best alignment for each suffix of A in B. Algorithms are presented that solve this problem in the same time as the standard dynamic programming algorithms used to align sequences. This algorithm allows incremental screen updates of the best alignment.

Parallel algorithms for approximate string matching

To solve the string to string correction problem defined earlier, R. Wagner & M. Fischer [279] compute the values of an $m \times n$ matrix. The dynamic programming algorithm to solve the string to string correction problem computes element (i, j) of the matrix from the elements $(i, j-1)$, $(i-1, j)$ and $(i-1, j-1)$ of the matrix. This permits the computation of an entire northeast southwest diagonal of the matrix in parallel. See chapter 3 for details of parallelizing the standard dynamic programming algorithm to compute the string to string correction algorithm.

In 1986, M. Veldhorst [276] described a parallel algorithm to solve a dynamic programming problem related to the dynamic programming algorithms used to solve the string to string correction problem. Although this algorithm could not be used to solve the problems we are interested in, it does contain the key ideas needed to parallelize the dynamic programming algorithms used to solve the string to string correction problem.

In 1986, Landau & Vishkin [167] gave a parallel algorithm to solve the k–mismatch problem in $O(\log m + n/\log n + k)$ time using $O(m^2 + n)$ processors. The text is length n, the pattern is length m, and k is the maximum number of mismatches allowed in the alignment. Landau & Vishkin [168] gave a parallel algorithm to solve the string to string correction problem when the number of edit operations is limited to k. The algorithm they give runs in $O(\log m + k)$ time using n processors. O. Ibarra, T. Pong, & S. Sohn [145, 146] give a parallel algorithms to solve the string edit problem and the longest common substring problem on the hypercube architecture. The algorithm uses $O(\log^2 n)$ time and constant space using $O(n^3/\log^2 n)$ processing elements of a SIMD hypercube. In 1992, O. Ibarra, T. Jiang, & H. Wang [144] gave an algorithm to solve the string edit distance problem using a linear array of m identical finite state machines in $O(n)$ time.

2.3.2 Biology literature

In 1984, M. Waterman [282] surveyed the problems of sequence comparison in biology. A number of topics were included in the discussion including sequence comparison, location of long matching subsequences, and comparison of several sequences simultaneously. In 1988 M. Waterman [284] again surveyed the problems of computer analysis of nucleic acid sequences. E. Tyler, M. Horton, & P. Krause [270] in 1991 examined each of the major algorithms used for sequence comparison and discussed the problems in molecular biology that each algorithm was most suited to solve.

Similarity matrices

Unlike the character matching that was discussed above in which characters either match or do not match, amino acids have physical and chemical properties that make some amino acids similar to one another and other dissimilar. To capture the similarity between amino acids, tables have been constructed to give the "cost" of replacing one amino acid with another [70]. These matrices, commonly referred to as PAM matrices, are further investigated in the 1985 paper by W. Wilber [291].

A similarity matrix can be used to favor sequence alignments in which amino acids have been replaced with similar amino acids over alignments in which amino acids have been replaced with dissimilar amino acids. S. Altschul [11] argues that certain matrices should be used when aligning individual sequences and other matrices should be used when searching the amino acid databases for sequences similar to a query sequence. P. Argos [23] gave a thorough procedure using PAM matrices to align protein sequences and to test the significance of the alignment.

Global sequence alignment

In 1970, Needleman & Wunsch [209] gave an algorithm to find the longest common subsequence of two amino acid sequences. Although a detailed algorithm is never explicitly stated, it is clear that the ideas of the dynamic programming algorithms presented in later papers appear in this paper. The algorithm of Needleman & Wunsch was modified slightly by Sankoff [231] and shown to take time and space proportional to the product of the length of the sequences being aligned. This paper introduced the matrix W and the equation

$$W(i,j) = \max\{W(i-1,j),\ W(i,j-1),\ W(i-1,j-1) + \delta(a_i,b_i)\} \qquad (2.1)$$

that is the basis for many of the dynamic programming algorithms. P. Sellers [240, 241] generalized the algorithm of Sankoff to use a variety of distance functions. Sellers' distance metrics and algorithms were further generalized by M. Waterman, T. Smith, & W. Beyer [289] to allow insertions and deletions of multiple characters. Waterman's algorithm requires $O(n^2m)$ time and $O(nm)$ space. In 1982, O. Gotoh [119] reduced the time required by Waterman's algorithm to $O(nm)$ by requiring the gap penalty function be linear. Waterman [281] gave an algorithm that used a convex gap penalty function and conjectured that it used $O(nm)$ time. M. Fredman [94] gave a version of the dynamic programming algorithm that penalizes each gap by a constant, regardless of the length of the gap. The algorithm of Gotoh [119] was improved by S. Altschul [14] in 1986 to use any affine gap penalty function. This allowed the separation of the cost of breaking the sequence and the cost associated with the length of the gap. In 1990, O. Gotoh [122] gave an algorithm that can simulate a convex gap penalty function with a piecewise linear gap penalty function. Gotoh's algorithm runs in $O(lmn)$ time where l is the number of linear pieces that make up the gap penalty function.

The effects of choosing various affine gap penalty functions used by the dynamic programming algorithm to align sequences were investigated by W. Fitch & T. Smith [91] in a 1983 paper. A similar study was done in 1985 by D. Feng, M. Johnson, & R. Doolittle [88]. In 1991, J. Spouge [256] presented empirical results of aligning sequences using unweighted end–gaps, affine gap penalty functions, constant gap penalty functions, and concave gap penalty functions.

In a previous section we saw that the string to string correction problem could be sped up if we limited the number of edit operations allowed to some constant, k. A similar idea is given in a paper by M. Waterman [280] in 1983. That paper describes an algorithm to find optimal alignments when the alignments are restricted to those using at most k insert, delete, or change operations. Papers by J. Fickette [89] and E. Ukkonen [271] presented similar ideas. In 1985, the ideas of Fickette and Ukkonen were extended by J. Spouge [255] to search for the optimal alignment from both the left and right ends of the sequences simultaneously.

In the late 1980's the biologists and computer scientists became more aware and interested in each others' work. The 1975 result of Hirschberg [132] that gave an algorithm for the longest common subsequence using linear space was introduced and applied to the sequence alignment problem by E. Myers & W. Miller [206] in 1988. A 1990 paper by S. Wu, U. Manber, G. Myers, & W. Miller [295] applied their algorithms to the problem of sequence alignment. The algorithms of G. Landau, U. Vishkin, & R. Nussinov (and many other algorithms) to solve the k–differences problem and the approximate string matching problem were given in volume 183 of the methods in enzymology series [171].

There have been several algorithms given to align or compare sequences that are not related to the dynamic programming algorithms. In 1982, J. Felsenstein, S. Sawyer, & R. Kochin [87] introduced the idea of using the fast Fourier transform to construct a fast algorithm to measure the similarity of sequences. Their algorithm computes the maximum number of exact matches between two nucleic acid sequences for any alignment with no gaps. Four indicator sequences are built for each sequence, one for each base. The indicator sequence for base X has a 1 at position i when base X appears at position i of the sequence. Position j of the product of the indicator sequences for base X has the number of exact matches of the X base when one sequence is aligned at position j of the other sequence. By using the FFT to do the multiplications the algorithm runs in $O(n \log n)$ time.

A maximum likelihood method has been developed to align sequences when explicit transition probabilities have been obtained to create a statistical model of DNA evolution. In 1986, M. Bishop & E. Thompson [41] introduced the optimal sequence alignment problem under a probabilistic model of DNA evolution. A similar model of evolution is introduced by J. Thorne, H. Kishino, & J. Felsenstein [266] in 1991. In a paper submitted to the *Journal of Molecular Evolution*, J. Thorne, H. Kishino, & J. Felsenstein [265] extend their algorithm to address multi–base insertions and deletions.

In 1990, two papers by Allison, Wallace, & C. Yee [8, 7] described a method to measure the similarity of sequences based on the minimum message length encoding of the pair of sequences. This method requires that the method used to encode the sequences (the model of evolution) and the encoded sequences be included in the measure of the message length. Using this scheme it is possible to consider different models of evolution to explain the differences between sequences.

In 1984, W. Wilbur & D. Lipman [293] developed a method to measure the similarity of two sequences based on the context of the nucleic acids or amino acids being compared. After they define the distance, they give a dynamic programming algorithm to compute the similarity of the sequences. In the same year, P. Sellers [244] gave an algorithm that measures the mismatch density of a region of the match and then used this measure to construct a dynamic programming algorithm. D. Davison & K. Thompson [69] developed an algorithm to measure the similarity between local regions of sequences instead of the entire sequence. This type of algorithm can be used when reconstructing a sequence from shotgun sequencing data. D. Torney, C. Burks, D. Davison, & K. Sirotkin [268] introduced a measure of dissimilarity between sequences that is used to search for similar sequences.

Most of the algorithms that we have considered in this section have used only the primary sequence information to measure the similarity between sequences. In 1987, M. Gribskov, M. McLachlan, & D. Eisenberg [125] gave a method called profile analysis that is based on the three dimensional structure of the protein as well as the primary amino acid sequence.

Local subsequence alignment

In the previous section we considered the problem of aligning two sequences. Frequently it is also interesting to search for parts of a sequence that are similar to other sequences, or parts of other sequences. In 1979 and 1980, P. Sellers [242, 243] gave an algorithm to list all pairs of intervals, one from each of two sequences, that are locally similar to one another. This algorithm is also presented in a very readable format in the book chapter by B. Erickson [84]. A variant of Sellers' algorithm was given by W. Goad & M. Kanehisa [117] that used a more sensitive distance measure so that the number of similar pairs of subsequences found by the algorithm was reduced.

Sellers' algorithm minimizes the dissimilarity between the two subsequences. In 1981, T. Smith and M. Waterman [253] gave an algorithm that maximizes the similarity between the subsequences and allows for arbitrarily long gaps in the alignment. This algorithm was extended by M. Waterman and M. Eggart [287] to find all non–intersecting similar subsequences that score above a given threshold. In 1992, M. Schoniger & M. Waterman [238] further extend the algorithm to find non–intersecting inversions.

In 1986, S. Altschul & B. Erickson [12, 13] introduced a new measure of subsequence similarity and an algorithm that uses this nonlinear similarity function to find subsequence alignments. The similarity measure that they use is based on the logarithm of the probability of finding mismatches in a specified length of the alignment. This algorithm is examined in a paper by C. Lawrence [175] and several examples of its use are given.

Parallel algorithms

The dynamic programming algorithm used for many of the sequence alignment problems lends itself easily to parallel implementation. In 1989, N. Core, E. Edmiston, J. Saltz, & R. Smith [56] gave a parallel implementation of the dynamic programming algorithm to align two sequences. Their implementation uses $O(nm)$ space and $O(n)$ time using $O(m)$ processors, $n \geq m$. The space requirement of the algorithm limit the practicality of this algorithm. In the same year, X. Huang [140] gave a parallel version of the dynamic programming algorithm for sequence comparison that runs in linear time and uses linear space. X. Huang,

W. Miller, S. Schwartz, & R. Hardison [141] gave a parallel algorithm that solves the local similarity problem using only linear space.

The problem of sequence comparison has been used by a number of researchers to demonstrate the ease of implementation of an algorithm using their parallel computer language. In 1989, A. Singh & R. Overbeek [248] implemented the standard dynamic programming algorithm using the parallel programming language unity. The algorithm has also been implemented by D. Sittig, D. Foulser, N. Carriero, G. McCorkle, & P. Miller [249] to demonstrate C–Linda.

Multiple sequence alignment

Biological molecules such as DNA, RNA, and proteins are frequently used to construct phylogenetic trees. A phylogenetic tree describes the evolution of a group of organisms or molecules. When these molecules are used, they must be aligned so that homologous positions in the molecules are aligned with one another. A particular position in a group of molecules is homologous if that position has evolved in each of the molecules from a single position of a common ancestor molecule. Homology in biological molecules if often inferred from similarity between the molecules. Several algorithms have been developed to align several sequences so that similar regions will be aligned with one another. In 1988, T. Friedemann [95] published a review article discussing seven multiple sequence alignment programs.

Another reason to perform multiple sequence alignments is to improve the sensitivity of a sequence similarity search. In 1990, S. Altschul & D. Lipman [17] described an algorithm to search amino acid databases for several sequences that are similar to the query sequence. By searching for more than one sequence, relationships that might otherwise be obscured by noise can be found.

The dynamic programming algorithm to align two sequences can be extended to align three or more sequences. This approach generally leads to algorithms that require time proportional to the product of the length of the sequences begin aligned so the dynamic programming algorithm for aligning multiple sequences quickly becomes impractical as the number of sequences increases. In 1985, M. Murata, J. Richardson, & J. Sussman [203] extended the dynamic programming algorithm to simultaneously align three sequences. Their algorithm runs in time proportional to the product of the length of the three sequences. In the following year, O. Gotoh [120] gave a dynamic programming algorithm to align three sequences that also produces a traceback of the edit operations used to create the alignment. It is shown that the time required to produce the traceback is small compared to the time required to compute the alignment. In 1986, E. Sobel & H. Martinez [254] presented an algorithm to compute near optimal alignments of two or more sequences by using the dynamic programming algorithm to align small segments of the sequences instead of the individual characters.

The dynamic programming approach to simultaneously align several sequences is not practical in many situations. An alternative is to cluster the sequences into small groups and then use the dynamic programming algorithms to align the sequences in the groups. In 1988, F. Corpet [58] and H. Carrillo & D. Lipman [50] gave algorithms that compute all of the pairwise alignments using the dynamic programming algorithm and then use the pairwise alignments to create a hierarchical clustering of the sequences. D. Lipman, S. Altschul, & J. Kececioglu [182] extend and implement the algorithm of Carrillo & Lipman. A similar method was given by S. Subbiah & S. Harrison [260] in 1989. S. Altschul & D. Lipman [16], in 1989, show that the multiple sequence alignment problem can be viewed as a circular programming problem, a generalization of the linear programming problem. It was not clear if Khachian's algorithm or Karmarkar's algorithm could be used to solve the problem in polynomial time. M. Berger & P. Munson [36] have developed an iterative randomized algorithm to align multiple sequences. Their algorithm divides the sequences into two groups and then uses a variant of the dynamic programming algorithm to align the two groups. Each group of sequences is treated as a single sequence during the alignment process. This process is iterated; each iteration improves the alignment of the groups.

The definition of a gap penalty when aligning multiple sequences is not as intuitive as the definition that applies when aligning two sequences. In a 1989 paper, S. Altschul [10] considers a new definition of the gap penalty function that uses information gathered from the results of pairwise alignments of the sequences to be aligned. In 1990, O. Gotoh [121] shows that pairwise optimal alignments between three or more sequences may not be consistent. It is argued that characters that lead to the inconsistency are usually distributed in clusters. Using this idea, an algorithm is developed that selects the consistent regions of the pairwise sequence alignments to construct the global alignment.

Due to the computational demands of computing the optimal alignment of multiple sequences, several heuristics have been proposed. A method given by M. Johnson & R. Doolittle [148] uses a window that restricts attention of the alignment algorithm to a small portion of the sequences. This window is moved along the sequences and gaps are inserted to align the sequences within the window. Similar methods are given by M. Waterman, R. Arratia, & D. Galas [286] and M. Waterman [283]. D. Bacon & W. Anderson [25] present a heuristic to find a good alignment between several sequences by scoring segments of the sequences using a similarity matrix. The heuristic does not handle gaps other than to align segments of the sequences in various positions. A. Landraud, J. Avril, & P. Chretienne [172] describe a method to align several sequences by first constructing a library of frequently occurring substrings that appear in the sequences and then searching for regions in the sequences where a majority of the sequences contain the same substring. This region is used to anchor the sequences and process is repeated with the subsequences to the left of the anchor point, and again with the subsequences to the right of the anchor point.

Visualization of alignments

Sequence similarity is frequently measured by a numeric value as computed by some algorithm. It is possible to represent the sequence visually in ways other than a sequence of alphanumeric characters. Nucleic acid sequences are frequently represented as four distinct colors when sequences are being aligned by hand to take advantage of the human visual pattern recognition abilities.

In 1986, M. Gates [112] used a two dimensional representation of DNA to make tandem repeats and other features obvious. The sequence of DNA is plotted on a two dimensional grid, one axis designated the A–T axis, the other axis designated the C–G axis. Assuming that the current position on the grid is (x, y), if the next character in the DNA sequence is an A then a line is drawn to position $(x + 1, y)$. If the next character of the DNA sequence was a T then a line is drawn to position $(x - 1, y)$ on the grid. Similarly for G and C. Using this type of display for DNA sequences causes certain structures, especially repeating structures, of the sequence to become immediately obvious.

In 1990, H. Jeffrey defined a representation of DNA sequences related to the chaos game or Sierpinski triangle. Given a square with each vertex labeled with one of the four bases and a current position of (x, y) within the square, move half of the distance to the corner of the square whose label matches the next character in the DNA sequence and place a dot at the new position. This procedure is repeated for each base in the sequence. It is shown that visible patterns represent both global and local patterns simultaneously.

Markov chain analysis and frequency analysis

Markov chain analysis of DNA sequences can be used to identify sequence similarities that may be difficult to find with other methods. In 1986, B. Blaisdell [42] presented a method to classify DNA sequences based on the homogeneity of their Markov chain representations. It was found that almost all of the sequences examined required at least a first order Markov chain for their expression. This method has some advantages over the dynamic programming algorithms, it does not require any preliminary alignment of the sets of sequences and it easily measures the global similarity between sets of more than two sequences. This method does not discriminate between closely related sequences well and does not produce an alignment of the sequences. H. Almagor [9], P. Garden [109], and others have also modeled DNA with Markov processes.

A number of people, including B. Silverman [247], D. Benson [34], and E. Cheever [52] have used Fourier transforms to measure similarities in sequences of DNA.

Significance

The algorithms that have been discussed to align sequences will generally align any sequences that are given to them, whether they are related to one another or not. Papers have been published that examine the statistical significance of the alignments produced. As early as 1970, J. Haber & D. Koshland Jr. [129] developed a procedure that grouped amino acids into groups with similar chemical properties and used these groups to test the predictions of relatedness between proteins. The idea of testing the significance of an alignment between two sequences by using a Monte–Carlo technique of sampling the sequences was introduced in 1973 by D. Sankoff & R. Cedergren [232]. They compared the alignment of the original sequence to alignments of random sequences with the same base composition as the original sequences. If the alignment score of the original sequences was statistically different from the alignment scores of the random sequences, the alignment was considered to be significant. In 1983, W. Fitch [90] extended these results by maintaining the nearest neighbor frequencies of the dinucleotides. It is shown that maintaining the nearest neighbor frequencies can affect the probability of observing a significant alignment. An algorithm given by M. Waterman, R. Arratia, & D. Galas [286] can be used to confirm or deny an alignment of sequences is different from an alignment of random sequences. More rigorous treatments of the significance of sequence alignments have been given by S. Karlin & G. Ghandour [154], I. Rinsma, M. Hendy, & D. Penny [226], and S. Karlin & S. Altschul [153]. In a recent paper, R. Mott [200] described a Maximum–likelihood algorithm to test the significance of sequence alignments.

2.3.3 Sequence analysis

There are several questions besides sequence alignment between two or more sequences that are related to approximate string matching. In a series of papers S. Karlin, G. Ghandour, F. Ost, & S. Tavare [155], S. Karlin, M. Morris, G. Ghandour, & M. Leung [156], and S. Karlin, M. Morris, G. Ghandour, & M. Leung [157] give algorithms to find the longest direct repeat, the highest frequency direct repeat of length k, dyad symmetries of length l, and other sequence patterns.

Biologists have described models of sequences for antigenic sites, amphipathic structures, globular proteins, and others. One sequence could code a number of these structures and knowing the structures being coded for along the sequence can be important in determining the three dimensional structure and function of a molecule. In 1989, Auger [24] gave algorithms to construct a composite model of a sequence.

2.4 Sequence searching

The algorithms that we have discussed to measure sequence similarity or to align sequences can be used to search databases of sequences for sequences that are close to the query sequence. Because of the size of the databases, algorithms such as the dynamic programming algorithms that generally take time proportional to the product of the length of the sequences being compared become impractical on all but the smallest databases. Many algorithms have been developed to search entire sequence databases for sequences similar to a query sequence and a few of the algorithms are presented below. A review of sequence database searching algorithms was given in 1985 by D. Davison [68].

2.4.1 Sequential algorithms

An algorithm given in 1983 by W. Wilbur & D. Lipman [292] finds all k–tuple matches between the two sequences of length m and n being compared. The k–tuple matches are placed in an m by n matrix and any diagonal of the matrix that contains many k–tuple matches is considered as a possible match. This process is repeated for each sequence in the database. If k is chosen properly, the time required by the algorithm will be related to the sum of the lengths of the sequences being compared instead of the product of the sequence lengths. If the sequences become too long then the word size of the computer limits the size of k and the time the algorithm requires increases. Variants of this algorithm have been presented by D. Lipman & Pearson [183], W. Pearson & Lipman [216], W. Pearson [215].

B. Orcutt & W. Barker [214] gave an algorithm to quickly search an amino acid database by preprocessing the database. An index of the locations in the database of each tri–peptide is created. Given a query sequence, to search for a similar sequence in the database the location of each tri–peptide of the query sequence is looked up in the database index and any regions in the database with a concentration of tri–peptide matches is considered as a possible similar sequence.

An algorithm presented by C. Lawrence, D. Goldman, & R. Hood [176] builds directly from the algorithms of Altschul & Erickson [13] and Wilbur & Lipman [292]. A similarity measure, $s(l, c)$ inspired by [13], is the number of standard deviations the alignment of length l with c mismatches is above a chance alignment of length l. Many of the values of $s(l, c)$ are precomputed so that they do not need to be repeatedly computed during the alignment procedure. This measure of similarity was combined with the substring searching ideas of Wilbur & Lipman [292] to create a quick and sensitive sequence searching algorithm.

In a 1990 paper, D. Brutlag, J. Dautricourt, S. Maulik, & J. Relph [48] gave a protein database searching algorithm that used PAM matrices to penalize amino acid mismatches based on the physical and chemical differences between the amino acids. The algorithm that they present also allows the matching of k–tuples instead of individual amino acids to speed the search process.

S. Altschul, W. Gish, W. Miller, E. Myers, & D. Lipman[15] describe the basic local alignment search tool (BLAST) algorithm to rapidly search DNA and protein databases. This algorithm introduces a measure of local similarity, the maximal segment pair (MSP) score, and an algorithm to approximate it. Using the MSP allows a rigorous mathematical analysis of the significance of the alignments found.

2.4.2 Parallel and distributed algorithms

Using a large vector computer to compute diagonals of the matrix used in the dynamic programming algorithm in parallel, O. Gotoh & Y. Tagashira [123] were able to search large databases for query sequences of DNA or protein.

Using MIMD computers to search the DNA or protein databases for a query sequence is simply supplying each of the processors with sequences from the database to compare with the query sequence. Each processor can independently compute the similarity scores of the query sequence and the sequences from the database. In 1987, A. Coulson, J. Collins, & A. Lyall [59] gave an algorithm to search large DNA and protein databases using the dynamic programming algorithm. The algorithm was implemented on the distributed array processor (DAP), a 64 × 64 array of processors. Programs have been written by A. Deshpande, D. Richards, & W. Pearson [72] for the Intel hypercube to search biological sequence databases for a query sequence using either the fasta algorithm of Pearson & Lipman [216] or the dynamic programming algorithm. Each processor is give a set of sequences from the database to compare with the query sequence and reports back the alignment scores of the sequences. The fasta algorithm has also been used by P. Miller, P. Nadkarni, & N. Carriero [194] to demonstrate the parallel computer programming language, Linda.

In 1991, G. Barton [32] developed a system to distribute the search of

DNA and protein sequences over several computers connected to a network that share a common filesystem. The algorithm used to compare individual sequences is the dynamic programming algorithm, but any algorithm could be used. The sequences from the database are distributed to individual computers to compare with the query sequence and the scores reported back to a central computer.

2.5 Motifs

There are small patterns in DNA and protein sequences that may indicate various characteristics of the DNA or protein sequence, such as the start of a coding regions or three dimensional structure. A frequently used example is the promoter sequences in *Escherichia coli*. A promoter sequence specifies the initiation site of transcription. There are two regions upstream of the transcription start site in *E. coli* that are highly conserved. Researchers are interested in automatic detection of similar patterns or motifs that indicate other functions and work in other organisms. These motifs are frequently small, less than twenty base pairs in length.

R. Harr, H. Haggstrom, & P. Gustafsson [131], in 1983, developed an algorithm that, when given a motif, would search a sequence for positions that match the motif. Another algorithm to search for a specified pattern was given the following year by R. Abarbanel, P. Wieneke, & E. Mansfield [1].

D. Galas, M. Eggert, M. Waterman [97] developed an algorithm to find previously unknown patterns in a set of sequences by searching for subsequences of size k that differ by at most w positions. The algorithm was used to discover promoter sites in bacterial sequences. In 1990, R. Smith & T. Smith [252] developed an algorithm to solve a similar problem using a clustering algorithm.

There have been several other interesting methods employed to specify and find motifs in sequences. In 1990, A. Alexandrov & A. Mironov presented some ideas for methods to learn, while being directed, a recognition matrix or discrimination vector to recognize *E. coli* promoter sites. The result of the learning phase is a vector that can be applied to *E. coli* sequences that have not previously been seen and predict the position of promoter sites in the sequence. G. Barton & M. Sternberg [33] introduced the flexible pattern structure in 1990. This structure is constructed from alternating character specifications and gap specifications. A character specification is a list of characters that can appear at the designated position in the pattern. Associated with each character is a weight so that site specific preferences may be specified. A gap specification is a definition of the length of the gap that may appear between the previous and the following character specifications. Algorithms are presented that used this data structure to search sequence for positions of the pattern or motif. A. Milosavljevic [198], in a PhD thesis, investigated the use of minimal length encoding to discover new motifs.

A. Danckaert, C. Chappey, & S. Hazout [67] proposed an interesting algorithm using the Fibonacci series to find matches of increasing size. Initially all matches of length F_i and F_{i+1} are computed. To find matches of length F_{i+2}, any pair of matches from the list of matches of length F_i and F_{i+1} that are adjacent are "joined" to be a match of length F_{i+2}. It is possible that a match of length F_i and a match of length F_{i+1} are not exactly juxtaposed, but could still be considered an approximate match of length F_{i+2}. This algorithm, the size leap algorithm, is used to construct the longest common motifs from a molecular sequence set. P. Taylor, P. Rosenberg, & M. Samsonova [263] have developed an algorithm to search for motifs by enumerating all possible strings that are consistent with the motif being searched for. For each possible motif pattern, the sequences being examined are compared at every position with this motif pattern and a similarity score is computed for each position. This exhaustive algorithm will only be useful for small motifs, less than ten to twelve DNA bases with the computers available today.

People have used neural networks in an attempt to automatically recognize promoter sequences. In 1991, K. Abremski, K. Sirotkin, & A. Lapedes [3] have selected informative base positions that distinguish between promoter and non–promoter sequences and then use those base positions to train both Perceptron nets and back propagation networks with hidden units. It is claimed that both methods resulted in the successful prediction of new promoter sequences more than 96.9% of the time. In the same year R. Rarber, A. Lapedes, & K. Sirotkin [85] reported on using a neural network to find open reading frames (ORF's) by looking at dicodons instead of individual codons. Their results suggest that the relationship between adjacent codons is important in exons and less important in introns.

2.6 Hardware for approximate string matching

The size of the DNA and protein databases requires that they be stored on secondary storage in all but the largest of computers. This means that the algorithms that are used to search the entire database may need to read the database from the disk. People have been investigating the possibility of constructing hardware that could be added to the disk controller to "screen" the database so that only the sequences that are sufficiently close to the query sequence are actually read into the computer.

S. Pramanik & C. King [222] present hardware that can perform exact pattern matching by using a set of cells, each capable of exactly matching a pattern. N. Tewari & M. Wagh [264], in 1986, describe a bit–sequential array for pattern matching problems. Using hardware that is proportional to the size of the pattern, strings matching the pattern can be found in time proportional to the size of the text or database being searched. A board that can be installed in a Sun computer has been developed that can be loaded with a pattern and a stream of characters can be passed by it. Each match between the pattern and the characters streaming by the pattern will be reported [227]. A simulator has been constructed by W. Isenman & E. Shasha [147] that compares strings with variable length don't cares. Using this simulator, a piece of hardware has been developed to do string matching as quickly as the disk can read characters.

The dynamic programming algorithms can be implemented in hardware since each node in the matrix only needs to communicate with three other nodes and we only need to remember two diagonals of the matrix to compute the next diagonal. In 1989, A. Mukherjee [202] gave a systolic algorithm to find the length of the longest common subsequence of two strings that runs in time proportional to the sum of the lengths of the strings being compared. J. Smith [250], in 1991, designed, implemented, and tested SEQWARP, a systolic array that can perform the Smith–Waterman dynamic programming algorithm, as a masters thesis.

2.7 Shortest common superstrings and sequence overlaps

In 1980, F. Sanger, A. Coulson, B. Barrell, A. Smith, & B. Roe [230] developed a method to sequence long stretches of DNA that is frequently referred to as shotgun sequencing. Many identical copies of the DNA to be sequenced are cleaved by one or more restriction endonucleases. These enzymes cleave the DNA sequence at each occurrence of a specific six to eight base subsequence. This results in a multiset of DNA fragments that is not ordered and essentially sampled at random to be sequenced. The original DNA sequence is reconstructed by finding overlaps between the DNA fragments that have been sequenced. There is a similar problem in computer science called the shortest common superstring problem. Garey & Johnson [110] define the shortest common superstring problem as

> INSTANCE: Finite alphabet Σ, finite set R of strings from Σ^*, and a positive integer K.
> QUESTION: Is there a string $w \in \Sigma^*$ with $|w| \leq K$ such that each string $x \in R$ is a substring of w, i.e., $w = w_0 x w_1$ where each $w_i \in \Sigma^*$?

Both the biological and computer science versions of the this problem are covered thoroughly in the PhD thesis of J. Kececioglu [159].

A 1980 paper by J. Gallant, D. Maier, & J. Storer [107] showed that the shortest common superstring problem and related problems are $\mathcal{NP}$–complete. In 1983, J. Gallant [108] showed that the problem of reconstructing shotgun sequence data from sequence fragments is computationally intractable to solve exactly.

In 1988, J. Tarhio & E. Ukkonen [262] gave a greedy heuristic to construct an approximate shortest common superstring using the Knuth Morris Pratt string matching algorithm and the greedy heuristic for finding the longest Hamiltonian path in a weighted graph. The heuristic ran in $O(mn)$ time where m is the number of strings and n is the sum of the length of the strings. The heuristic was guaranteed to find a common superstring that had at least 1/2 of the compression of the shortest common superstring. Compression is the difference between the sum of the length of the fragments and the length of the common superstring. J. Turner [269] improved the heuristic to run in $O(n \log m)$ time with the same performance guarantee. In the following year, E. Ukkonen [273] further improved the algorithm to run in $O(n)$ time with the same performance guarantee. In 1990, M. Li [181] gave the first algorithm guaranteed to find a common superstring with length that is no more than $O(s \log s)$ where s is the length of the shortest common superstring. In 1991, A. Blum, T. Jiang, M. Li, J. Tromp, & M. Yannakakis [43] showed that the heuristics of J. Tarhio & E. Ukkonen [262], J. Turner [269], and E. Ukkonen [273] actually found common superstrings that were at most four times the length of the shortest common superstring. They also gave a modification of the algorithm that was guaranteed to find common superstrings that were at most three times the length of the shortest common superstring. The similar problem of finding for all pairs of strings, the longest suffix/prefix match is solved optimally by D. Gusfield, G. Landau, & B. Schieber [128].

In 1990, D. Foulser [93] gave a variant of the shortest common superstring problem, the unique common superstring problem and an algorithm to solve it in time proportional to the size of the input. In 1983, H. Peltola, H. Soderlund, J. Tarhio, & E. Ukkonen [217] defined the shortest approximate superstring problem.

In this problem the superstring needs only to contain approximate copies of each string in the set instead of exact copies. P. Cull & J. Holloway [64, 66, 65] have considered a similar problem and constructed an algorithm based on suffix arrays to solve the problem.

In 1967, M. Shapiro [246] gave an algorithm specifically designed to reconstruct protein and RNA sequences from short fragment data. A complete system to handle the data produced by shotgun sequencing is given by R. Staden [257]. Similar systems have been described by J. Clayton & L. Kedes [53], H. Peltola, H. Soderlund, & E. Ukkonen [218], and R. Grymes, P. Travers, & A. Engelberg [127]. Probabilistic analysis of the problem has been carried out by L. Barnett [30] and by J. Beran–Koehn & W. Gillett [35].

2.8 Gene rearrangement and inversions

In 1989, D. Sankoff & M. Goldstein [234] introduced a probabilistic model of genome shuffling. This was further considered in the 1990 paper by D. Sankoff, R. Cedergren, & Y. Abel [233]. A basic assumption frequently made when considering gene sequence evolution is that it can be modeled with the elementary operations of inserting a base, deleting a base, and changing a base. Typically the sequence comparison algorithms try to find an alignment of the sequences using a minimal number of these operations. These papers proposes another model of genetic evolution where not only do these local operations occur, but larger blocks of the genome are rearranged.

The dynamic programming algorithms that are frequently used to align sequences obey the non–intersection property, that is the order of the characters in a sequence will not be changed as the sequences are aligned. Thus the dynamic programming algorithms will not easily be well suited to align sequences with inversions or rearrangements.

There have been a few attempts to include inversion as an operation in the dynamic programming algorithms. In spelling error detection and correction applications, the operation of reversing two adjacent characters is important. R. Lowrance & R. Wagner [184] give, in 1975, an extension to the string to string correction problem that allows the operation of interchanging adjacent characters. An interesting bibliography on spelling error detection and correction was written by J. Pollock [221]. In 1992, M. Schoniger & M. Waterman [238] described a dynamic programming algorithm to find alignments of DNA sequences using the operations of substitution, deletion, insertion, and non–intersecting inversion. Although this algorithm does not completely eliminate the non–intersection property since block rearrangements are not allowed, it does allow sequence alignments with non–intersecting inverted segments.

In 1984, W. Tichy [267] defined the string to string problem with block moves where the only operation available was to move a block of characters from one string to cover a portion of the other string. The goal of the problem was to completely cover one string with blocks from the other string. This problem and its solution are not restricted by the non–intersection property since there is no restrictions placed on where the blocks may be placed relative to one another.

2.9 Repeated substrings

There are several problems in the analysis of molecular sequences than can be expressed as finding repeats in strings. When aligning sequences or searching for similar sequences, the first step may be to find all substrings that appear in both sequences being examined. The problem of finding the two dimensional structure of RNA requires searching for inverted repeats in the sequence. The RNA sequence and its inversion can be concatenated and then searched for repeated substrings.

In 1980, L. Jones [150] gave an algorithm and program to solve the problem: given a text and an integer k find all substrings in the text of length at least k and report how frequently they occurred. In 1982, J. Dumas & J. Ninio [74] gave an algorithm to find all common subsequences in a sequence. They then use this algorithm to predict RNA secondary structure. In the same year M. Main &

R. Lorentz [187] gave an $O(n \log n)$ algorithm based on the Knuth Morris Pratt pattern matching algorithm to find the location of each repetition in a string. A repetition is a non–empty string of the form xx. In 1983, H. Martinez [190] gave an algorithm based on sorting the substrings of a sequence to find all of the repeats. In the same year R. Nussinov [210] gave algorithms to search for exact nucleotide sequence repeats.

By knowing the expected length of the longest common substring of two unrelated strings it is easier to judge the significance of long common substrings found when comparing DNA and protein sequences. In 1990, R. Mott, T. Kirkwood, & R. Curnow [201] gave an accurate approximation of the distribution of the length of the longest expected common substring between two random DNA sequences.

2.10 Other interesting and related problems

There are many other areas that are related to searching for sequences in DNA and protein databases and aligning similar genetic sequences. Several of the following problems contain approximate string matching as a subproblem. Frequently the people working in the area have come up with their own novel solutions to the problems.

2.10.1 Tertiary structure

The problem of determining the tertiary structure of a protein given only the primary DNA sequence is important since the relative cost of directly finding the three dimensional structure of a protein is much higher than inferring it given the primary sequence. This is an extraordinarily difficult computational problem, particularly since the biology is not understood well enough to supply reliable rules.

In 1987, R. Lathrop, T. Webster, & T. Smith [174] described Ariadne, an

AI blackboard system to generate possible three dimensional protein structures when given a transfer RNA along with some other information. M. Zuker & R. Somorjai [297], in 1989, applied the standard dynamic programming techniques to align proteins in three dimensions. Neural networks have been used to try to recognize subsequences coding for particular secondary structures in proteins by P. Stolorz, A. Lapedes, & Y. Xia [258] in 1991. In the same year, J. Garnier & J. Levin [111] published a paper titled "The protein structure code: What is its present status?" wherein the current methods of prediction of the three dimensional structure of proteins are reviewed.

2.10.2 Restriction map construction

Before large segments of DNA are sequenced, landmarks in the sequence are found by constructing a restriction map. A restriction map is constructed by digesting the DNA with a restriction enzyme and recording the lengths of the fragments created. The DNA is digested with a second, different, restriction enzyme and the lengths of the fragments are recorded. Finally, the DNA is digested with both restriction enzymes and the lengths of the fragments are recorded. With the fragment length data it is possible to construct a map of the DNA with the positions where each of the restriction enzymes cuts the DNA. This problem is complicated by the fact that the fragment length data are frequently only approximate.

In 1983, W. Fitch, T. Smith, & W. Ralph [92] and 1988 M. Krawczak [163] give algorithms to construct a restriction enzyme map given the fragment length data. W. Miller, J. Barr, & K. Rudd [195] give an algorithm that, when given a restriction map and a smaller probe map, finds the position in the restriction map that the probe map fits best. In 1986, M. Waterman & J. Griggs [288] use interval graphs to represent restriction maps. With this representation it is possible to use the linear time algorithm of K. Booth & G. Lueker [44] to construct the restriction map.

2.10.3 RNA secondary structure

The problem of finding the secondary structure of RNA can make fundamental use of the algorithm to solve two standard computer science problems, finding palindromic strings and matching parentheses. A abstract version of the RNA folding problem is given by R. Nussinov, G. Pieczenik, J. Griggs, & D. Kleitman [212] in 1978. A dynamic programming approach that minimizes the free energy in the folded structure is frequently used to find the secondary structure of RNA. M. Kanehisa & W. Goad [152], M. Zuker [296], and L. Larmore [173] gave dynamic programming algorithms to solve the problem using the dynamic programming approach. This dynamic programming algorithm is actually similar to the dynamic programming algorithms used to do sequence alignment. S. Le, R. Nussinov, & J. Maizel [177], in 1989, describe a method to represent an RNA sequence as a binary tree where the nodes represent a loop and the edges represent stems. Representing the secondary structure of RNA in this way is used to facilitate the comparison and alignment of RNA sequences.

2.10.4 Phylogenetic reconstruction

One of the major reasons people align genetic sequences is to infer taxonomic relationships between the various taxa. There is a vast literature associated with the construction of phylogenetic trees. An excellent, although somewhat out of date, review of the methods used to construct phylogenetic trees has been published by J. Felsenstein [86] in 1982.

2.10.5 Multi–dimensional matching

Some of the algorithms that have been developed to do approximate string matching can also be used to do two dimensional (and higher dimensional) pattern matching. In 1990 and 1991, A. Amir & G. Landau [18, 19] gave serial and parallel algorithms based on the k–mismatch problem to solve the multidimensional array matching problem. Another solution to the two dimensional k–mismatch pattern matching problem is given by S. Ranka & T. Heywood [224]. The divide and conquer approach to approximate string matching given by P. Cull & J. Holloway [63] can easily be extended to do multidimensional pattern matching.

Chapter 3

The Edit Distance Problem

The divide and conquer paradigm has been used to construct efficient algorithms for many problems. Typically, a problem of size n that is amenable to a divide and conquer solution can be broken into k sub–problems of size $\frac{n}{k}$, each of the sub–problems solved, and the solutions for the sub–problems used to construct a solution for the original problem.

Occasionally, it is not obvious how to decompose a problem of size n into sub–problems of size $\frac{n}{k}$, but it may be easy to decompose the problem into problems of size $n - k$. For problems of this type, it is often advantageous to compute and store the results for all of the sub–problems and then use these results to compute the answers for the original problem. This general method of computing and storing small results for later use is referred to as dynamic programming.

3.1 Problem definition

The edit distance problem is: Given two strings α and β from the alphabet Σ where $|\alpha| = m$, $|\beta| = n$, $n \leq m$, find the minimum cost of a sequence of edit operations that transforms β into a substring of α.

We must define the edit operations and their cost. Since I am interested in applying this algorithm to measuring the evolutionary distance between two genetic sequences, I will choose the operations used in a simple model of the evolution of genetic sequences [180]. Initially, I will only consider substitution, deletion, and insertion of a single character in the sequence.

- The function $substitute(\alpha, x, c),\ 1 \leq x \leq m,\ c \in \Sigma$, transforms the string $\alpha_1\alpha_2 \ldots \alpha_{x-1}\alpha_x\alpha_{x+1} \ldots \alpha_m$ into the string $\alpha_1\alpha_2 \ldots \alpha_{x-1}c\alpha_{x+1} \ldots \alpha_m$.
- The function $insert(\alpha, x, c),\ 1 \leq x \leq m,\ c \in \Sigma$, transforms the string $\alpha_1\alpha_2 \ldots \alpha_{x-1}\alpha_x \ldots \alpha_m$. into the string $\alpha_1\alpha_2 \ldots \alpha_{x-1}c\alpha_x \ldots \alpha_m$.
- The function $delete(\alpha, x),\ 1 \leq x \leq m$, transforms the string $\alpha_1\alpha_2 \ldots \alpha_{x-1}\alpha_x\alpha_{x+1} \ldots \alpha_m$ into the string $\alpha_1\alpha_2 \ldots \alpha_{x-1}\alpha_{x+1} \ldots \alpha_m$.

A positive cost is assigned to the insertion, deletetion, or substitution of a character.

- The function $C_1(c_1, c_2)$ returns the cost of substituting c_1 for c_2. The cost is typically 0 when $c_1 = c_2$.
- The function $C_2(c)$ returns the cost of inserting the character c.
- The function $C_3(c)$ returns the cost of deleting the character c.

The edit distance between α and β refers to the minimum cost of a sequence of edit operations that transforms β into a substring of α. Note that this sequence of edit operations is not necessarily unique.

3.2 The serial algorithm

In the early 1970's several people independently discovered a dynamic programming algorithm that could be used to find the edit distance between two strings. In sections 2.3.1 and 2.3.2 I reviewed the previous work on the edit distance and closely related problems. The dynamic programming algorithms to solve the edit distance problem were first proposed by Needleman & Wunsch [209], Ulam [274], Sankoff [231], Sellers [240], and Wagner & Fischer [279]. Since the algorithms were first introduced, many improvements and variants have appeared in the literature.

Erickson & Sellers [84] present an algorithm to solve the edit distance problem that is representative of the several dynamic programming algorithms found in the early 1970's. I briefly review the algorithm here. An $(n+1)\times(m+1)$ matrix M is constructed such that the value of $M(i,j)$ is the minimum cost of converting $\beta_1\beta_2\ldots\beta_j$ to $\alpha_k\alpha_{k+1}\ldots\alpha_i,\ 0\leq k\leq i$.

It is assumed in this algorithm that we are searching for the substring of α that is the smallest edit distance from β. To ensure that the alignment is not penalized for events that occur before α_k and after α_i, row 0 and column 0 of M are initialized as follows.

$$M(0,j) = \sum_{h=1}^{j} C_3(\beta_h),\ 1\leq i\leq n.$$

$$M(i,0) = 0,\ 0\leq i\leq m.$$

We can compute the value $M(i,j)$ by first computing the values of $M(i-1,j)$, $M(i-1,j-1)$, and $M(i,j-1)$ and then adding the cost of moving to $M(i,j)$.

- The cost of moving from $M(i-1,j)$ to $M(i,j)$ is $C_2(\alpha_i)$, the cost of inserting α_i into β. Using the insert operation from $M(i-1,j)$, the cost at $M(i,j)$ is $M(i-1,j)+C_2(\alpha_i)$.

- The cost of moving from $M(i-1,j-1)$ to $M(i,j)$ is $C_1(\alpha_i,\beta_j)$, the cost of changing α_i to β_j. Using the change operation from $M(i-1,j-1)$, the cost at $M(i,j)$ is $M(i-1,j-1)+C_1(\alpha_i,\beta_j)$.

- The cost of moving from $M(i,j-1)$ to $M(i,j)$ is $C_3(\beta_j)$, the cost of deleting β_j from β. Using the delete operation from $M(i,j-1)$, the cost at $M(i,j)$ is $M(i,j-1)+C_3(\beta_j)$.

$M(i,j)$ is assigned the minimum of the three values computed above. The remainder of the matrix M is computed inductively by letting each

$$M(i,j) = \min\begin{cases} M(i-1,j)+C_2(\alpha_i) \\ M(i-1,j-1)+C_1(\alpha_i,\beta_j) \\ M(i,j-1)+C_3(\beta_j) \end{cases} \tag{3.1}$$

for $1 \leq i \leq n,\ 1 \leq j \leq m.$

The scores in column n of the matrix $M,\ M(j,n),\ 1 \leq j \leq m$, are the costs to change β into some suffix of $\alpha_1 \cdots \alpha_j$. The cost of the minimum cost sequence of edit operations to convert β into a substring of α is the minimum value in the last column of M. Once the matrix M has been computed, find each q such that

$$M(q,n) = \min_{1 \leq h \leq m} M(h,n).$$

Each minimum value in column n of the matrix M represents one or more minimum cost conversions of β to α. By finding the values of $M(q,n)$, we know the minimum cost of converting β to α, but we do not know the sequence of operations used in the conversion. To find the sequence of edit operations used in the minimum cost conversion of β to α, we can, as we compute the matrix M, also compute a graph with a node representing each element of M and an arc from the node representing $M(a,b)$ to the node representing $M(i,j)$ if

$$M(a,b) = \min \begin{cases} M(i-1,j) + C_2(\alpha_i) \\ M(i-1,j-1) + C_1(\alpha_i,\beta_j) \\ M(i,j-1) + C_3(\beta_j) \end{cases}$$

Any path in the graph from a node representing $M(i,0),\ 0 \leq i \leq m$, to a node representing $M(q,n)$ represents a minimum cost sequence of edit operations to convert β into a substring of α.

The algorithm as it has been outlined here requires $O(mn)$ time since each element of the matrix M is computed, there are $O(mn)$ elements in M, and each element in M can be computed in constant time. The space used by this algorithm is also $O(mn)$ since we store the entire $m \times n$ matrix M. If we are only interested in the cost to convert β into a substring of α, and not the sequence of operations, the space used by the algorithm can be reduced to $O(m+n)$. Since each diagonal of M is dependent only on the previous two diagonals, there is no need to store more than the two most recently computed diagonals of M.

C. Wong & A. Chandra [294] and X. Huang [139] have shown an $\Omega(mn)$ lower bound to find the minimum cost sequence of edit operations to convert

β into α. D. Hirschberg [132], and later E. Myers & W. Miller [206] gave an algorithm to find the minimum cost sequence of edit operations to convert β into α, simultaneously using $O(m+n)$ space and $O(mn)$ time.

3.3 The parallel algorithm

Recently there have been several parallel algorithms to solve the edit distance problem. In 1988 and 1990, O. Ibarra gave parallel algorithms to solve the string edit distance problem for two strings of length n in $O(\log^2 n)$ time using $O(n^3/\log^2 n)$ processing elements on a hypercube. In 1992, O. Ibarra [144] gave an algorithm to solve the string edit distance problem using a linear array of n identical finite state machines in $O(m)$ time. In 1989, N. Core, E. Edmiston, J. Saltz, & R. Smith [56] gave a parallel implementation of the dynamic programming algorithm to align two sequences using $O(mn)$ space and $O(m)$ time on an $O(n)$ processor shared memory computer, $n \leq m$. In the same year X. Huang [140] gave a parallel algorithm that runs in linear time and used linear space. The edit distance problem has been used by A. Singh & R. Overbeek [248] and D. Sittig, D. Foulser, N. Carriero, G. McCorkle, & P. Miller [249] to demonstrate the usefulness of general purpose parallel programming languages. At the time the parallel algorithm in this chapter was developed (1988 and 1989), I was not aware of the papers mentioned above and many had not been published.

The model of computation we will be assuming is the EREW PRAM model as discussed in [77]. The PRAM model assumes an arbitrary number of identical processors that each have constant time access to a shared memory. The EREW version of the PRAM model assumes that a memory cell can be read by at most one processor during one instruction cycle. Likewise a memory cell can be written to by at most one processor during an instruction cycle. The EREW is the weakest of the PRAM models so the algorithms developed here will run on stronger models with, at most, the same upper bounds on time.

Notice that $M(i,j)$ is dependent only on $M(i-1,j)$, $M(i-1,j-1)$, and

```
compute_matrix ($\alpha$, $\beta$)

    n = $|\alpha|$
    m = $|\beta|$
    for (i = 1; i < n + m; i++)
        if (i < n)
            r = min (m, i)
            matrix [0, r] = 0
            for each $1 \leq j \leq r$ do in parallel
                matrix [j, i - j] = compute_element (j, i - j)
            if (i < m)
                matrix [i, 0] = matrix [n - 1, 0] + cost_of_delete ($\beta$[i])
        else
            for each (i - n) $\leq$ j $\leq$ m do in parallel
                matrix [j, n - j] = compute_element (j, n - j)

compute_element (i, j)

    min (
        matrix [i-1, j] + cost_of_insert ($\alpha_i$),
        matrix [i-1, j-1] + cost_of_substitute ($\alpha_i$, $\beta_j$),
        matrix [i, j-1] + cost_of_delete ($\beta_j$)
    )
```

Figure 3.1: A parallel dynamic programming algorithm to compute the minimum edit distance between two strings.

$M(i, j-1)$. If the diagonals

$$M(1,p-1),\ M(2,p-2),\ \ldots, M(p-2,2),\ M(p-1,1)$$

$$M(1,p-2),\ M(2,p-3),\ \ldots, M(p-3,2),\ M(p-2,1)$$

have been computed, then every element in the diagonal

$$M(1,p),\ M(2,p-1),\ \ldots, M(p-1,2),\ M(p,1)$$

can be computed in parallel. This observation leads almost immediately to a parallel algorithm to compute the matrix M.

Figure 3.1 is a parallel version of the dynamic programming algorithm given by Erickson & Sellers [84] and many others. The outside for loop in the function compute_matrix is executed once for each diagonal in the matrix. The end points of the diagonal are computed and the entire diagonal is computed in parallel in the inside for loop. The function compute_element implements the equation 3.1.

3.3.1 Parallel implementation

The parallel dynamic programming algorithm to compute the minimum edit distance between two strings has been implemented on a sequence balance 21000 with 28 processors and 24 Megabytes of memory. The number of processors that the algorithm actually uses can be varied from one to sixteen. Since the length of a diagonal is typically greater than the number of processors that are available, each processor is assigned a region of the diagonal to compute. When all processors have completed computing their region of the diagonal, the computations of the next diagonal are started. In this parallel implementation of the algorithm, I compute the minimum cost of converting one string into a substring of the other, not the edit script required to change one string into a substring of the other.

3.3.2 Performance

Theorem 3.1 *The parallel dynamic programming algorithm runs in $O(mn/p)$ time using $O(p)$, $p \leq \min(m, n)$ EREW PRAM processors.*

Proof. The outer for loop of the function compute_matrix executes once for each of the $m + n - 1$ diagonals. If the diagonal is in the upper left corner of the matrix then the argument of the first if statement will be true. Each statement in the clause following the first if statement, except the parallel for loop, can be computed using constant work. The parallel for loop will compute a diagonal with at most $\min(m, n)$ elements. Each element can be computed independently and if the reading of the three elements matrix [i-1, j], matrix [i-1, j-1], and matrix [i, j-1] is synchronized, no cell will need to be read from simultaneously. Using p, $p \leq \min(m, n)$ EREW PRAM processors, the diagonal will be computed in $O(\min(m, n)/p)$ time. Similarly for the else clause of the first if statement. The algorithm will use $O((m + n - 1)\min(m, n)/p)$ time. ∎

The algorithm was tested by computing the cost of the minimum cost edit script required to convert one string into a substring of another string for several pairs of strings. In the paper by Erickson & Sellers [84], two nucleotide sequences from satellite DNA of *Drosophila melanogaster* (fruit fly) are aligned using the dynamic programming algorithm. The first nucleotide sequence contained 359 base pairs and the second contained 254 base pairs. The parallel dynamic programming algorithm was used to align these sequences varying the number of processors from one to seven. Table 3.1 shows the running times used by the program to compute the cost of the minimum cost edit sequence to convert one sequence into a subsequence of the other while varying the number of processors from one to seven. Since the overhead of starting a processor on its region of the diagonal is large compared to the time it takes to do its share of the work, the speedup is not ideal. The time used by one processor is about 4.7 times the the time used by seven processors.

I have used these satellite DNA sequences to create, by concatenation, longer sequences to give to the parallel dynamic programming algorithm. Tables 3.2 and 3.3 give the results of using this algorithm to compare various length strings with either one, two, four, eight, or sixteen processors. The length of the strings being aligned were approximately equal and the size is reported as the product of the string lengths divided by 1000. The elapsed time, in seconds, used by the parallel dynamic programming algorithm to find the minimum cost required to convert one string into a substring of the other is reported in Tables 3.2 and 3.3. The speedup for n processors is reported as the elapsed time used by one processor to solve the problem, divided by the elapsed time used by n processors to solve the problem. Tables 3.2 and 3.3 show that, although the elapsed time generally decreases as processors are added, the efficiency of using more than four processors is low.

The size of the problem solved by the dynamic programming algorithm is often limited by the $O(mn)$ memory requirement of the algorithm. Memory

number of processors	time (seconds)	speedup	percent utilization
1	26.14	1.00	100.0
2	13.96	1.87	93.5
3	9.96	2.62	87.3
4	7.92	3.30	82.5
5	6.71	3.90	78.0
6	5.93	4.41	73.5
7	5.52	4.74	67.7

Table 3.1: Time to compute the edit distance between two satellite DNA sequences from *D. melanogaster* using the parallel dynamic programming algorithm while varying the number of processors.

Processors	$\frac{\|string1\| \times \|string2\|}{1000}$	Speedup	Time (seconds)
1	20	1.0	5.23
2	20	1.75	2.97
4	20	2.72	1.92
8	20	3.19	1.64
16	20	2.84	1.84
1	40	1.0	10.19
2	40	1.82	5.61
4	40	2.89	3.52
8	40	3.82	2.67
16	40	3.79	2.69
1	57	1.0	14.40
2	57	1.83	7.88
4	57	2.99	4.82
8	57	4.01	3.59
16	57	4.11	3.50
1	92	1.0	23.20
2	92	1.86	12.46
4	92	3.27	7.08
8	92	4.99	4.65
16	92	5.89	3.94

Table 3.2: Speedup of parallel dynamic programming algorithm, part 1

Processors	$\frac{\lvert string1\rvert \times \lvert string2\rvert}{1000}$	Speedup	Time (seconds)
1	130	1.010	32.11
2	130	1.87	17.20
4	130	3.31	9.70
8	130	5.18	6.20
16	130	6.35	5.06
1	183	1.0	45.95
2	183	1.88	24.48
4	183	3.39	13.57
8	183	5.49	8.37
16	183	6.90	6.66
1	259	1.0	63.48
2	259	1.88	33.78
4	259	3.41	18.65
8	259	5.61	11.32
16	259	7.36	8.63
1	366	1.0	91.47
2	366	1.89	48.48
4	366	3.46	26.45
8	366	5.81	15.74
16	366	7.80	11.72

Table 3.3: Speedup of parallel dynamic programming algorithm, part 2

requirements limit the size of the problem, and therefore the number of processors that can be effectively used to solve the problem. If the alignment does not need to be constructed, the similarity score can be reported using only $O(m+n)$ space. By storing only the two previously computed diagonals, instead of the complete matrix, the similarity score of much large problems can be computed.

In this chapter I have reviewed the widely used dynamic programming algorithm to align sequences. I have given a parallel algorithm to computer the matrix used by the dynamic programming algorithms that runs in $O(mn/p)$ time using $O(\min(m,n))$ EREW PRAM processors. The limitations of this algorithm are discussed in chapter 7 where I give a divide and conquer algorithm to solve similar problems.

Chapter 4

The Most Frequent Substring Problem

A biochemist requested a program that would print the number of times that each possible substring of length m appeared in a string of length n. The biochemist was interested in finding small DNA or RNA substrings (six to fifteen nucleotides) that were most frequently repeated in sequences that consisted of a few thousand nucleotides. It is clear that simply enumerating all of the substrings of length m and counting the number of times each substring appears in the sequence is not feasible since there will be, in the case of RNA, 4^m such substrings. Since there are at most $n - m + 1$ distinct substrings of length m in a string of length n it is not necessary to enumerate all 4^m possible strings of length m.

In this chapter I will give a simple algorithm to solve the most frequent substring problem. I have implemented the algorithm and it has been used by the biochemist that originally requested the program (and others) to investigate the subsequence composition of certain RNA sequences. The main result in this chapter is the optimal parallel algorithm to solve this problem using $O(p)$, $p \leq n/\log n$ or fewer EREW[1] PRAM[2] processors, that runs in $O(mn/p)$ time. I have implemented the sorting phase of the algorithm and present results of running the program with various numbers of processors on various sized problems.

[1] exclusive read exclusive write

[2] parallel random access machine

4.1 Problem definition

The most frequent substring problem is: Given a string T of length n over the fixed, finite alphabet Σ, $\sigma = |\Sigma|$, and an integer m, find the length m substring P of T that occurs most frequently in T and the location of each occurrence of P in T.

4.2 Sequential algorithm

Theorem 4.1 *The most frequent substring problem can be solved in $O(mn)$ time.*

Proof. Create a list of the $n - m + 1$ length m substrings of T and associate with each substring its position in T. This can be done in $O(n)$ time by creating an array of $n - m + 1$ pointers so that the j^{th} element of the array points to the j^{th} character in T. Store with each pointer, the value j to indicate the position of the substring in T.

Sort the list of substrings using radix sort. This can be done in $O(mn)$ time using σ buckets by the well known algorithm presented in [6].

Scan the list of sorted substrings noting the longest run of identical substrings. By comparing consecutive substrings in the sorted list and keeping track of the longest run of equal substrings we can report the number and positions of the most frequently occurring substring in T. Since there will be at most $n - m$ substring comparisons and each requires at most m character comparisons, this final step can be completed in $O(mn)$ time. ∎

4.3 Parallel algorithm

The model of computation we will again be assuming is the EREW PRAM model as discussed in [77]. The parallel algorithm uses the same three distinct steps that the sequential algorithm used; create the list of strings, sort the list of strings, and scan the list of sorted strings.

We will show that the most frequent substring problem can be solved in $O(mn/p)$ time using $O(p)$, $p \leq n/\log n$ EREW PRAM processors. We use lemmas 4.1, 4.2, and 4.3 to show that the sorting phase of the algorithm can be completed in $O(mn/p)$ time using $O(p)$ EREW PRAM processors. Lemma 4.4 will show that given a sorted list of strings, each of length m, we can find the longest run of identical strings in $O(mn/p)$ time using $O(p)$ processors.

Lemma 4.1 *The array of length m strings, $A_s[1], \ldots, A_s[n]$ can be ordered by the d^{th} character of the strings, $0 \leq d < m$, in $O(n/p)$ time using $O(p)$ EREW PRAM processors, $p \leq n/\log n$.*

Proof. Each element of the length $n \cdot \sigma$ array A_l is initialized to $\alpha \notin \Sigma$. Each processor, P_j, $0 \leq j < p$, sequentially writes α to $A_l[j \cdot n/p \cdot \sigma + q]$, $0 \leq q < n/p \cdot \sigma$. Each P_j sequentially moves the n/p values from $A_s[j \cdot n/p + q]$ to $A_l[\mathrm{ord}(A_s[j][d]) \cdot n + j \cdot n/p + q]$ where $\mathrm{ord}(x)$ returns the ordinal value of x. Note that $A_l[u \cdot n + j \cdot n/p + q]$, $0 \leq u < \sigma$, is written to exclusively by P_j. ▮

In the following lemmas, $C_r[x]$, $D_r[x]$, and $E_r[x]$ are used to represent the values at the nodes of a full binary tree. $C_{\log n}[0]$ is the root of the tree C with n leaves, $C_{r-1}[2x]$ and $C_{r-1}[2x+1]$ are the children of $C_r[x]$, and $C_0[0], C_0[1], \ldots, C_0[n-1]$ are the leaves of the tree. The nodes at level r, $0 \leq r \leq \log n$, are $C_r[x]$, $0 \leq x < \frac{n}{2^r}$.

Lemma 4.2 *Given a length $n \cdot \sigma$ array, A_l that contains n legal values (a legal value is any string from Σ^m), the legal values can be put in $A_s[1], A_s[2], \ldots, A_s[n]$, preserving the original order of the legal values of A_l in $O(n/p)$ time using $O(p)$ EREW PRAM processors, $p \leq n/\log n$.*

Proof. Let $s = \log \sigma$. First we compute $D_r[x]$, $0 \leq r \leq \log(n \cdot \sigma)$, $0 \leq x < \frac{n \cdot \sigma}{2^r}$, the number of illegal values in

$$A_l[2^r \cdot x], A_l[2^r \cdot x + 1], \ldots, A_l[2^r \cdot (x+1) - 1].$$

Each processor, P_j, $0 \leq j < p$, writes the value 0 to $D_0[u \cdot n + j \cdot n/p + q]$, $0 \leq u < \sigma$, $0 \leq q < n/p$, if $A_l[u \cdot n + j \cdot n/p + q]$ contains a legal value, otherwise a 1 is written. D_0 can be computed with each P_j performing n/p independent operations. Again, notice that each element of D_0 can be written to by exactly one P_j and each element of A_l is read by exactly one P_j. Let $D_r[x] = D_{r-1}[2x] + D_{r-1}[2x+1]$, $1 \leq r \leq \log(n \cdot \sigma)$, $0 \leq x < \frac{n \cdot \sigma}{2^r}$. We will compute D_r, $0 \leq r \leq (n/p + s)$, in two steps, D_r for $0 \leq r \leq \log(n/p+s)$ and D_r for $\log(n/p+s) < r \leq (n/p+s)$. In the first step there are more elements in D_r than processors so we can keep all of the processors busy. The time to compute D_r, $0 \leq r \leq \log(n/p+s)$ is

$$\sum_{r=0}^{\log(n/p+s)} \frac{(n/p+s)}{2^r} < 2(n/p+s).$$

While computing D_r, $\log(n/p+s) < r \leq (n/p+s)$ some of the processors will be idle since there are more processors than elements in D_r and the computation of D_r is dependent on D_{r-1}. Each D_r can be computed in constant time. The time used to perform the second step is

$$\sum_{r=\log(n/p+s)}^{(n/p+s)} c \ = c((n/p+s) - \log(n/p+s) + 1).$$

Therefore, the total time to compute D_r, $0 \leq r \leq (n/p+s)$, is $O(n/p)$.

We will now compute $E_r[x]$, $0 \leq r \leq \log(n \cdot \sigma), 0 \leq x < \frac{n \cdot \sigma}{2^r}$, a pair of values $E_r[x]_{left}$ and $E_r[x]_{right}$. $E_r[x]_{left}$ is the number of illegal values in

$$A_l[0], A_l[1], \ldots, A_l\left[x \frac{n \cdot \sigma}{2^{\log(n \cdot \sigma) - r}} - 1\right].$$

$E_r[x]_{right}$ is the number of illegal values in

$$A_l[0], A_l[1], \ldots, A_l\left[(x+1) \frac{n \cdot \sigma}{2^{\log(n \cdot \sigma) - r}} - 1\right].$$

The value $E_0[x]_{left}$ is the number of illegal values in

$$A_l[0], A_l[1], \ldots, A_l[x-1].$$

To compute $E_0[x]$ we let $E_{\log(n\cdot\sigma)}[0]_{left} = 0$ and $E_{\log(n\cdot\sigma)}[0]_{right} = D_{\log(n\cdot\sigma)}[0]$. Now, for $0 \le r \le \log(n \cdot \sigma)$ and $0 \le x < \frac{n\cdot\sigma}{2^r}$, we let

$$\begin{aligned}
E_{r-1}[2x]_{left} &= E_r[x]_{left} \\
E_{r-1}[2x]_{right} &= E_r[x]_{left} + D_{r-1}[2x] \\
E_{r-1}[2x+1]_{left} &= E_r[x]_{right} - D_{r-1}[2x+1] \\
E_{r-1}[2x+1]_{right} &= E_r[x]_{right}
\end{aligned}$$

The same arguments used to show that D_r, $0 \le r \le \log(n \cdot \sigma)$, can be computed in $O(n/p)$ time can be used to show that E_r, $0 \le r \le \log(n\cdot\sigma)$, can be computed in $O(n/p)$ time using $p \le n/\log n$ EREW PRAM processors.

Let $\alpha = u \cdot n + j \cdot n/p + q$, $0 \le u < \sigma$, $0 \le q < n/p$, $0 \le j < p$. Each processor, P_j, sequentially moves the values in $A_l[\alpha]$ to $A_s[\alpha - E_0[\alpha]_{left}]$ if $A_l[\alpha]$ is a legal value. Since exactly one of $A_l[u \cdot n + x]$, $0 \le u < \sigma$, $0 \le x < n$, contains a legal value for a given value of x, only one value will be written to each element of A_s. ∎

Lemma 4.3 *An array of n strings, each of length m, over a finite, fixed alphabet can be sorted in $O(mn/p)$ time using $O(p)$ EREW PRAM processors, $p \le n/\log n$.*

Proof. The standard radix sort as presented in [6] uses m iterations of ordering the strings by position d, $0 \le d < m$, of the strings. Since, by Lemmas 4.1 and 4.2, we can order the n strings by the character in position d in $O(n/p)$ time, we can sort the n strings in $O(mn/p)$ time. ∎

Lemma 4.4 *Given a sorted array of strings, $A[1], A[2], \ldots, A[n]$, each of length m, the position e such that $A[e] = A[e+1] = \cdots = A[e+d]$ that maximizes d can be found in $O(mn/p)$ time using $O(p)$ EREW PRAM processors, $p \le n/\log n$.*

Proof. We will compute the values of $C_r[x]$, $0 \le r \le \log(n \cdot \sigma)$, $0 \le x < \frac{n\cdot\sigma}{2^r}$, the length of the longest run of equal strings in a region of A. Each $C_r[x]$

has four values associated with it: e, d, L, and R. The value e points to the start of the longest consecutive run of equal strings, that is e such that $A[e] = A[e+1] = \cdots = A[e+d]$ that maximizes d in the range $A[2^r \cdot x], A[2^r \cdot x + 1], \ldots, A[2^r(x+1) - 1]$. L is the smallest value such that $2^r \cdot x \leq L < 2^r(x+1)$ and $A[L] \neq A[L+1]$, or if no such L exists, L is assigned $2^r(x+1) - 1$. R is the largest value such that

$$2^r \cdot x \leq R < 2^r(x+1) \text{ and } A[R-1] \neq A[R],$$

or if no such R exists, R is assigned $2^r \cdot x$. Each processor, P_j, $0 \leq j < p$, sequentially computes $C_0[j \cdot n/p + q]$, $0 \leq q < n/p$. $C_r[x]$, $1 \leq r \leq n/p$, $0 \leq x < \frac{n \cdot \sigma}{2^r}$, is computed by

$$C_r[x] = \begin{cases} X & \textbf{IF} \quad C_{r-1}[2x]_d > C_{r-1}[2x+1]_d \textbf{ AND} \\ & \quad (C_{r-1}[2x]_d > C_{r-1}[2x+1]_L - C_{r-1}[2x]_R \textbf{ OR} \\ & \quad A[C_{r-1}[2x+1]_L] \neq A[C_{r-1}[2x]_R]) \\ & \\ Y & \textbf{IF} \quad C_{r-1}[2x+1]_d > C_{r-1}[2x]_d \textbf{ AND} \\ & \quad (C_{r-1}[2x+1]_d > C_{r-1}[2x+1]_L - C_{r-1}[2x]_R \textbf{ OR} \\ & \quad A[C_{r-1}[2x+1]_L] \neq A[C_{r-1}[2x]_R]) \\ & \\ Z & \textbf{IF} \quad C_{r-1}[2x+1]_L - C_{r-1}[2x]_R > C_{r-1}[2x]_d \textbf{ AND} \\ & \quad C_{r-1}[2x+1]_L - C_{r-1}[2x]_R > C_{r-1}[2x+1]_d \textbf{ AND} \\ & \quad A[C_{r-1}[2x+1]_L] = A[C_{r-1}[2x]_R] \end{cases}$$

where

$$\begin{aligned} X_e &= C_{r-1}[2x]_e & \qquad Y_e &= C_{r-1}[2x+1]_e \\ X_d &= C_{r-1}[2x]_d & \qquad Y_d &= C_{r-1}[2x+1]_d \\ X_L &= C_{r-1}[2x]_L & \qquad Y_L &= C_{r-1}[2x]_L \\ X_R &= C_{r-1}[2x+1]_R & \qquad Y_R &= C_{r-1}[2x+1]_R \end{aligned}$$

$$Z_e = C_{r-1}[2x]_R$$

$$
\begin{aligned}
Z_d &= C_{r-1}[2x+1]_L - C_{r-1}[2x]_R \\
Z_L &= \begin{cases} C_{r-1}[2x]_L & \textbf{IF } C_{r-1}[2x]_R > C_{r-1}[2x]_L \\ C_{r-1}[2x+1]_L & \text{otherwise} \end{cases} \\
Z_R &= \begin{cases} C_{r-1}[2x+1]_R & \textbf{IF } C_{r-1}[2x+1]_L < C_{r-1}[2x+1]_R \\ C_{r-1}[2x]_R & \text{otherwise} \end{cases}
\end{aligned}
$$

The arguments used in lemma 4.2 can be used to show that $O(p)$ EREW PRAM processors can compute $C_{\log n}[0]$ in $O(mn/p)$ time, $p \leq n/\log n$. ▮

Theorem 4.2 *The most frequent substring problem can be solved in $O(mn/p)$ time using $O(p)$ EREW PRAM processors, $p \leq n/\log n$.*

Proof. Using $O(p)$ processors we can create the $n - m + 1$ length m substrings in $O(n/p)$ time. By using lemma 4.3 we can sort the substrings in $O(mn/p)$ time. The final step is to find the longest run of identical substrings in the sorted array A_s. By lemma 4.4 this can be done in $O(mn/p)$ time using $O(p)$ EREW PRAM processors, $p \leq n/\log n$. ▮

It appears that a weaker model can be used by assuming that during each instruction cycle each processor is executing the same instruction. If this is true, it may be possible to adapt this algorithm to run on a vector processor or a SIMD[3] machine.

4.4 Parallel implementation

The sorting phase of the algorithm was implemented in C on a Sequent B21000 with 28 processors and 24 Megabytes of memory.. Table 4.1 shows the time in seconds used to sort various length substrings (2, 4, 8, and 16 characters) of a string of length 2^{15} The number of processors used were 2, 4, 8, and 16. Ideally, the time to solve a problem of size n with p processors would be the time to solve a problem of size n with one processor divided by p. Table 4.2 gives, $\mathcal{E}_{p,s}$,

[3] Single Instruction Multiple Data

Number of Processors	Substring Size			
	2	4	8	16
1	45.5	89.0	176.6	351.8
2	24.4	47.8	92.8	183.3
4	13.4	25.4	49.7	99.6
8	7.9	13.8	27.1	54.6
16	5.1	8.9	17.9	32.4

Table 4.1: Elapsed time in seconds to sort substrings of a string of 2^{15} characters

Number of Processors	Substring Size			
	2	4	8	16
1	100	100	100	100
2	93.2	93.1	95.2	96.0
4	84.9	87.6	88.8	88.3
8	72.0	80.6	81.5	80.5
16	55.8	62.5	61.7	67.9

Table 4.2: Percent processor utilization for a string of 2^{15} characters

the percent of the available processing power that is effectively used to solve the problem with this algorithm. $\mathcal{E}_{p,s} = \frac{t_{1,s}}{p \cdot t_{p,s}} \cdot 100$ where p is the number of processors and $t_{p,s}$ is the time used by p processors to solve the problem with substrings of length s.

4.5 Space usage

This algorithm uses $c_1 \sigma n + c_2$ words of memory (c_1 and c_2 are constants). For large alphabets, this algorithm will become impractical on a shared memory parallel computer. The large memory needs would require that some of the information be stored on a secondary device such as a disk. Since the secondary storage is typically serial (i.e. one disk controller) and slow with respect to the processors, the advantage of parallelizing the algorithm would disappear.

4.6 Related parallel bucket sort algorithms

Hirschberg [133] developed a parallel bucket sort algorithm to sort n numbers with n processors in $O(\log n)$ time and $O(mn)$ space. The numbers were required to be in the range $\{1 \ldots m-1\}$ and to be unique. The algorithm was generalized to sort lists of numbers with duplicates. Using $O(n^{1+1/k})$ processors, the generalized algorithm could sort n numbers in $O(k \log n)$ time.

Nassimi & Sahni [208] developed a parallel radix sort algorithm that sorts n elements using $O(n^{1+1/k})$ processors in $O(k \log n)$ time. This algorithm is developed specifically for processors arranged in cube–connected cycles or in a perfect shuffle structure.

Reif [225] presents an algorithm to sort n elements in the range $\{1 \ldots \log n\}$ in $O(\log n)$ time using $O(n/\log n)$ processors. This algorithm is then generalized to sort n numbers in the range $\{1 \ldots (\log n)^c\}$ in $O(c \log n)$ time using $O(n/\log n)$ processors.

My algorithm uses the processors more efficiently than the algorithm presented by Nassimi & Sahni [208] although they are designed for very different architectures. The algorithm presented by Reif [225] has the same time bounds as my algorithm, although I used the weaker EREW PRAM model of computation.

The most frequent substring problem and a simple sequential algorithm to solve it were introduces in this chapter. A parallel algorithm that can find the positions of the most frequent length m substring of a length n string in $O(mn/p)$ time using $O(p)$, $p \le n/\log n$ EREW PRAM processors. This algorithm required the development of an efficient parallel radix sort on a slightly weaker model of parallel computation than has been done previously.

It may be possible to extend these results to the “most frequent **motif** problem” searching for length m motifs, as discussed in chapter 8, instead of length m substrings.

Chapter 5

Shotgun Sequencing

Biological and physical limitations require that DNA be sequenced in fragments. Because of these limitations there are two approaches used to obtain the appropriate sized fragments of DNA to sequence.

One class of methods for sequencing DNA is loosely termed ordered sequencing. These methods use an oligonucleotide primer to initiate the DNA sequencing at a known point and the sequencing reaction proceeds from this point. The leading several hundred bases of the DNA strand are sequenced and then removed using exonucleases exposing the next segment of the DNA to be sequenced. The process of sequencing several hundred bases and removing them is continued until the entire DNA sequence has been determined. We will not consider ordered sequencing in this paper.

Another class of methods for sequencing DNA is loosely referred to as shotgun sequencing. Many identical copies of the DNA to be sequenced are cleaved by one or more restriction endonucleases. A restriction endonuclease will cleave the DNA sequence at each occurrence of a specific six to eight base subsequence (sonication can also be used to create unordered fragments). This results in a multiset[1] of DNA fragments that are not ordered. DNA fragments are essentially selected at random from this multiset and sequenced. A consensus sequence that is believed to represent the original DNA sequence is assembled by finding overlaps between the DNA fragments that have been sequenced.

[1] A multiset is analogous to a set, but elements may appear more than once [160].

In this chapter we will present three algorithms that could be used to assemble the sequenced DNA fragments into a contiguous sequence. In section 5.1 we review the previous work done on the minimal length superstring problem. Since the minimal length superstring problem is NP–complete, this section also reviews several approximation algorithms. We define the perfect string consensus problem in section 5.2. Examining this problem will motivate some of the assumptions that we will make later in this chapter. Finally, in section 5.3, we define the string consensus problem and present three algorithms to solve it.

We will use calligraphic letters such as $\mathcal{S}$ to represent multisets of strings. Capital letters late in the alphabet will be used to indicate strings, while lower case letters will be used for characters of a string such as $S = s_1 s_2 \ldots s_m$. The strings are composed of characters from the alphabet Σ. We will use the following notation,

- n is the number of strings in the multiset $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$
- $N = \sum_{i=1}^{n} |S_i|$
- L_{min} = length of shortest common superstring
- L = length of superstring found by heuristic
- C = compression = $N - L_{min}$
- $C_H = N - L$ = compression found by heuristic.
- suffix $(T,\ j)$ is the suffix of T that has length j, similarly for prefix $(T,\ j)$.

5.1 Previous work

In this section we will examine the work that has been done on reconstructing a the original sequence from the individual sequence fragments. We use the term consensus string as the name of the string that is reconstructed from the sequence fragments. Several people have studied this problem from the biologist's perspective [53, 257, 149] and many people have studied similar problems in computer science [186, 107, 217, 262, 269, 273, 181, 43].

In 1980 Gallant, Maier & Storer [107] showed that the shortest superstring problem is NP–complete. They first defined superstring,

> a *superstring* of a set of strings $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ is a string S containing each S_i, $1 \leq i \leq n$, as a substring,

and then defined the shortest superstring problem,

> Given a set of strings $\mathcal{S}$ and a positive integer l, does $\mathcal{S}$ have a superstring of length l?

They showed that the shortest superstring problem is NP–complete even when the alphabet is restricted to {0,1}. The NP–completeness result suggests that there is no polynomial time algorithm for this problem. Therefore, one should probably attack this problem in one of several ways:

1. Add assumptions about the substrings so that the approximate problem, which is the original problem with the extra assumptions, is no longer NP–complete.

2. Show that the hard instances of the problem are rare, so that the problem can usually be solved quickly.

3. Instead of finding the superstring with the minimum length, find a superstring with a length that can be shown to be close to the minimum length.

Although we mention three approaches to deal with an NP–complete problem, approximation has been used most frequently to solve this problem quickly. We will assume that, given a minimum acceptable overlap, there is a unique construction of the superstring. This is an example of approach one, adding assumptions so that the problem is no longer NP–complete.

In a paper by Peltola, Söderlund, Tarhio & Ukkonen [217] a heuristic for a generalized minimal length superstring problem is given. The problem is generalized by allowing errors in the string matching. They use a different definition of superstring, namely,

> a superstring of a set of strings, $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, is a string S containing each S_i, $1 \leq i \leq n$, as an **approximate** substring.

An approximate substring S_i of S with an error ratio δ is defined to be,

> a substring of S that can be transformed into S_i with at most $\delta|S_i|$ delete, insert, replace, and transpose operations is an approximate substring..

The minimum number of delete, insert, and replace operations needed to transform one string into another is frequently called the minimum edit distance. They then define the generalized minimal length superstring problem,

> Given a set of strings $\mathcal{S}$, a positive integer l, and an error ratio δ, $0 < \delta < 1$, does $\mathcal{S}$ have a superstring of length at most l?

A heuristic is given to solve the problem, although no performance guarantees are given for the heuristic.

The heuristic has three basic steps. First a complete pairwise alignment graph is computed using the standard dynamic algorithm of Sellers [240] and others. Each node of the graph represents some $S_i \in \mathcal{S}$. An edge exists between S_i and S_j iff the minimum edit distance is less than $\delta|S_j|$. The value associated with the edge is the minimum edit distance between S_i and S_j. From this graph, a global alignment is computed by essentially computing the minimal spanning tree of the pairwise alignment graph. The minimal spanning tree provides an ordering of the strings and the optimal alignment of adjacent strings, but does not provide the optimal global alignment, or the optimal local alignment when more than two strings overlap. The final step in the heuristic computes the consensus string from the overlap graph and the minimal spanning tree. The running time of the heuristic is $O(\delta N^2)$.

Papers by Tarhio & Ukkonen [262] and Turner [269] develop approximation algorithms that they conjecture will find an approximate shortest superstring that is, at worst, twice the length of the actual shortest superstring. Tarhio & Ukkonen [262] analyzed their algorithm in terms of the compression of the strings in $\mathcal{S}$ instead of the length of the shortest superstring of $\mathcal{S}$. The main result of the paper is that $C_H \geq \frac{1}{2}C$. The running time of the algorithm presented by Tarhio & Ukkonen is $O(nN)$.

The algorithm computes the maximal overlap of all pairs of strings in $\mathcal{S}$. It then behaves in a greedy fashion to construct a short common superstring by selecting the longest overlap between two strings. When a string has been overlapped on both ends it is removed from $\mathcal{S}$. The process of selecting the longest overlap continues until no strings remain in $\mathcal{S}$.

Turner [269] presents an algorithm to approximate the shortest superstring with the same performance guarantees as the algorithm of Tarhio & Ukkonen [262]. Turner uses suffix arrays to reduce the number of suffix/prefix comparisons that need to be done. The running time of the algorithm is $O(N \log N)$ or $O(N \log n)$, depending on whether direct indexing over the alphabet Σ is allowed.

Later, in a paper by Ukkonen [273], an $O(N)$ or $O(N \min(\log n, \log |\Sigma|))$ algorithm, depending on whether direct indexing over the alphabet Σ is allowed, to solve the approximate shortest superstring problem is presented. This reduction in time is achieved by a clever use of the Aho–Corasick [4] string matching automaton. This algorithm achieves the same compression ratio as the algorithms of Tarhio & Ukkonen and Turner.

The first approximation algorithm that approximated the shortest superstring of $\mathcal{S}$ instead of the maximal compression of the strings in $\mathcal{S}$ was given by Li [181]. Li was able to give an algorithm to compute an approximate shortest superstring of length $O(L_{min} \log L_{min})$. The algorithm is similar to the greedy algorithms given above by Tarhio & Ukkonen, Turner, and Ukkonen. It differs in that when the strings with the maximum overlap are joined, not only are the strings that were joined removed from the set of strings, but all substrings of the resulting joined string are removed from the set of strings. This results in the size of the set of strings decreasing fast enough to show the $O(L_{min} \log L_{min})$ bound on the length of the approximate shortest superstring. Although the running time of the algorithm is not considered, it is clearly polynomial in the size of the input.

Blum et. al. [43] recently showed that the greedy algorithms of Tarhio & Ukkonen [262] and Turner [269] do find superstrings that are, at worst, a multiplicative factor of four longer than the minimum length superstring. Blum et. al. also give a modification of the algorithm to get the multiplicative factor down to three.

5.2 The perfect match string consensus problem

Previous work on finding the shortest common substring problem has ignored the process of generating the fragment strings. In this section we will look at a problem similar to the shortest common superstring problem with some added information about how the fragment strings were generated. In section 5.3 we will make much more severe assumptions about how the fragment strings were generated.

5.2.1 Problem definition

In the perfect match string consensus problem, I will assume that we start with k copies of the string, each string is fragmented, and we are given the multiset $\mathcal{S}$ that contains each fragment from the k copies of the string. We will use the symbol $\uplus$ for multiset union. Let there be k identical copies of the string $W \in \Sigma^n$, $W_1, W_2, \ldots, W_k$. Associated with each W_i, $1 \leq i \leq k$, is a multiset of substrings, $\mathcal{S}_i = \{T_1, T_2, \ldots, T_{l_i}\}$, such that $W_i = T_1 \cdot T_2 \cdots T_{l_i}$. Let $\mathcal{S} = \biguplus_{1 \leq i \leq k} \mathcal{S}_i$. The perfect string consensus problem is to find all W that could generate $\mathcal{S}$ given the multiset $\mathcal{S}$ and the integer k.

5.2.2 How many distinct common superstrings exist?

Given the multiset $\mathcal{S}$, it is important to know if there exists a unique W that can be constructed from the strings in $\mathcal{S}$. If W is not unique, we should know how many different W would be of superstrings $\mathcal{S}$. For the remainder of this section, we will assume that we started with two copies of W, that is $k = 2$.

There are two simple situations that will result in W not being unique. The first results when any two of the W's are cleaved at the same position. W is not unique if, for any h, i, and j, $j \neq |W|$

$$\begin{aligned} \boxed{w_i w_{i+1} \cdots w_j} &\in \mathcal{S} \text{ and} \\ \boxed{w_h w_{h+1} \cdots w_j} &\in \mathcal{S} \end{aligned}$$

since

$$W = \boxed{w_1 \cdots w_{i-1}}\boxed{w_i \cdots w_j}\boxed{w_{j+1} \cdots w_{|W|}} \quad = \quad \boxed{w_1 \cdots w_{h-1}}\boxed{w_h \cdots w_j}\boxed{w_{j+1} \cdots w_{|W|}}$$

and

$$W = \boxed{w_{j+1} \cdots w_{|W|}}\boxed{w_1 \cdots w_{h-1}}\boxed{w_h \cdots w_j} \quad = \quad \boxed{w_{j+1} \cdots w_{|W|}}\boxed{w_1 \cdots w_{i-1}}\boxed{w_i \cdots w_j}$$

The second situation that will result in W not being unique can be described as follows. If we have seven distinct strings,

$$\begin{aligned} R1, R2, R3, R4, R5, R6, R7 &\in \mathcal{S} \\ R1, R2, R3, R4, R5 &\in \mathcal{S}_1, \quad \text{and} \\ R6, R7 &\in \mathcal{S}_2 \end{aligned}$$

such that

$$\begin{aligned} \mathsf{suffix}(R1) \cdot R2 \cdot \mathsf{prefix}(R3) &= R6 \\ \mathsf{suffix}(R4) \cdot \mathsf{prefix}(R5) &= R7 \end{aligned}$$

$$\begin{aligned}\mathsf{suffix}(R1) &= \mathsf{suffix}(R4)\\ \mathsf{prefix}(R3) &= \mathsf{prefix}(R5)\end{aligned}$$

then there must be more than one arrangement of the R's to construct different W's.

$$W = \ldots R1 \cdot R2 \cdot R3 \ldots R4 \cdot R5 \ldots = \ldots .R6 \ldots\ldots\ldots .R7 \ldots$$

and

$$W = \ldots R1 \cdot R3 \ldots R4 \cdot R2 \cdot R5 \ldots = \ldots R7 \ldots R6 \ldots$$

The first case is actually a special case of the second. This can be seen by letting $\mathsf{prefix}\ (R5) = R3$ and noting that both $R3$ and $R6$ must end at the same position in W.

In a similar way we can create a multiset of n strings that can be arranged to form $n!$ distinct W's where, for $k = 2$, $|W| = \frac{n^2}{2} + \frac{7n}{2} + 6$. Let the multiset of strings, $\mathcal{X}_n$, be defined as

$$\begin{aligned}\mathcal{X}_0 &= \{\mathrm{xyyx},\ \mathrm{xy},\ \mathrm{yxxy},\ \mathrm{xy}\}\\ \mathcal{X}_n &= \mathcal{X}_{n-1} \uplus \{\mathrm{yx}^{n+1}\mathrm{y},\ \mathrm{xyyx},\ \mathrm{x}^{n-1}\}\end{aligned}$$

The length of W is

$$\begin{aligned}|W| &= \frac{\sum_{X\in\mathcal{X}_n} |X|}{k}\\ &= \frac{n^2}{2} + \frac{7n}{2} + 6\end{aligned}$$

There are $n!$ distinct arrangements of the strings in $\mathcal{X}_n$ that form W. The form of these arrangements will be:

$$W = \mathrm{xyyx}\cdot\mathrm{x}^{h_1}\cdot\mathrm{xyyx}\cdot\mathrm{x}^{h_2}\cdots\mathrm{xyyx}\cdot\mathrm{x}^{h_n}\cdot\mathrm{xy} = \mathrm{xy}\cdot\mathrm{yxx}^{h_1}\mathrm{xy}\cdot\mathrm{yxx}^{h_2}\mathrm{xy}\cdots\mathrm{yxx}^{h_n}\mathrm{xy}$$

where $h_1, h_2, \ldots, h_n$ are distinct integers, $0 \leq h_i < n$. The order of the n pairs x^{h_i} and $\mathrm{yxx}^{h_i}\mathrm{xy}$ in the construction of W is arbitrary, so there are $n!$ distinct strings that can be W. Notice that in no case are the two constructions of W cleaved at the same position (case 1). Similar arguments can be made for $k > 2$.

5.2.3 Algorithm to find the common superstrings

Since it is possible that so many different W exist, if we must produce all possible W then the best we can do is to search the entire space. Given two strings, U and V, $i = |U|$, $j = |V|$, $i \leq j$, such that $u_1 = v_1$, $u_2 = v_2, \cdots, u_i = v_i$, we will define the extension of U to V to be the string $v_{i+1} \cdots v_j$. We can build candidate strings, W_1 and W_2, from strings in $\mathcal{S}$ by choosing some string $S_1 \in \mathcal{S}$ and letting it be a prefix of W_1. We must then remove S_1 from $\mathcal{S}$ so that it will not be reused in the construction of W. W_2 is initially the null string.

$$\begin{aligned} W_1 &= \boxed{\quad S_1 \quad} \\ W_2 &= \end{aligned}$$

The extension of W_2 to W_1 is just S_1. Next we compute $\mathcal{S}'$,

$$\begin{aligned} \mathcal{S}' = \{S | (S \in \mathcal{S}) \wedge (\mathsf{prefix}(S) = \mathsf{extension}(W_1, W_2) \vee \\ S = \mathsf{prefix}(\mathsf{extension}(W_1, W_2)))\} \end{aligned}$$

We next pick some $S_2 \in \mathcal{S}'$ and make it a prefix of W_2. We now have

$$\begin{aligned} W_1 &= \boxed{\quad S_1 \quad} \\ W_2 &= \boxed{\qquad S_2 \qquad} \end{aligned}$$

The process of computing the extension of W_1 to W_2 and finding the multiset of strings that could be used to extend the shorter of W_1 and W_2 is repeated as long as it is possible to do so. When it becomes impossible to continue and $\mathcal{S}$ is not empty, we must backtrack and pick some new S_i from some $\mathcal{S}'$ and try again.

The algorithm in Figure 5.1 assumes that $k = 2$, although the same ideas will work for any k. Let E be the extension of the candidate strings for W_1 and W_2. Let $\mathcal{U}$ be the submultiset of $\mathcal{S}$ whose elements have not been used in the construction of W_1 or W_2. Let $\mathcal{P}$ be the submultiset of $\mathcal{U}$ whose elements have prefixes that exactly match E. The function $\mathsf{prefixq}$ is true if either argument is a

```
find–W (E, $\mathcal{U}$, $\mathcal{P}$, Results)
if ($\mathcal{U} = \Lambda$) $\wedge$ ($|$E$| = 0$)
    Print reverse (Results)
else
    if $\mathcal{P} = \Lambda$ backtrack
    else
        for each nextP $\in \mathcal{P}$
            Results $\leftarrow$ nextP $\uplus$ Results
            newE $\leftarrow$ extension (E, nextP)
            newP $\leftarrow$ $\{S | S \in \mathcal{S} \wedge$ prefixq (newE, $S$)$\}$
            find–W (newE, $\mathcal{U}$ - nextP, newP, Results)
```

Figure 5.1: An algorithm to compute all possible strings, W, that could have been used to create the multiset $\mathcal{S}$ from two copies of W.

prefix of the other. This algorithm has been implemented in lisp and has run on various computers.

5.3 The string consensus problem

The perfect string match consensus problem is not a good abstraction of the problem biochemists are faced with when they need to produce a consensus sequence from sequence fragments. In the perfect string consensus problem, every segment of each of the k copies of W must be in $\mathcal{S}$. In the process of shotgun sequencing, many copies of the original sequence will be fragmented, but relatively few of the fragments will actually be sequenced. We believe the string consensus problem, defined below, is a better abstraction of the problem biochemists are trying to solve.

Finding the minimum length superstring of a set of strings seems to require us to look at many of the possible arrangements of the strings in the set. When molecular biologists try to solve the similar problem of arranging their sequence fragments into a contiguous sequence, they assume that prefix/suffix matches of a length greater than some constant are "significant" and that all significant alignments are correct. In this section will use this assumption to construct an algorithm to build a contiguous sequence. This assumption will allow us to find alignments that are "good enough" and not require us to search the entire space of alignments. We shall see that in these algorithms, the run time is directly related to the compression.

If we let k be the minimum length match that we consider significant, one naive algorithm to solve this problem would compare the $\frac{n(n+1)}{2}$ pairs of strings to prefix/suffix matches of length k. If the $n-1$ prefix/suffix matches have not been found, k is incremented and the process is repeated. Eventually, if it is possible to align the sequences in a contiguous sequence, we will find it.

In this section we will present two new algorithms to solve the exact shotgun sequencing problem. The first uses the ideas of Rabin–Karp [158] string matching. The second algorithm sorts the strings and uses the sorted list to speed the prefix/suffix match searches.

5.3.1 Assumptions

We will make the following assumptions before we define the exact shotgun sequencing problem.

1. An integer k can be supplied that defines the minimum acceptable overlap between two strings.

2. There is a unique alignment of the strings in $\mathcal{S}$ such that all suffix/prefix overlaps are of length k or greater.

3. All suffix/prefix overlaps are exact matches.

We feel that if k is chosen carefully, these assumptions are reasonable. For large k, it will be very unlikely that any overlap other than overlaps that arise from the shotgun procedure will appear. If these assumptions are too strong, a simple modification to the algorithms described here (essentially incrementally increasing the value of k) will allow the algorithms to produce a series of alignments, each successive alignment being more compressed than the previous.

5.3.2 The problem definition

We are given a multiset of strings, $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, and an integer k. We make the following assumptions about $\mathcal{S}$ and k

1. S_i is not a substring of S_j for $1 \leq i,\ j \leq n,\ \ i \neq j$.

2. An ordering, H, of the strings in $\mathcal{S}$ exists such that

$$\forall_{1 \leq i < n}\, i \ \ \exists_{j \geq k}\, j \ \ \mathsf{suffix}(S_{H_i}, j) = \mathsf{prefix}(S_{H_{i+1}}, j).$$

The problem is, given the multiset $\mathcal{S}$ and the integer k, find the ordering H.

5.3.3 Naive algorithm

A naive algorithm to solve the string consensus problem is presented in Figure 5.2. The length k prefixes and suffixes of each pair of strings $S_1,\ S_2 \in \mathcal{S}$ are compared. If a prefix and suffix match, the strings S_1 and S_2 are removed from $\mathcal{S}$ and the string that results when S_1 and S_2 are joined is added to $\mathcal{S}$. Note that when a string is added to $\mathcal{S}$ it will be used in future prefix/suffix comparisons. When no more prefix/suffix matches of length k exist in $\mathcal{S}$ and $|\mathcal{S}| > 1$, k is incremented and the prefix/suffix matching of the strings in $\mathcal{S}$ is repeated.

We will assign a comparison to the string S if a character in a suffix of S is compared to a character in the prefix of some T. For each iteration of the outer for loop, each $S_b \in \mathcal{S}$, the prefix of S_b will be compared to the suffix of S_a and since the sum of the lengths of the length k prefixes of the S_b's is less than or equal to N, the number of comparisons done for each iteration of the outer for loop will be $O(N)$. The algorithm in Figure 5.2 has a worst case running time of $O(C_H N)$.

In the average case we expect that almost all prefix/suffix comparisons either match, or disagree after looking at a small constant number of characters. The expected time for the if statement in the inner for loop is constant. Since

```
consensus_naive ($\mathcal{S}$)

k $\leftarrow$ minimum match length acceptable - 1
while $|\mathcal{S}| > 1$
    k $\leftarrow$ k + 1
    for each $S_a \in \mathcal{S}$
        for each $S_b \in \mathcal{S}$
            if suffix ($S_a$, k) = prefix ($S_b$, k)
                remove $S_a$ and $S_b$ from $\mathcal{S}$
                add (join ($S_a$, $S_b$)) to $\mathcal{S}$
            if suffix ($S_b$, k) = prefix ($S_a$, k)
                remove $S_a$ and $S_b$ from $\mathcal{S}$
                add (join ($S_b$, $S_a$)) to $\mathcal{S}$
```

Figure 5.2: A naive algorithm that will solve the string consensus problem by searching all possible prefix/suffix matches.

there are n strings in $\mathcal{S}$ and the outer for loop will execute C_H times, the expected time for the algorithm in Figure 5.2 is $O(nC_H)$.

The algorithm in Figure 5.2 has been implemented in C on a Sun 3/260. Figure 5.3 shows the results of running the program with strings of length 507. The length of the overlaps between strings was between 180 and 200 characters. The number of strings was varied between 25 and 500 and the amount of CPU time used to compute the consensus string is plotted. Since the length of the strings and the overlap was held constant we will get a constant amount of compression for each string and therefore we expect the time to grow as $O(n^2)$. Figure 5.4 shows the square root of the running time plotted against the number of strings.

5.3.4 Rabin–Karp type algorithm

Given two strings S and T, $i = |S|,\ j = |T|,\ i < j$, the Rabin–Karp algorithm for string searching computes a hash value for the shorter string, S and a hash value for each length i substring of T. The hash value of S is compared with the hash values for the substrings of T. By cleverly choosing the hashing function, the hash value for the substring of T ending at position h can be computed from the hash value for the substring of T ending at position $h-1$ and the character t_h.

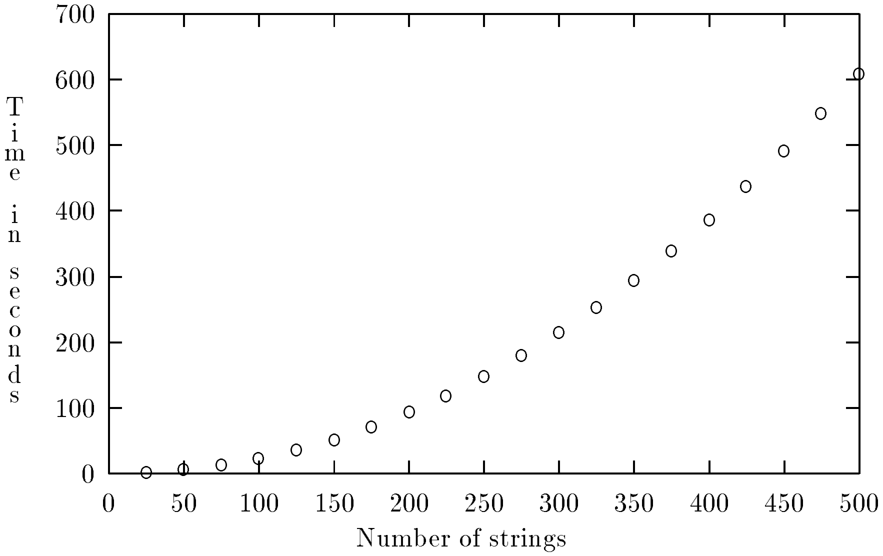


Figure 5.3: The running time in CPU seconds for the naive algorithm to solve the string consensus problem.

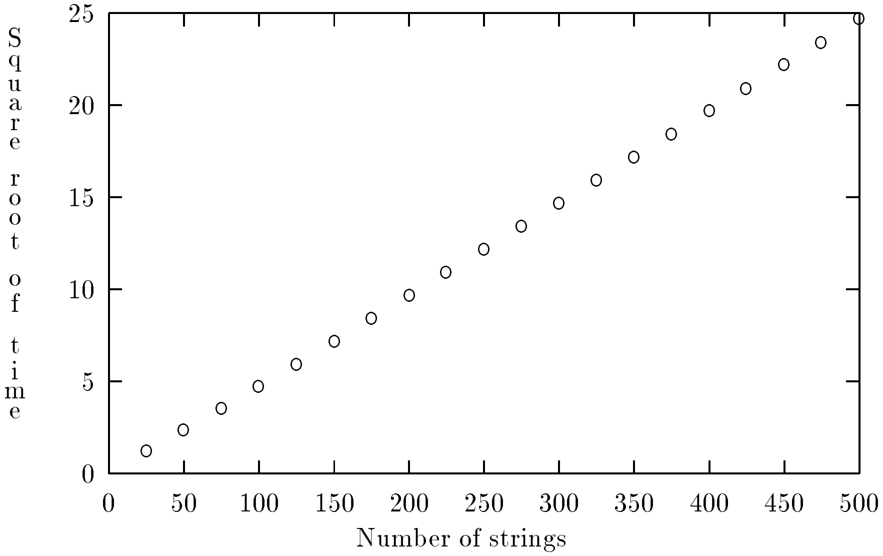


Figure 5.4: The square root of the running time for the naive algorithm to solve the string consensus problem.

The hash function Rabin and Karp [158] choose was $\mathsf{hash}(m) = m \bmod p$ where p is a large prime and m is an integer representation of the string $T_{h-i+1} \dots T_h$. To compute the hash value of the length i substring of T ending at position h

$$\mathsf{hash}(h) = ((\mathsf{hash}(h-1) - \mathsf{index}(t_{h-i}) \cdot \sigma^{i-1}) \cdot \sigma + \mathsf{index}(t_h)) \bmod p$$

Associated with each $c \in \Sigma$ is a unique integer $l,\ 0 \leq l < \sigma$. The function $\mathsf{index}(c)$ returns the integer associated with c.

The probability of two strings drawn randomly from Σ^i having the same hash value is shown by Gonnet & Baeza–Yates [118] to be

$$\frac{1}{p} + O\left(\frac{1}{\sigma^i}\right).$$

With the appropriate choices of p and k, the frequency of collisions will be small.

The Rabin–Karp algorithm has two properties that make it particularly well suited to the exact shotgun sequencing problem.

- The hash value for the prefix and suffix of a string can be computed incrementally. Given the hash value for the length i prefix of a string, the hash value for the length $i+1$ prefix can be computed with a constant number of operations.
- The hash value of a substring can be incrementally computed equally well from the right or the left end of the substring.

Figure 5.5 gives an algorithm based on the ideas of Rabin and Karp for the exact shotgun sequencing problem. Since the strings we are comparing are always prefixes (suffixes), we do not need to subtract the value of the leading (trailing) character of the string. The j^{th} forward hash value of the string S, (fhv$[S]$), is the hash value of the string $s_1 s_2 \dots s_j$. The function we use to compute the j^{th} fhv$[S]$ from the $(j-1)^{st}$ fhv$[S]$ is

$$\mathrm{fhv}[S] = (\mathrm{fhv}[S] \cdot \sigma + \mathsf{index}(s_j)) \bmod p.$$

```
consensus_RK ($\mathcal{S}$)

dm ← 1
k ← minimum suffix–prefix length
compute initial values of fhv and bhv
while $|\mathcal{S}| > 1$
    hashtree ← nil
    for each S ∈ $\mathcal{S}$
        j ← $|S|$ - k + 1
        fhv[S] ← (fhv[S] * $\sigma$ + index ($s_k$)) mod p
        bhv[S] ← (bhv[S] + (index ($s_j$)·dm)) mod p
        if (fhv[S] ∉ hashtree) add (fhv[S], hashtree)
        else join (S, matched string from hash tree)
        if (bhv[S] ∉ hashtree) add (bhv[S], hashtree)
        else join (S, matched string from hash tree)
    dm ← (dm·$\sigma$) mod p
    k ← k + 1
```

Figure 5.5: An algorithm based on Rabin–Karp style string matching to solve the string consensus problem.

Let $i = |S| - j + 1$. The j^{th} backward hash value of the string S, (bhv[S]), is the hash value of the string $s_i s_{i+1} s_{|S|}$. The function we use to compute the j^{th} bhv[S] from the $(j-1)^{th}$ bhv[S] is

$$\text{bhv}[S] = ((\mathsf{index}(s_i) \cdot \sigma^{j-1}) + \text{bhv}[S]) \bmod p.$$

The initial length k forward and backward hash values are computed for each $S \in \mathcal{S}$. Each fhv[S] and bhv[S] is added to a binary tree. Whenever the hash value being added to the tree matches a hash value already in the tree, the associated strings are compared. If the prefix of one matches the suffix of the other, the strings are removed from $\mathcal{S}$ and joined, the resulting string is added back to $\mathcal{S}$. When all of the possible prefixes and suffixes of length k have been joined, the value of k is incremented, the search tree is cleared, and the hash values for the new value of k are computed.

If we assume that no collisions occur, the number of operations used by the Rabin–Karp type algorithm for the exact shotgun sequencing problem is

$$O(C_H \cdot \log n).$$

The lines of the inner for loop will be executed once for each position of compression in the final consensus sequence. The “if (fhv[S] $\notin$ hashtree) add (fhv[s], hashtree)” and “if (bhv[S] $\notin$ hashtree) add (bhv[s], hashtree)” lines take $O(\log n)$ time to execute and each of the other lines of the inner loop takes constant time, therefore the running time of the algorithm in Figure 5.5 is $O(C_H \cdot \log n)$.

The hash tree is set up as a binary tree. Associated with each node is a hash value and a linked list of pointers to each string that has the hash value associated with the node. When a string has the same hash value as a node, a linear search is performed on the strings associated with the node and if a match is found, it is returned. If no match is found, the string is added to the linked list. In the extraordinary case where each string hashes to the same value, the algorithm performs just as the naive algorithm does.

The algorithm in Figure 5.5 has been implemented in C on a Sun 3/260. Figure 5.6 shows the results of running the program with strings of length 507. The length of the overlaps between strings was between 180 and 200 characters. The minimal acceptable overlap, k, was set to 20. The number of strings was varied between 25 and 1000 and the amount of CPU time used to compute the consensus string is plotted. Figure 5.7 shows the time divided by the log of the number of strings as the number of strings was varied.

5.3.5 Algorithm based on sorting

In the naive algorithm, a great deal of time is spent searching for a string in $\mathcal{S}$ with a particular prefix. If the list of prefixes were sorted, the time needed to search $\mathcal{S}$ for a string with a particular suffix could be significantly reduced. In this section we will use a trie to speed the search.

By sorting the strings using a bucket sort and keeping track of the positions of the buckets at each stage of the sort, we can find the strings with a prefix of length i using $O(i)$ operations. During the standard bucket sort of the strings, we will build a trie [161] where a node at depth i represents a bucket containing all

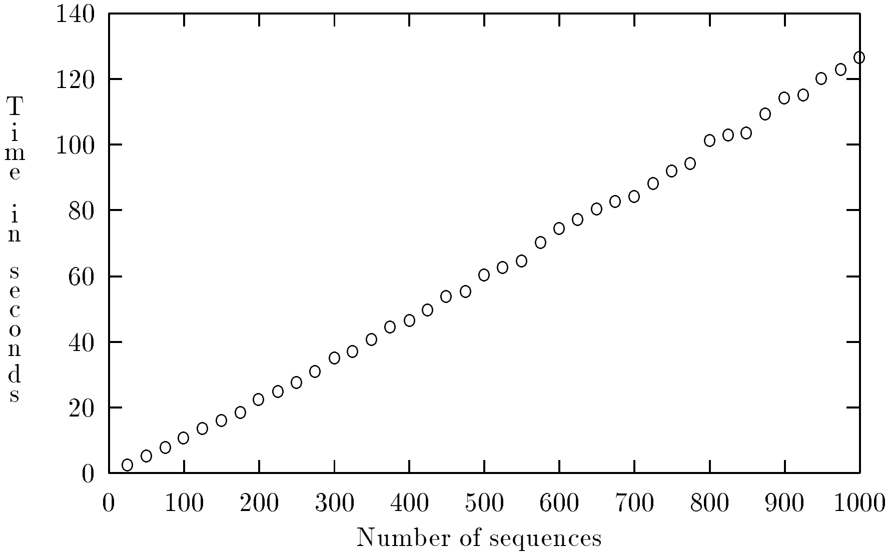


Figure 5.6: The running time in CPU seconds for the Rabin–Karp based algorithm to solve the string consensus problem.

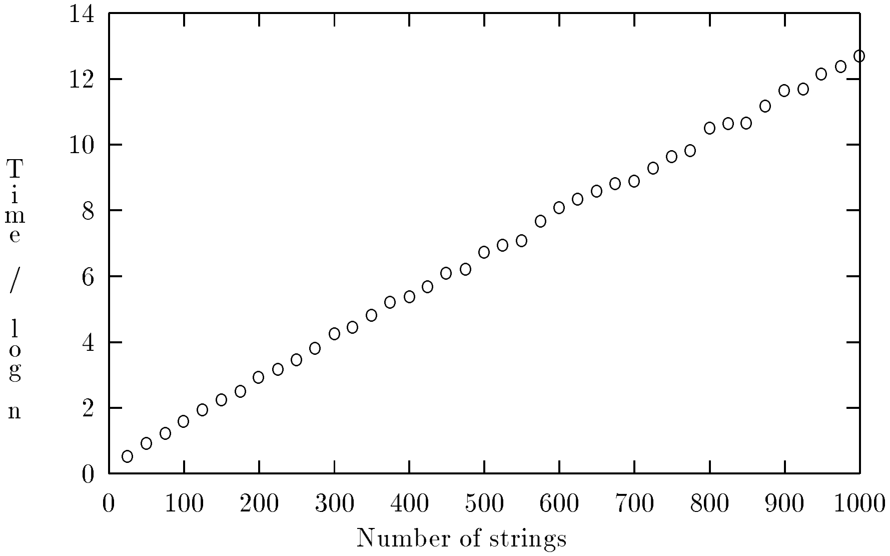


Figure 5.7: The running time divided by the log of the number of strings for the Rabin–Karp based algorithm to solve the string consensus problem.

```
consensus_trie ($\mathcal{S}$)

trie $\leftarrow$ sort_seqs ($\mathcal{S}$, 0)
k $\leftarrow$ minimum match length acceptable
while $|\mathcal{S}| > 1$
    for each S $\in \mathcal{S}$
        R $\leftarrow$ search (S, trie, k)
        if R $\neq \Lambda$
            T $\leftarrow$ join (R, S)
            add T to $\mathcal{S}$
            remove R and S from $\mathcal{S}$
    k $\leftarrow$ k + 1
print one remaining entry in $\mathcal{S}$
```

Figure 5.8: An algorithm based the the trie data structure to solve the string consensus problem.

strings in $\mathcal{S}$ with a particular prefix of length i. The children of a depth i node represent the strings in $\mathcal{S}$ with prefixes of length $i+1$ and the strings associated with the parent node and child nodes agree in positions 1 through i.

The function sort_seqs in Figure 5.9 recursively sorts the sequences and builds the trie. When $|\mathcal{S}| > 1$, the strings in $\mathcal{S}$ are sorted by the position indicated by the variable "column". The strings are then divided into at most σ groups, one group for each distinct character appearing at position "column" in some $S \in \mathcal{S}$. A node in the trie is created for each of the non–empty groups of strings. Each of these nodes is a child of the node representing $\mathcal{S}$. The function sort_seqs is then called recursively for each of the groups of strings.

The function sort_by_position orders the strings passed to it by the characters in the position passed to it and returns a node of the trie that points to the beginning of σ buckets along with the size of each bucket. The function sort_by_position takes time proportional to the number of strings being sorted. The time to sort the strings and construct the trie is proportional to N, the sum of the lengths of the strings, since each character of the strings in $\mathcal{S}$ is compared at most once.

The algorithm used to search the trie is given in Figure 5.10. It is essentially

```
    sort_seqs ($\mathcal{S}$, column)
/* column is the position in the strings that the strings are to be ordered by */
    root $\leftarrow$ nil
    if $|\mathcal{S}| > 1$
        root $\leftarrow$ sort_by_position ($\mathcal{S}$, column)
        for each i, $0 \leq$ i $< \sigma$
            $\mathcal{S}_i \leftarrow$ root.bucket_pos$_i$
            root.child$_i \leftarrow$ sort_seqs ($\mathcal{S}_i$, column + 1)
    return root
```

Figure 5.9: An algorithm to sort the string and construct the trie as the strings are being sorted.

```
    search (S, trie, j)

 /* Traverse trie */

    while (trie $\neq \Lambda$) $\wedge$ (j $>$ 0)
        prev $\leftarrow$ trie
        c $\leftarrow$ s$_{|S|-j}$
        trie $\leftarrow$ trie.next$_c$
        j $\leftarrow$ j - 1
    j $\leftarrow$ j + 1
    T $\leftarrow$ prev.bucket_pos$_c$
    c$_s \leftarrow$ s$_{|S|-j}$
    h $\leftarrow$ 0
    c$_t \leftarrow$ t$_{h-j}$

 /* Compare strings */

    while (c$_s \neq$ c$_t$) $\wedge$ (j $>$ 1)
        j $\leftarrow$ j - 1
        h $\leftarrow$ h + 1
        c$_s \leftarrow$ s$_{|S|-j}$
        c$_t \leftarrow$ t$_h$
    if (c$_s$ = c$_t$) $\wedge$ (j = 1) return T
    else return nil
```

Figure 5.10: An algorithm to search the trie for a string.

just a σ–ary tree search for the length j suffix of the string S. It assumes that you can index by the characters in Σ in constant time. If it is not possible to index by the characters in Σ then an $O(\log|\Sigma|)$ search would have to be used to index the buckets and trie pointers. The worst case time required to determine if a prefix of length l exists in the $|\mathcal{S}|$ strings is $O(l)$. This can be seen by noting that in the search procedure, at most one comparison is done at each character position and only characters in the prefix are compared.

The algorithm to find the consensus sequence using the sorted list of strings and the trie is given in Figure 5.8. For a given value of k, the algorithm searches the sorted list of prefixes for matching length k suffixes. If a match is found, the strings are removed from $\mathcal{S}$, joined, and the result of the join is added back to $\mathcal{S}$. When all of the the suffixes of length k have been searched for and more than one string remains in $\mathcal{S}$, k is incremented and the process is repeated.

In the worst case, the branching factor on the trie is nearly one and the time to build the consensus string is

$$O(\sigma n C_H).$$

Each iteration of the inner loop results in one character of compression. When the branching factor is near to σ, the expected time to build the consensus string is

$$O(C_H \log n).$$

The algorithm in Figures 5.8, 5.9 and 5.10 has been implemented in C with a Sun 3/260. Figure 5.11 shows the results of running the program on strings of length 507. The length of the overlaps between strings was between 180 and 200 characters. The minimum acceptable overlap, k, was set to 20. The number of strings was varied between 250 and 10000 and the CPU time used to compute the consensus string is plotted. Figure 5.12 shows the time divided by the log of the number of strings as the number of strings was varied.

In this chapter I gave algorithms to assemble shotgun sequence data into

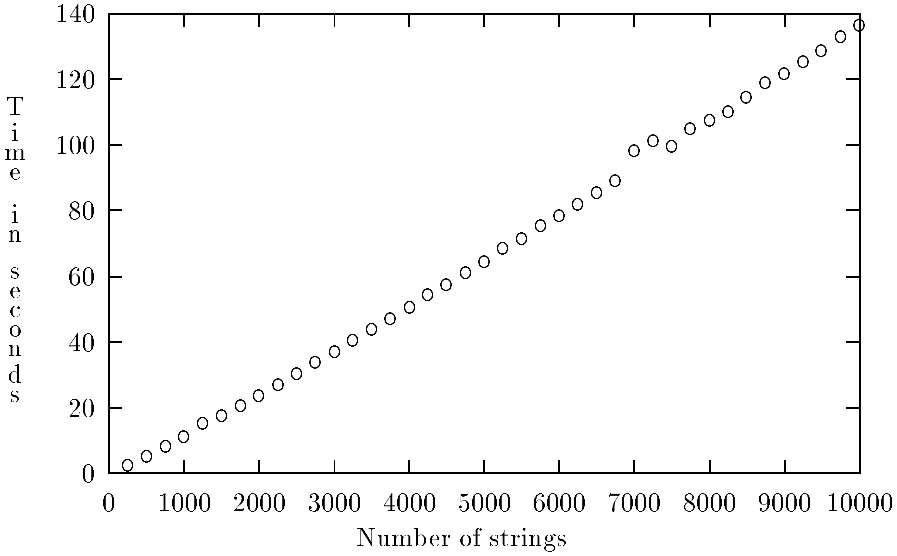


Figure 5.11: The running time in CPU seconds for the sort based algorithm to solve the string consensus problem.

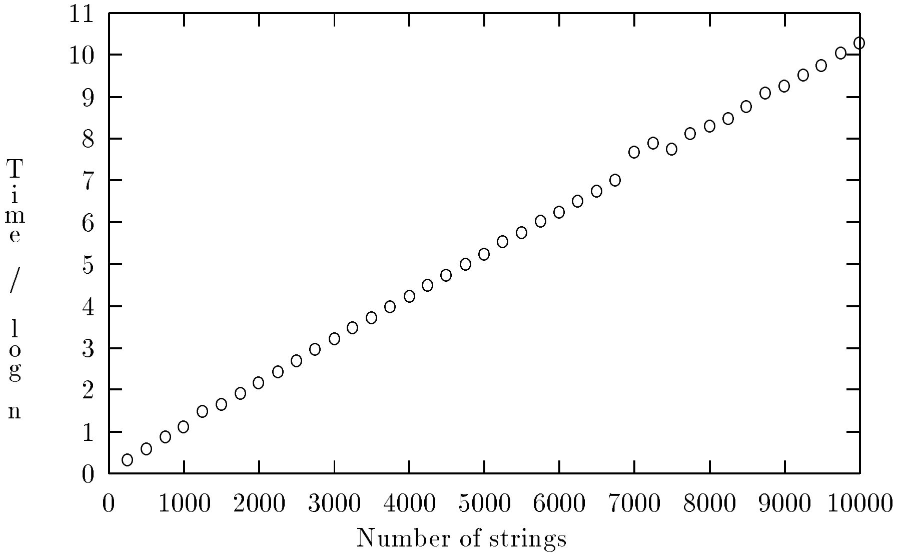


Figure 5.12: The running time divided by the log of the number of strings for the sort based algorithm to solve the string consensus problem.

the original sequences. I have assumed that all overlaps between prefixes and suffixes of the seuqence fragment data are of length at least k and match exactly. These assumptions allow me to construct algorithms that run in time proportional to the product of the amount of compression and the log of the number of sequence fragments. The algorithms that are given are capable of reconstructing sequences from tens of thousands of sequence framents. In the following chapter I give algorithms to reconstruct sequences from shotgun sequence data assuming the overlaps do not match exactly.

Chapter 6

Log Inexact Shotgun Sequencing

6.1 Introduction

The process of sequencing DNA and RNA is not perfect and mistakes are occasionally made in processing the gels, reading the gels and entering the data into a database. In this section, we will not assume that all suffix/prefix overlaps match perfectly. Instead we will suppose that there are at most $\log(v)$ positions where the suffix and prefix do not match where v is the length of the overlap between the two strings.

We will discuss three algorithms to solve the log inexact string consensus problem. The first is a naive algorithm that we present for comparison. The second algorithm is based on the Rabin–Karp string matching algorithm. The final algorithm that we present is based loosely on the sorting algorithm presented in section 5.3.5.

The algorithm based loosely on the Rabin–Karp string matching algorithm returns, with high probability, the correct solution in

$$O(Cn(\log\log^2 p) + n\log p)$$

time where p is some large prime, usually chosen to fit in a small number of words of memory. The worst case time bound for the Rabin–Karp based log inexact string matching algorithm is

$$O(nC_H \log p)$$

The large table required by the Rabin Karp algorithm makes it useful only for consensus sequences with very short prefix/suffix overlaps.

The third algorithm, based very loosely on the sorting algorithm presented in section 5.3.5, returns a correct answer in

$$O\left(\frac{N^3 \log k}{k}\right)$$

worst case time and

$$O\left(\frac{nN \log k}{|\Sigma|^{\frac{k}{\log k}}} + N\right)$$

expected time. The sorting based algorithm for the log inexact consensus sequence problem seems to be practical for a variety of problems.

We will say that the two length n strings S and T “log inexact match” or $S \approx T$ if

$$\sum_{i=0}^{n-1}(s_i \oplus t_i) \leq \log n$$

where

$$s_i \oplus t_i = \begin{cases} 0 & \text{if } s_i = t_i \\ 1 & \text{if } s_i \neq t_i \end{cases} .$$

6.1.1 Assumptions

We will make the following assumptions and then define the log inexact shotgun sequencing problem.

- An integer k can be supplied that defines the minimum acceptable overlap between two strings.
- There is a unique alignment of the strings in $\mathcal{S}$ such that all suffix/prefix overlaps are of length k or greater.
- All suffix/prefix overlaps are log inexact matches.

```
naive ($\mathcal{S}$, k)
    while $|S| > 1$
        for each $S_i \in \mathcal{S}$
            for each $S_j \in \{\mathcal{S} - S_i\}$
                if verify $(S_i, S_j, k)$
                    add (join $(S_i, S_j), \mathcal{S}$)
                    remove $(S_i, \mathcal{S})$
                    remove $(S_j, \mathcal{S})$
        k $\leftarrow$ k + 1
```

Figure 6.1: A naive algorithm that will check all length k prefix/suffix overlaps to solve the log inexact consensus string problem

6.1.2 Problem definition

We are given a multiset of strings, $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, and an integer k. We make the following assumptions about $\mathcal{S}$ and k

1. S_i is not a substring of S_j for $1 \leq i,\ j \leq n,\ \ i \neq j$.

2. An ordering, H, of the strings in $\mathcal{S}$ exists such that

$$\forall_{1 \leq i < n}\, i\ \ \exists_{j \geq k}\, j\ \ \mathsf{suffix}(S_{H_i}, j) \approx \mathsf{prefix}(S_{H_{i+1}}, j).$$

The problem is, given the multiset $\mathcal{S}$ and the integer k, find the ordering H.

6.2 Naive log inexact algorithm

The naive algorithm presented in Figure 6.1 solves the log inexact consensus sequence problem. The algorithm simply tries all possible length k prefix/suffix log inexact matches, merges the log inexact matches that are found and increments k. This process is iterated until the only string remaining in $\mathcal{S}$ is the single consensus string.

Theorem 6.1 *The algorithm presented in Figure 6.1 uses*

$$O(nC_HN)$$

time in the worst case to compute the log inexact consensus string.

Proof. Each iteration of the outer for loop increases the compression by one. The time for $\mathsf{verify}(S_i, S_j, k)$ will be at most $\mathsf{min}(|S_i|, |S_j|)$ since we can use a simple character by character comparison of the strings to determine if they log inexact match. The time to complete one iteration of the outer for loop will be no more than nN. So the time to compute the log inexact consensus sequence will be $O(nC_H N)$. ▮

Theorem 6.2 *The algorithm presented in Figure 6.1 uses*

$$O(nC_H \log N)$$

time in the average case to compute the log inexact consensus string.

Proof. Each iteration of the outer for loop increases the compression by one. The expected time for a verify call that fails is $O(\log N)$ since we need to find only $\log k$ errors and k may be as large as the longest string in $\mathcal{S}$. The total time spent in successful calls to verify is $O(C_H)$. Since the outer for loop is iterated once for each character of compression, the expected time is $O(nC_H \log N)$. ▮

If the overlap between adjacent strings in the log inexact consensus sequence and string length are both held constant, the worst case performance of the naive algorithm is $O(n^3)$ and the expected case is $O(n^2 \log n)$.

The naive algorithm was implemented in C and run on a Sun 3/260. Figure 6.2 shows the running time as the number of strings is varied from 4 to 120. Each string was 100 characters long and had an overlap length of 50 with its neighbor in the log inexact consensus sequence. The minimum overlap accepted (k) was 24 characters. Figure 6.3 shows the square root of the running times presented in Figure 6.2.

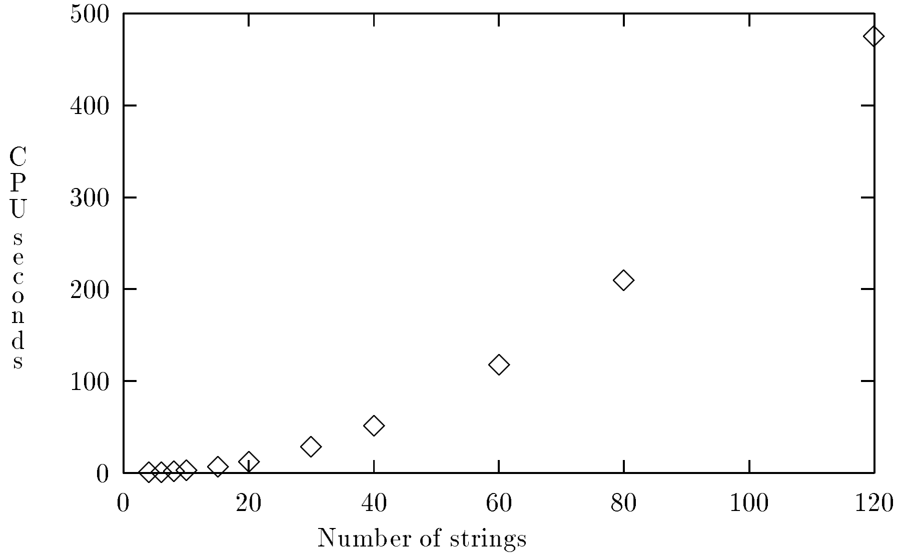


Figure 6.2: The running time in CPU seconds used by the naive algorithm to solve the log inexact consensus string problem.

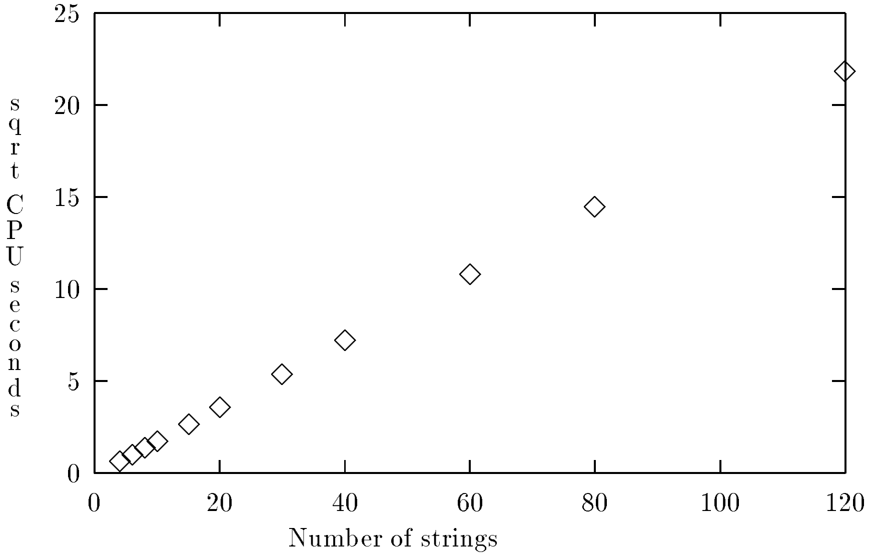


Figure 6.3: The square root running time used by the naive algorithm to solve the log inexact consensus string problem

```
enum–diffs (XOR–value, x, y, position)
    if bitposition (position, XOR–value) is set
        enum–diffs (XOR–value - $2^{\text{position}}$, x + $2^{\text{position}}$, y, position + 1)
        enum–diffs (XOR–value - $2^{\text{position}}$, x, y + $2^{\text{position}}$, position + 1)
    else if XOR–value > 0
        enum–diffs (XOR-value, x, y, position + 1)
    else if x > y
        print (x - y)
```

Figure 6.4: An algorithm to convert the XOR value of two numbers to the differences of two numbers that would result in the XOR value.

6.3 Rabin Karp log inexact algorithm

We can use the ideas developed in section 5.3.4 and extend the definition of string equality to build an algorithm that will solve the log inexact shotgun sequencing problem. Let the function F have the domain of all $S \in \Sigma^*$ and the range of non–negative integers. Let $\sigma = |\Sigma|$. $F(S)$ will be the base σ integer representation of the string S. Given two strings S and T, from Σ^q we let

$$\begin{aligned} F(S) &= \mathsf{index}(s_1) \cdot \sigma^{q-1} + \mathsf{index}(s_2) \cdot \sigma^{q-2} + \cdots + \mathsf{index}(s_q) \\ F(T) &= \mathsf{index}(t_1) \cdot \sigma^{q-1} + \mathsf{index}(t_2) \cdot \sigma^{q-2} + \cdots + \mathsf{index}(t_q). \end{aligned}$$

When there is exactly one position i such that $s_i \neq t_i$, $F(S)$ and $F(T)$ will differ by $d\sigma^{q-i}$, $0 \leq d < \sigma$. If $s_i \neq t_i$ and $s_j = t_j$ for all j, $0 \leq j < i$, $i < j \leq q$, then

$$F(S) = F(T) + (\mathsf{index}(s_i) - \mathsf{index}(t_i))\sigma^{q-i}$$

If the two strings S and T differ in positions i_1 and i_2 then $F(S)$ and $F(T)$ will differ by some additive combination of $d_1\sigma^{q-i_1}$ and $d_2\sigma^{q-i_2}$, $1 \leq d_1, d_2 \leq \sigma$, and so on for larger numbers of differences.

Before the log inexact consensus string is computed, a table of differences between $F(S)$ and $F(T)$ for all $S \approx T$ must be computed. A list of integers, XORlist, is generated. The binary representation of each integer in the list has at most $\log p$ ones, each one representing the position of a mismatch between S and T. Since the mod operator can be distributed over addition, but mod can not be distributed over XOR, the XORlist must be converted into a list of differences. For each integer in the XORlist, there may be several different values of S and T that will generate $F(S)$ XOR $F(T)$. The table of differences is the set of all positive differences between $F(S)$ and $F(T)$ where $F(S)$ XOR $F(T)$ is in the XORlist. The algorithm in Figure 6.4 will return the set of all possible values $F(S) - F(T)$ such that XORvalue $= F(S)$ XOR $F(T)$. The enum–diffs algorithm in Figure 6.4 is initially called as

$$\mathsf{enum-diffs}(F(S)\ \mathrm{XOR}\ F(T), 0, 0, 0).$$

The RK log inexact algorithm presented in Figure 6.5 will find an ordering of the strings in $\mathcal{S}$ that solves the log inexact consensus string problem. In this algorithm, the length of the prefix/suffix matches that are examined will be limited to $\log p$ where p is the prime number chosen as the modulo for the hash values. The algorithm first computes the forward and backward hash values for each $S \in \mathcal{S}$ for the length k prefixes and suffixes modulo p (fhv[S] and bhv[S] respectively). The while loop will proceed as long as two conditions hold, there is more than one string in $\mathcal{S}$ and the length of the prefixes being examined is less than $\log p$ times some constant. Each iteration of the while loop looks for prefix/suffix matches of length one greater than the previous iteration and starts at k, the minimal acceptable prefix/suffix match length. The two for loops simply select pairs of strings to be examined. The forward and backward hash values are computed for strings being compared and then the difference between the two hash values is computed. The errorlist is searched for the difference and if the difference is found, it is verified that the two strings do have a prefix/suffix log inexact match of length at least k. Once the prefix/suffix log inexact match has been verified, the strings are removed from $\mathcal{S}$, joined, and the joined string is returned to $\mathcal{S}$.

6.3.1 Reliability

The algorithm in Figure 6.5 relies on the values $F(S) \bmod p - F(T) \bmod p$, $S \approx T$, to be sparsely distributed between 0 and $p-1$. To estimate the running time

```
match ($\mathcal{S}$, k, errorlist)

for each $S \in \mathcal{S}$
    $\text{fhv}[S] \leftarrow \mathsf{index}(s_1) \cdot \sigma^{k-1} + \mathsf{index}(s_2) \cdot \sigma^{k-2} + \cdots + \mathsf{index}(s_k) \pmod p$
    $\text{bhv}[S] \leftarrow \mathsf{index}(s_{|S|-k+1}) \cdot \sigma^{k-1} + \mathsf{index}(s_{|S|-k+2}) \cdot \sigma^{k-2} + \cdots + \mathsf{index}(s_{|S|}) \pmod p$
dm $\leftarrow$ 1
while $(|\mathcal{S}| > 1) \wedge (k < c \cdot \log p)$
    k $\leftarrow$ k + 1
    for each $S \in \mathcal{S}$
        $\text{fhv}[S] \leftarrow \text{fhv}[S] * \sigma + \mathsf{index}(s_k) \pmod p$
        for each $T \in (\mathcal{S} - S)$
            $\text{bhv}[T] \leftarrow \text{bhv}[T] + \mathsf{index}(t_{|T|-k}) * \text{dm} \pmod p$
            diff $\leftarrow$ $|\text{fhv}[S] - \text{bhv}[T]|$
            if diff $\in$ errorlist
                if verify (S, T, k)
                    add (join $(S, T)$, $\mathcal{S}$)
                    remove ($S$, $\mathcal{S}$)
                    remove ($T$, $\mathcal{S}$)
    dm $\leftarrow$ dm * $\sigma \pmod p$
```

Figure 6.5: An algorithm based on the Rabin–Karp method of strings matching to solve the log inexact consensus string problem.

of this algorithms we need to know how frequently $S \not\approx T$ and $F(S) \bmod p - F(T) \bmod p \in$ errorlist. We will show that the number of error values when $S_1 \approx S_2$ is small compared to the complete range of error values. Therefore, when $S_1 \not\approx S_2$, it is unlikely that $F(S_1) \bmod p - F(S_2) \bmod p \in$ errorlist.

Lemma 6.1 *Given q pairs of strings, $<S_1,\ S_2>$, where S_1 and S_2 are randomly selected from $\Sigma^{\log q}$ such that $S_1 \not\approx S_2$, we expect $\log\log q \cdot \log q^{\log\log q}$ of the q pairs to have $F(S_1) - F(S_2) \in$ errorlist.*

Proof. We know that

$$\begin{pmatrix} x \\ \log x \end{pmatrix} = \frac{x \cdot (x-1) \cdot \cdots \cdot (x - \log x + 1)}{\log x \cdot (\log x - 1) \cdot \cdots \cdot 2} \leq x^{\log x}$$

so the number of error values in the error list is at most

$$\sum_{i=0}^{\log\log q} \begin{pmatrix} \log q \\ i \end{pmatrix} \leq \sum_{i=0}^{\log\log q} \begin{pmatrix} \log q \\ \log\log q \end{pmatrix} \leq \log\log q \cdot \log q^{\log\log q}.$$

Since we are assuming that the hashing function is randomly distributing the hash values from $0 \ldots q-1$, the number of times that $F(S_1) \bmod p - F(S_2) \bmod p \in$ errorlist will be about the same as the number of values in errorlist. ▮

6.3.2 Running time

The log inexact RK algorithm in Figure 6.5 does not build a tree of prefix values as the exact RK algorithm (See Figure 5.5) does, but must scan the entire list of prefix values sequentially since strings that are nearly the same may have very different hash values. If we let v be the maximum length suffix/prefix overlap and limit the number of mismatches to $O(\log v)$, the number of error values is at most $O(v^{\log v})$ since each error value is a combination of $\log v$ or fewer powers of σ, $\{\sigma^0,\ \sigma^1,\ \ldots,\ \sigma^v, -\sigma^1,\ \ldots,\ -\sigma^v\}$ multiplied by some i, $i < \sigma$. These values can be precomputed and sorted so that the time to search the list of error values for a particular value is $O(\log^2 v)$.

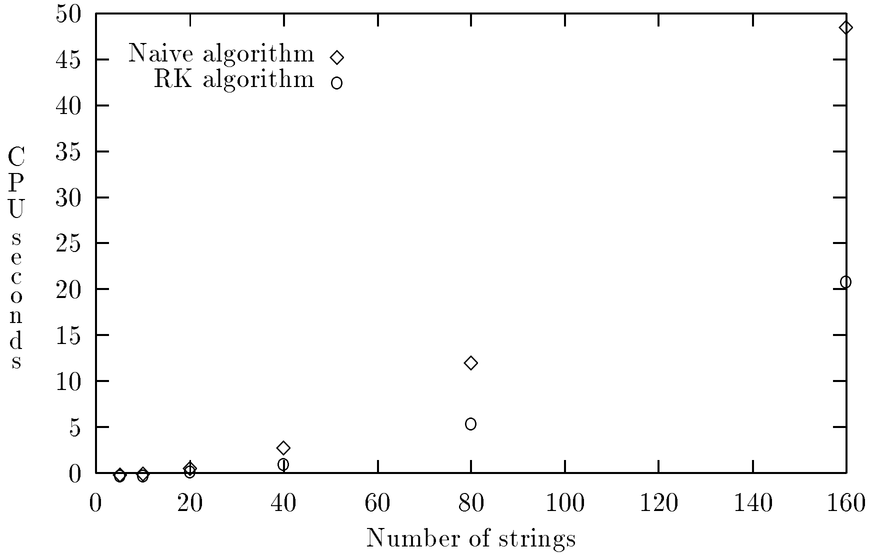


Figure 6.6: The running times in CPU seconds for the naive and RK algorithms to solve the log inexact consensus string problem.

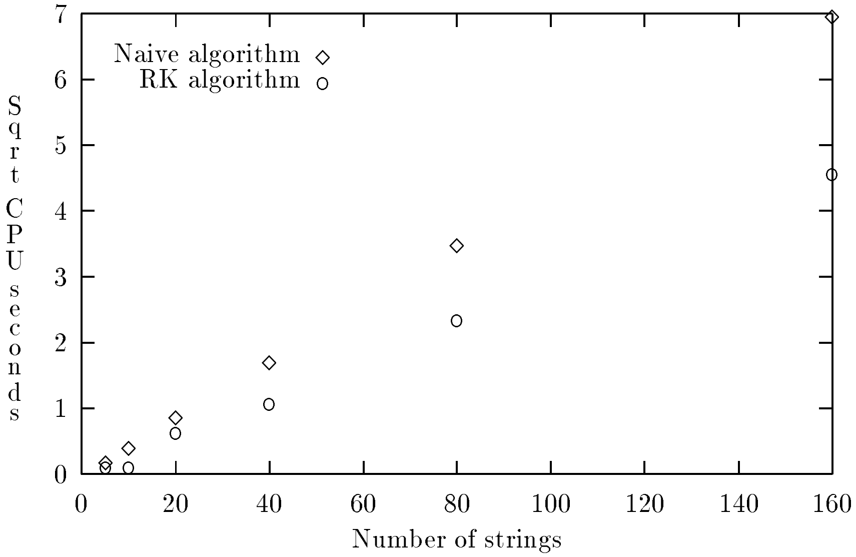


Figure 6.7: The square root of running times for the naive and RK algorithms to solve log inexact consensus string problem.

Theorem 6.3 *The algorithm presented in Figure 6.5 uses*

$$O(Cn(\log\log^2 p) + n\log p)$$

expected time to compute the log inexact consensus string when v, the maximum overlap length between strings, is at most $\log p$.

Proof. The for loop that initializes the forward and backward hash values executes in $O(nk)$. Each statement in the while loop will execute in constant time with the exceptions of the "if diff ∈ errorlist" statement and the "if verify (S, T, k)" call. Since the number of values in the errorlist is $O(v^{\log v})$ and v is at most $\log p$, the "if diff ∈ errorlist" statement will execute in $O((\log\log p)^2)$ time. Each iteration of the outer for loop will result in one character of compression. At most n searches of the errorlist will be done for each character of compression. Since there are C_H characters of compression, the running time without the calls to verify will be $O(nC_H(\log\log p)^2)$. By lemma 6.1 we see that if we let $v = \log p$ then we expect $\log\log p(\log p)^{\log\log p}$ incorrect inexact matches for every p pairs of strings that we look at. Since we will be looking at nC_H pairs of strings the expected number of incorrect calls to verify will be $O\left(nC_H\frac{(\log p)^{\log\log p}\log\log p}{p}\right)$. The expected time to discover that two strings do not log inexact match is $O(\log\log p)$ since there can be at most $\log\log p$ mismatches and we expect to find each mismatch by looking at a constant number of positions. The time contributed by incorrect calls to verify will be $O\left((\log\log p)^2\left(nC_H\frac{(\log p)^{\log\log p}}{p}\right)\right)$. Each of the $n-1$ correct calls to verify will take $O(\log p)$ time. The calls to verify will take $O\left((\log\log p)^2\left(nC_H\frac{(\log p)^{\log\log p}}{p}\right) + n\log p\right)$ time. Adding these times we get the running time of the RK inexact match algorithm.

$$O\left[(\log\log p)^2\left(nC_H\frac{(\log p)^{\log\log p}}{p}\right) + n\log p + nC_H(\log\log p)^2\right]$$

Since $\frac{(\log p)^{\log\log p}}{p} \leq 1$

$$n(\log\log p)^2 C_H \geq (\log\log p)^2\left(nC_H\frac{(\log p)^{\log\log p}}{p}\right)$$

leading to the running time of

$$O(Cn(\log\log^2 p) + n\log p)\blacksquare$$

Theorem 6.4 *The algorithm presented in Figure 6.5 uses*

$$O(nC_H \log p)$$

worst case time to compute the log inexact consensus string when v, the length of the overlap between strings, is at most $\log p$.

Proof. The time used to initialize the forward and backward hash values and the number of iterations of the for loops is $O(nk)$. In the worst case, the "if diff $\in$ errorlist" statement is always true and verify must always be called. Therefore there will be nC searches of the errorlist and calls to verify. Since each search of the errorlist takes $O\left((\log\log p)^2\right)$ time and each call to verify could, in the worst case, take $O(\log p)$ time, the worst case time is

$$O(nC_H(\log\log p)^2 + \log p[nC_H + n - 1]) \text{ or}$$
$$O(nC_H \log p).$$

The algorithm given in Figure 6.5 to solve the log inexact string consensus problem was implemented in C on a sun 3/260. The value of p was set to 33,554,393; the value of v, the maximum overlap was set to 16, and k, the minimum acceptable overlap was set to 5. The running time of this algorithm, as well as the running time of the naive algorithm, is shown in Figures 6.6. Figure 6.7 shows the square root of the running times. $\blacksquare$

6.3.3 Increased match length

We can increase the effective size of p by using the Chinese remainder theorem. Let p_1, p_2 $\ldots$, p_m be pairwise relatively prime positive integers, then the system

```
match ($\mathcal{S}$, k, rprimes, errorlist)

for each $S \in \mathcal{S}$
    for each $p_i \in$ rprimes
        $\mathrm{fhv}[S, i] \leftarrow \mathsf{index}(s_1) \cdot \sigma^{k-1} + \mathsf{index}(s_2) \cdot \sigma^{k-2} + \cdots + \mathsf{index}(s_k) \pmod{p_i}$
        $\mathrm{bhv}[S, i] \leftarrow \mathsf{index}(s_{|S|-k+1}) \cdot \sigma^{k-1} + \mathsf{index}(s_{|S|-k+2}) \cdot \sigma^{k-2} + \cdots + \mathsf{index}(s_{|S|})$
                                        $\pmod{p_i}$
for each $p_i \in$ rprimes dm[i] $\leftarrow$ 1
while $(|\mathcal{S}| > 1) \wedge (k < c \cdot \log p)$
    k $\leftarrow$ k + 1
    for each $S \in \mathcal{S}$
        for each $p_i \in$ rprimes
            $\mathrm{fhv}[S, i] \leftarrow \mathrm{fhv}[S, i] * \sigma + \mathsf{index}(s_k) \pmod{p_i}$
        for each $T \in (\mathcal{S} - S)$
            for each $p_i \in$ rprimes
                $\mathrm{bhv}[T, i] \leftarrow \mathrm{bhv}[T, i] + \mathsf{index}(t_{|T|-k}) * \mathrm{dm}[i] \pmod{p_i}$
            for each $p_i \in$ rprimes
                diff[i] $\leftarrow$ $|\mathrm{fhv}[S, i] - \mathrm{bhv}[T, i]|$
            if diff $\in$ errorlist
                if verify (S, T, k)
                    add (join ($S, T$), $\mathcal{S}$)
                    remove ($S$, $\mathcal{S}$)
                    remove ($T$, $\mathcal{S}$)
    for each $p_i \in$ rprimes
        dm[i] $\leftarrow$ dm[i] * $\sigma \pmod{p_i}$
```

Figure 6.8: The extended RK based algorithm for the log inexact match problem.

```
log-inexact-suffix-array ($\mathcal{S}$, k)
    $S_{tot} \leftarrow S_1 \circ \gamma \circ S_2 \circ \gamma \circ \cdots \circ \gamma \circ S_n$
    build a suffix-array for $S_{tot}$
    for each string $S_i \in \mathcal{S}$
        position[i] $\leftarrow$ search-suffix-array ($S_i$, $S_{tot}$, k, suffix_array)
```

Figure 6.9: An algorithm to solve the log inexact match problem using suffix arrays.

of congruences

$$\begin{aligned} x &\equiv a_1 \pmod{p_1}, \\ x &\equiv a_2 \pmod{p_2}, \\ &\vdots \\ x &\equiv a_m \pmod{p_m}, \end{aligned}$$

has a unique solution modulo $p = p_1 p_2 \cdots p_m$. [229] With a few simple modifications to the algorithm in Figure 6.5 we can construct an algorithm that takes advantage of the Chinese remainder theorem. The extended RK inexact string matching algorithm is given in Figure 6.8. The analysis of reliability does not change since the system of congruences has a unique solution modulo p. The running time of the extended RK inexact algorithm increases by a factor of m.

6.4 Suffix array based log inexact algorithm

Two strings, S_i and S_j of length m, with at most $\log m$ positions that do not match must have a common substring S', $|S'| \geq \frac{m}{\log m}$. In this section we will develop an algorithm based on this simple observation to solve the log inexact shotgun sequencing problem.

6.4.1 Algorithm

Figures 6.9 and 6.10 give an algorithm for the log inexact shotgun sequencing problem. Given the set of strings $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ and an integer k that

```
search-suffix-array ($S_i$, $S_{tot}$, k, suffix_array)
    ptr ← 0
    blocksize ← $\left\lfloor \frac{\mathrm{k}}{\log \mathrm{k}} \right\rfloor$
    while ptr < $|S_i|$
        $\mathcal{P}$ = search ($S_i$ [ptr] ... $S_i$ [ptr + blocksize - 1], suffix_array)
        for each $P_j \in \mathcal{P}$
            if verify ($S_i$, $S_{tot}$ [$P_j$ - ptr])
                return $P_j$ - ptr
        ptr ← ptr + blocksize
    return nil
```

Figure 6.10: An algorithm to find log inexact matches using a suffix array.

specifies the minimum length overlap, the algorithm computes the log inexact consensus sequence. The string

$$S_{tot} = S_1 \circ \gamma \circ S_2 \circ \gamma \cdots \gamma \circ S_n,$$

where $\gamma \notin \Sigma$, is composed and a suffix array is created for S_{tot} (See [189]). Each string $S_i \in \mathcal{S}$ is positioned in S_{tot} so that a prefix of S_i log inexact matches a prefix of some suffix of S_{tot}. The log inexact match must be at least k characters in length and must not include the character γ. Knowing the position of each S_i in S_{tot} will allow us to easily construct a log inexact consensus sequence.

Positioning each $S_i \in S_{tot}$ is done using the algorithm in Figure 6.10. The string S_i is partitioned into $\frac{|S_i| \log k}{k}$ substrings, each of length $\frac{k}{\log k}$. While a log inexact match has not been found for S_i, the suffix array is searched for the substrings of S_i. For each exact match between a substring of S_i and a substring of S_{tot}, the location of the exact match is used to verify that a log inexact match exist between a prefix of S_i and a suffix of some S_j in S_{tot}.

6.4.2 Worst case running time

Theorem 6.5 *The worst case running time of the algorithm in Figures 6.9 and 6.10 is* $O\left(O\left(\frac{N^3 \log k}{k}\right)\right)$.

Proof. The worst case time to build the suffix array is $O(N \log N)$ [189]. The function verify can be computed in time linear in the size of the strings using a simple character by character comparison. For each $S_i \in \mathcal{S}$ there will be at most $N\frac{|S_i|\log k}{k}$ calls to verify since there will be at most N substrings returned from the function search and at most $\frac{|S_i|\log k}{k}$ calls to search will be made. The total time spent in the function verify will be at most

$$\begin{aligned} N|S_1|\frac{|S_1|\log k}{k} + N|S_2|\frac{|S_2|\log k}{k} + \ldots + N|S_n|\frac{|S_n|\log k}{k} \\ = \frac{\log k}{k}\sum_{i=1}^{n} N|S_i|^2 \\ = O\left(O\left(\frac{N^3 \log k}{k}\right)\right) \end{aligned}$$

Searching the suffix array (calling the function search) for S_i takes $O(|S_i| + \log|S_{tot}|)$ [189] time. For each S_i there will be at most $\frac{|S_i|\log k}{k}$ calls to search so the total time spent in search will be at most

$$\begin{aligned} \sum_{i=1}^{n}\left[\left(\frac{|S_i|}{\log|S_i|} + \log|S_{tot}|\right)\frac{|S_i|\log k}{k}\right] &= \frac{\log k}{k}\left(\sum_{i=1}^{n}\frac{|S_i|^2}{\log|S_i|} + \sum_{i=1}^{n}|S_i|\log N\right) \\ &\leq O(\frac{N^3\log k}{k}) \end{aligned}$$

6.4.3 Expected running time

The expected running time is significantly better than the worst case running time. Before we give the expected running time we need to prove the following lemma that will be used to show that we can expect to look at a constant number of the $P_j's$ in Figure 6.10.

Lemma 6.2 *Given m urns and $m-1$ balls, each ball placed in a randomly selected urn, the expected number of empty urns after each ball has been placed in an urn is me^{-1} as m goes to infinity.*

Proof. Let P_i be the probability that urn U_i is empty.

$$P_i = \left(1 - \frac{1}{m}\right)^{m-1}$$

The expected number of empty urns is

$$\begin{aligned}\sum_{i=1}^{m} P_i &= m\left(1-\frac{1}{m}\right)^{m-1} \\ &= \frac{m}{1-\frac{1}{m}}\left(1-\frac{1}{m}\right)^{m} \\ \lim_{m\to\infty}\left(\frac{m}{1-\frac{1}{m}}\left(1-\frac{1}{m}\right)^{m}\right) &= me^{-1} \quad \blacksquare\end{aligned}$$

Theorem 6.6 *The expected running time of the algorithm in Figures 6.9 and 6.10 is*

$$O\left(\frac{nN\log k}{|\Sigma|^{\frac{k}{\log k}}}+N\right).$$

Proof. S_i is segmented into $\log k$ pieces of length $\frac{k}{\log k}$. We are likely to find a segment of S_i that exactly matches some substring of S_{tot} looking at a constant number of segments. This can be seen by lemma 6.2, let $m = \log k$, treat each segment of S_i as an urn, and each of the $\log k$ errors in the log inexact match as a ball. From lemma 6.2 we expect $\frac{\log k}{e}$ segments to contain none of the $\log k$ errors. Therefore, since the expected ratio of segments with no errors to total segments is a constant, we expect to look at a constant number of the segments to find a match with no errors.

We expect that search will return

$$\frac{N}{|\Sigma|^{\frac{k}{\log k}}}$$

substrings of S_{tot} since we are assuming that all length $\frac{k}{\log k}$ strings are equally likely to be substrings of S_{tot}. Each call to search will take $O\left(\frac{k}{\log k}\right)$ time. We expect each unsuccessful call to verify to take $\log k$ time since at each position of the potential match there is a $\frac{|\Sigma|-1}{|\Sigma|}$ chance that the characters do not match and we only need to find $\log k$ positions where the characters do not match. to verify will take no more than comparisons. The total time spent in successful calls to the function verify will be less than $O(N)$. The time to build the suffix array is expected to be $O(N)$[189]. So, the expected time to solve the log inexact match

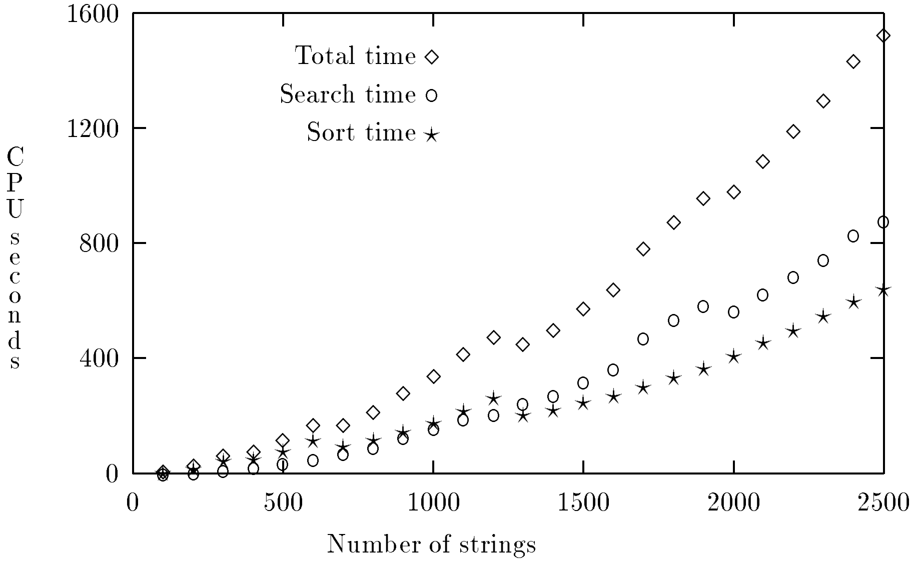


Figure 6.11: The running time in CPU seconds of the suffix array algorithm to solve the log inexact match problem while varying number of strings.

problem using the algorithm in Figures 6.9 and 6.10 is

$$O\left(\frac{nN\log k}{|\Sigma|^{\frac{k}{\log k}}}+N\right) \qquad \blacksquare$$

The seeming improvement as $|\Sigma| \rightarrow \infty$ is an artifact of the assumption that direct addressing via Σ is possible.

6.4.4 Implementation and Discussion

The algorithm discussed in section 6.4 has been implemented in C and run on a Sun 3/260. Figure 6.11 shows the running time of the algorithm presented in figures 6.9 and 6.10 as the number of strings is varied from 100 to 2500 while the length of the strings is 100, the minimum acceptable overlap is 24, and the overlap between adjacent strings is 50. Figure 6.12 shows the running time of the algorithm as the size of the strings is varied from 100 to 2000 characters while the number of strings is 100, the minimal acceptable overlap is 24, and the actual overlap is 50. Figure 6.13 shows the running time of the algorithm as the size of

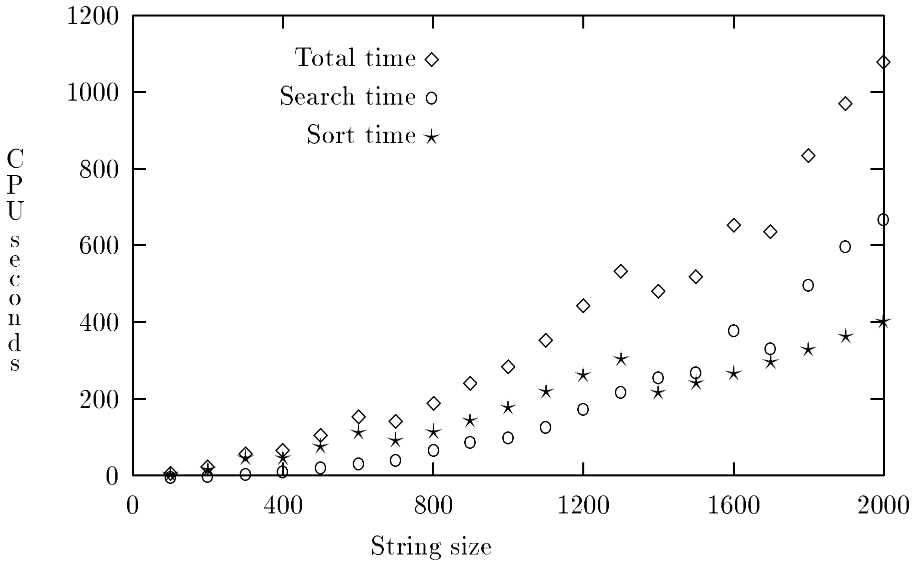


Figure 6.12: The running time in CPU seconds of the suffix array algorithm to solve the log inexact match problem varying size of strings.

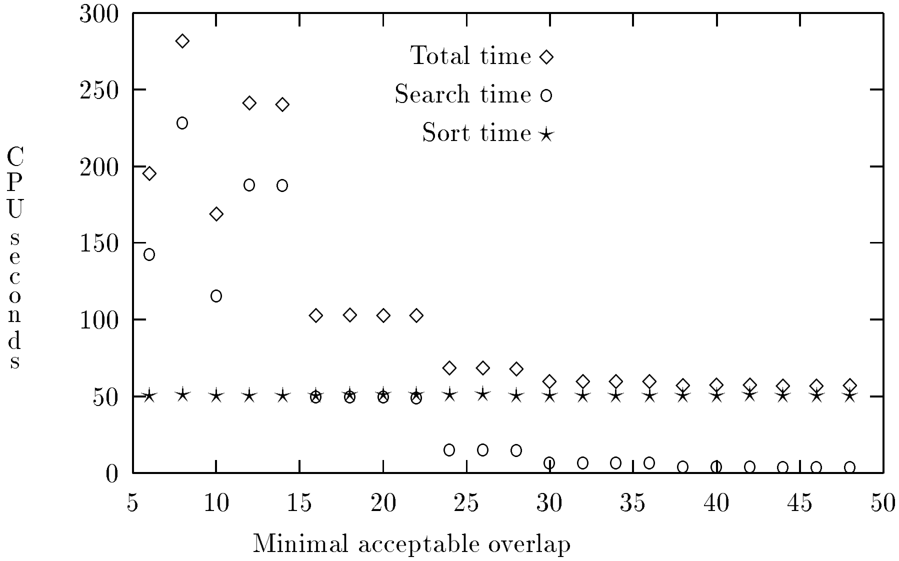


Figure 6.13: The running time in CPU seconds of the suffix array algorithm to solve log inexact match problem varying size of minimum overlap.

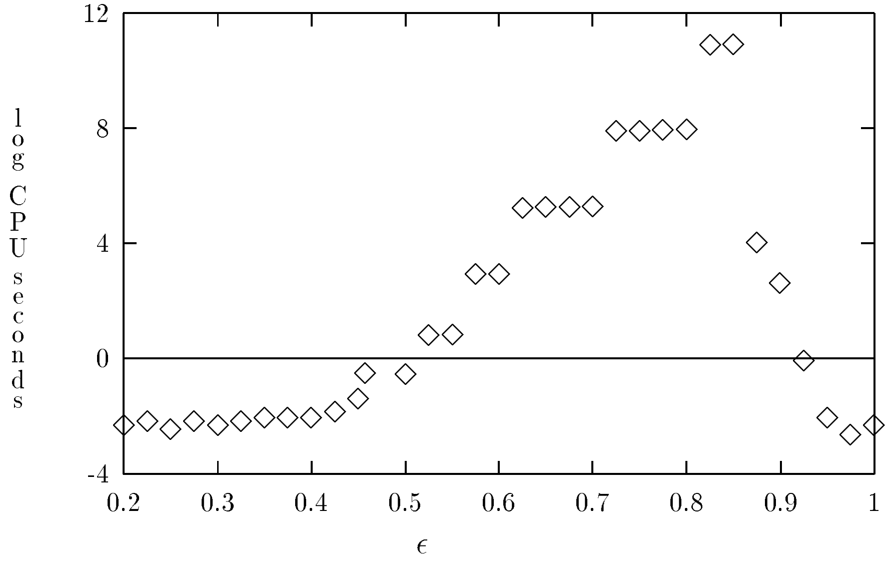


Figure 6.14: The log of running time of the suffix array algorithm allowing n^ϵ errors.

the minimum acceptable overlap is varied from 6 to 48. The number of strings is 200, the size of the strings is 200, and the actual overlap is 150.

The sets of strings are generated by initially generating a string that is the length of the strings in $\mathcal{S}$. Each successive string is generated by using a suffix of the previous string as the prefix of the current string. The length of the suffix that is used as a prefix of the current string is the length of the overlap between strings. When generating data for the algorithms to solve the log inexact shotgun sequencing problem the suffix from the previous string was modified before being used as the prefix of the current string. Once all of the strings had been generated, the order of the strings is randomized.

Although the suffix array based log inexact shotgun sequencing algorithm was initially designed to construct a consensus sequence from sequence fragments, it can be used to align similar sequences. As an example of this, we have used the algorithm to align the following three sequences from GenBank [38].

1. *Saccharomyces cerevisiae* TATA-box factor (TFIID) gene, $5'$ flank. This 1157 base pair DNA sequence was published by Schmidt et. al. [237] and

has GenBank accession number M26403.

2. *S. cerevisiae* TATA-binding protein (TFIID) gene, complete cds. This 2439 base pair DNA sequence was published by Hahn et. al. [130] and has GenBank accession number M27135.

3. *S. cerevisiae* transcription initiation factor IID (TFIID). This 1140 base pair DNA sequence was published by Horikoshi et. al. [136] and has GenBank accession number X16860.

The sequences M26403 and M27135 can be aligned with 6 differences, the sequences M26403 and X16860 can be aligned with 7 differences and the sequences M27135 and X16860 can be aligned with 1 difference. Running the suffix array based log inexact algorithm uses 2.62 CPU seconds to build the suffix array and 0.08 seconds to align the sequences with a minimum acceptable overlap (the value k) of 24. The naive algorithm uses over 37 CPU seconds with the minimum acceptable overlap set to 24. To help the naive algorithm we could remove the last 1000 base pairs of the sequence M27135 and set the minimum acceptable overlap to 900. With this help the naive algorithm still takes over 11 CPU seconds to compute the alignment.

6.4.5 Generalization

We have used $\log n$ errors in a match of length n simply because we used $\log n$ errors in the RK log inexact shotgun sequencing algorithm in section 6.3. Any function of n, $\mathrm{f}(n)$, such that $\mathrm{f}(n) \leq n$ could be used as the maximum number of errors allowed in a length n match. Following the same arguments that were used in sections 6.4.2 and 6.4.3 it can be shown that allowing $\mathrm{f}(n)$ errors in matches of length n, the algorithm in Figures 6.9 and 6.10 uses

$$O\left(\frac{N^3\mathrm{f}(k)}{k}\right)$$

worst case time and

$$O\left(\frac{nN\mathrm{f}(k)}{|\Sigma|^{\frac{k}{\mathrm{f}(k)}}}+N\right)$$

expected time.

We allowed n^ϵ, $0 < \epsilon < 1$, errors for matches of length n. Figure 6.14 shows the log of the running time of the algorithm in Figure 6.9 and 6.10 when allowing n^ϵ errors in length n matches while varying ϵ. The data shown in Figure 6.14 is for 10 sequences, each 410 bases long with overlaps of 180 bases and a minimum acceptable overlap of 40 bases.

Figure 6.14 shows a knee at about $\epsilon = 0.45$. As the number of errors allowed increases, the likelihood that some substring of S_{tot} will be falsely matched by the string

$$R = S_i[\mathrm{ptr}]\ldots S_i[\mathrm{ptr}+\mathrm{blocksize}-1]$$

(as used in Figure 6.10) increases. When the number of errors allowed is increased by one, the number of substrings in S_{tot} that falsely match R is expected to increase by a factor of σ. There is some value μ of ϵ where we expect there to be one substring in S_{tot} that falsely matches R. We expect the running time to be constant for $0 < \epsilon\mu$ since we expect one call, the correct call, to verify for each pair of strings that match. As ϵ grows above μ we expect false calls to verify. The number of false calls to verify will grow by a multiplicative factor of σ for each additional error allowed in a match. This is seen in the exponential growth in running time as ϵ increases from $\epsilon = 0.45$ to $\epsilon = 0.85$ in Figure 6.14. Eventually, as ϵ grows, there will be so many errors allowed that nearly any pair of strings will match and the number of calls to verify will fall.

In chapters 5 and 6 we have defined the consensus string problem and presented two new algorithms to solve it. We let

- n be the number strings in the multiset $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$
- $N = \sum_{i-1}^{n} |S_i|$
- $L =$ length of superstring found by heuristic

- C_H be the compression, $C_H = N - L$
- k be the minimum acceptable overlap length between strings

The first algorithm using ideas of Rabin–Karp exact string matching, is expected to solve the problem in $O(C_H \log n)$ time. The second algorithm developed to solve the consensus string problem is also expected to run in $O(C_H \log n)$ time, but in practice runs faster than the Rabin–Karp type algorithm by a factor of about ten.

We also define a similar problem, the log inexact consensus sequence problem, and present two new algorithms to solve this problem. The first algorithm used an extension of the ideas based on Rabin–Karp string matching developed for the consensus string problem. This algorithm, although not practical in many situations, is useful when the prefix/suffix overlap is known to be small. The algorithm is also easily parallelizable.

The second algorithm uses suffix arrays as developed by [189] and is expected to run in $O\left(\frac{nN\log k}{|\Sigma|^{\frac{k}{\log k}}} + N\right)$ time. This algorithm is generalized to allow not only $\log k$ errors but any reasonable function of k errors. Finally, we give an example of the use of this algorithm to align three nearly identical DNA sequences of the TFIID gene in yeast, although this was not the intended use of the algorithm.

These algorithms allow a few bases to be transformed but do not allow bases to be deleted. If, for example, a base is missed while reading a GC rich region (an occasional problem) the algorithms presented in this paper will not work since part of the match will be offset by one position. One area of future research will be designing algorithms for this problem that allows a small number of deletions as well as the transformations.

Chapter 7

Divide and Conquer Approximate String Matching

7.1 Introduction

Approximate string matching has been used in a biological context to find sequences that are similar in order to infer homology. Approximate string matching has also been used to search genetic sequence databases such as GenBank for sequences similar to the query sequence. There are a number of algorithms, many based on the dynamic programming algorithms, to solve the edit distance problem and the longest common subsequence problem, that are used to determine sequence similarity and search genetic sequence databases. The dynamic programming algorithms require the non–intersection property; if the characters A_i and B_j are aligned, A_k and B_l are aligned, and $i < k$, then $j < l$.

Gene rearrangements and inversions have been observed in many genes and genome fragments such as the histone gene cluster. To use a model of evolution that includes gene rearrangements and inversions when aligning or searching for sequences, an abstraction of the problem must be made that does not require the non–intersection property. We define the string to string rearrangement problem and corresponding algorithm that do not require the non–intersection property.

We give a divide and conquer algorithms to solve the string to string rearrangement problem. It is an improvement over previous dynamic programming algorithms that solve similar problems in several ways:

1. Since our algorithms do not require the non–intersection property, they allow

operators such as swap substrings and invert a substring.

2. Our algorithms allow any positive, monotonically increasing function of n to be used to penalize gaps of length n.

3. Our algorithms use $O(p)$ or $O(p \log p)$ work space where p is the length of the shorter string.

4. It is easy to parallelize our algorithms using a very simple model of parallel computation.

7.1.1 Previous work

The edit distance problem is: given two strings, find the fewest number of edit operations required to make the two strings identical. In chapter two, we noted that this problem has been solved by several researchers. The operations allowed are, delete a single character from either string and change a single character in one string to match the corresponding character in the other string. The dual of the edit distance problem is the longest common subsequence problem. $B = b_{x_1} b_{x_2} \cdots b_{x_m}$ is a subsequence of the string $A = a_1 a_2 \cdots a_n$ when $1 \leq x_1 < x_2 < x_3 \cdots < x_m \leq n$. The longest common subsequence problem is, given two strings, find the longest string that is a subsequence of both strings given.

In 1970, Needleman & Wunsch [209] gave an algorithm to find the longest common subsequence of two amino acid sequences. Although a detailed algorithm is never explicitly stated, it is clear that the ideas of the dynamic programming algorithms presented in later papers appear in this paper. The algorithm of Needleman & Wunsch was modified slightly by Sankoff [231] in 1972 and shown to take time and space proportional to the product of the length of the sequences being aligned. This paper introduced the equation

$$W(i,j) = \max\{W(i-1,j),\ W(i,j-1),\ W(i-1,j-1) + \delta(a_i, b_i)\} \qquad (7.1)$$

that is the basis for many of the dynamic programming algorithms. Sellers [240] generalized the algorithm of Sankoff to use a variety of distance functions in 1974.

Independently, in 1974, Wagner & Fischer [279] introduced the string to string correction problem and a dynamic programming algorithm to solve the problem. Although the terms are different, the algorithm is similar to the algorithm presented by Sankoff.

All of the algorithms to solve the longest common subsequence and edit distance problems previous to 1974 and many since 1974 have used $O(nm)$ time and space. These algorithms are frequently impractical because of their space usage, not their running time. In 1975 Hirschberg [132] introduced an algorithm to solve the longest common subsequence problem in $O(nm)$ time using only $O(n + m)$ space. Hirschberg's technique can be applied to many of the dynamic programming algorithms used to find the longest common subsequence or the shortest edit distance. This algorithm was frequently overlooked until 1988 when Myers & Miller [206] introduced Hirschberg's algorithm to the computational biology community.

When computing the edit distance between two strings, the insertion or deletion (indel) of n consecutive characters is treated as n distinct operations. It may be desirable to treat n consecutive indels as a single event when aligning sequences of nucleic acids or amino acids. In 1976 Waterman, Smith, & Beyer [289] introduced a metric that allowed the computation of the distance between two sequences when consecutive indels are treated as a single event. They gave an algorithm that can use any monotonically increasing function of the length of the indel, the gap penalty function, to penalize the indel. The algorithm runs in $O(n^2m)$ time where n and m are the lengths of the sequences, $n > m$. In 1982, Gotoh [119] developed an algorithm, based on Waterman's dynamic programming algorithm, that runs in $O(nm)$ time using an affine gap penalty function. In 1983, Fitch & Smith [91], and 1985, D. Feng, M. Johnson, & R. Doolittle [88], examined the effects of varying the parameters of the affine gap penalty function $g+lr$ where g is the penalty for the introduction of a gap, r is the penalty for each position of the gap, and l is the length of the gap.

Several recent algorithms allow more general gap penalty functions. In 1988 Miller & Myers [197], in 1989 Galil & Giancarlo [102], and in 1990 Eppstein, Galil, & Giancarlo [80] developed several algorithms to improve the speed of the dynamic programming algorithms allowing more general gap penalty functions. They are able to reduce the time required to align two sequences from $O(mn \cdot \mathsf{max}(m,n))$ to $O(mn \log mn)$ if the gap penalty function is convex or concave, and to $O(mn)$ if the gap penalty function has the closest zero property. In 1990, Gotoh [122] gave an algorithm that uses a piecewise affine gap penalty function to approximate a concave gap penalty function. The algorithm uses $O(Lmn)$ time where L is the number of linear pieces that are used to approximate the concave gap penalty function.

There are situations when we should consider rearrangements of the string where the order of the characters in the string is not maintained. Spelling mistakes are frequently caused by the transposition of adjacent characters. The model of genome shuffling as proposed by Sankoff and Goldstein [234] suggests that the operations should not be limited to those that maintain the order of the characters in the strings. The dynamic programming algorithms that we have discussed do not produce an alignment that alters the order of the characters in the strings. Lowrance & Wagner [184] added the operation of swapping two adjacent characters in one of the strings to the string to string correction problem of Wagner and Fischer. They show that with the addition of the swap operation the algorithm runs in $O(nm)$ time. In 1990, A. Bertossi, E. Lodi, F. Luccio, & L. Pagli [37] considered context dependent approximate string matching and added the operation of transposing two characters. In 1992, M. Schoniger & M. Waterman [238] gave a dynamic programming algorithm that finds local alignments with inversions although the algorithm does not find inverted segments within another inverted segment.

Tichy [267] gives an algorithm that converts one string to another using only block move operations. Given a source string and a target string the algorithm constructs a sequence of block moves from the source string to cover the target string using a minimum of move substring operations. Unlike the dynamic programming algorithms that solve the longest common subsequence problem and the edit distance problem, this algorithm allows substrings from any position in the source string to be used to cover any position in the target string. The algorithm runs in time and space that is proportional to the length of the input.

Tyler, Horton, and Krause [270] review several of the dynamic programming algorithms discussed here and many other interesting algorithms for database searching and aligning sequences.

The divide and conquer algorithm to find the longest common subsequence naturally leads to parallel and distributed implementations [32, 140, 146, 144]. By noticing that in equation 7.1 $W(i, j)$ is dependent only on the values of $W(i - 1, j)$, $W(i, j - 1)$, and $W(i - 1, j - 1)$ it is easy to compute, in parallel, an entire diagonal of the matrix W in one time step. Mukherjee [202] has exploited the natural parallelism to construct a pipelined hardware algorithm suitable for implementation in VLSI to find the longest common subsequence of two strings. Isenman [147] has used simulations to investigate hardware to search databases for strings that match query strings with variable length gaps.

7.1.2 Overview

In section 7.2 we will define the string to string rearrangement problem. Section 7.3 gives a basic divide an conquer algorithm to find an approximate solution to the string to string rearrangement problem. Several modifications to the algorithm are given that allow it to position gaps optimally, to construct the alignment, to find inversions, to execute efficiently on a parallel computer, and to use similarity matrices to measure the similarity between characters. We examine the resource consumption of the basic divide and conquer algorithm and its variants in section 7.4. In section 7.5 we will compare the alignments produced by our divide and conquer algorithm with the alignments produced by the dynamic programming algorithm for a number of nucleic acid sequences retrieved from GenBank. Finally, in section 7.6, we summarize and suggest possible areas of future research.

7.2 Problem definition

We have two strings, the pattern, P, and the text, T, and we want to find the pattern within the text. The string matching problem asks if P is a substring of T; the problem has been well studied [106]. In many cases we do not expect to find P. Instead, we expect to find a string P_T within T that is in some sense similar to P. Traditionally one counts local changes such as insertion or deletion of a character and replacement of one character with another to measure the similarity of P_T and P. In the edit distance problem we seek the substring P_T of T that can be transformed (edited) to P with the lowest cost sequence of operations. Since blocks of characters can be deleted by specifying the beginning and end of the block to delete, a gap penalty function is often introduced that penalizes consecutive inserts and deletes less severely than individual insertions and deletions.

We will allow the operations swap substrings and insert substring so that we will be able to find P broken into pieces within T. These pieces might be in a different order within T than they are in P. To capture these intuitions we define an alignment of P within T, and define a profit function which evaluates the value of a given alignment. An alignment is a mapping that assigns each character position in P to a character position in T. For example, the alignment $X = [x_0, x_1, \ldots, x_{p-1}]$ assigns the character at position j in P to align with the character at position x_j in T.

To build the profit function $\mathcal{F}$, we want a positive contribution when the character at position i in P is aligned with a similar character at position x_i in T. Let $\alpha(p_i, t_{x_1})$ be the profit associated with the similarity of the characters p_i and t_{x_i}. In the simplest case, we can use $\alpha(p_i, t_{x_i}) = 1$ when the characters are identical and $\alpha(p_i, t_{x_i}) = 0$ when the characters differ. In some applications we have a measurable definition of similarity so that $\alpha(p_i, t_{x_i})$ can have positive values for pairs of distinct characters. This similarity might measure, for example (in molecular biology), the profit of aligning a purine with another purine instead of a pyrimidine. When aligning amino acid sequences, the function α could be used to measure similarity in function or structure of different amino acids [70, 11].

For the profit function, we want a negative contribution when the string P is rearranged to match T. When substrings of P are rearranged, gaps are introduced into the alignment and the profit of the alignment should be penalized by a function of the sizes of the gaps that were introduced. Given the alignment X, the size of the gap is $|x_i - x_{i-1} - 1|$, e. g. the gap is zero when contiguous characters in P are mapped to contiguous characters in T. Since we are using absolute values to compute the size of the gap, we can use the same measure for a "gap" when $x_i < x_{i-1}$. Insertions into P are represented by having a gap between x_{i-1} and x_i where $x_i > x_{i-1} + 1$. Since there is no penalty for mismatched characters, a deletion in P is handled by inserting a gap between x_{i-1} and x_i where $x_i < x_{i-1} + 1$. We will use a monotonically increasing gap penalty function, $\mathcal{G}(l)$, such that $\mathcal{G}(0) = 0$. Our profit function, $\mathcal{F}$, has the form

$$\mathcal{F}(P, T, X) = \sum_{i=0}^{|P|-1} \left[\alpha(p_i, t_{x_i}) - \mathcal{G}(|x_{i-1} - x_i - 1|)\right].$$

We will refer to the value of the profit function for a particular alignent of P in T as the score of the alignment.

Finally, the string to string rearrangement problem is: given a similarity function α, a gap penalty function $\mathcal{G}$, a text string T, and a pattern string P, find the alignment X that maximizes the profit function $\mathcal{F}$.

7.3 Algorithm

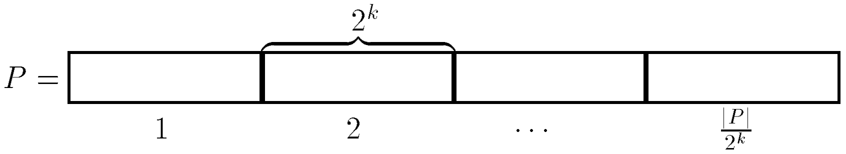
We will first give an algorithm to find the score of an alignment that approximates the optimal alignment with no inversions. We will modify this algorithm to produce an algorithm to compute the optimal placement of gaps, an algorithm that constructs the alignment, an algorithm that allows inversions in the alignment, and a parallel version of the algorithm. Each of the extensions to the basic algorithm may have a cost in terms of space or time that will be discussed in Section 7.4. The extensions to the basic algorithm may be used individually or they may be used in combination.

7.3.1 Alignments

Our algorithm computes the alignment of P at each position in T from $|P|-1$ to $|T|-1$. We will refer to these alignments as $X_{|P|-1}, X_{|P|}, \ldots, X_{|T|-1}$ where $X_i = [x_{i_1}, x_{i_2}, \ldots, x_{i_{|p|-1}}]$. Element j, $0 \leq j < |P|$, of X_i is the position in T that P_j is mapped to. Our algorithm iterates $\log |P|$ times to compute the alignment of P at position i in T. In the first iteration P_j, $0 \leq j < |P|$, is compared with T_{l+j} where $l = i - |P| + 1$. In successive iterations, gaps may be introduced into the alignment so that P_j no longer aligns with T_{l+j}. When the iteration completes we have computed X_i, the alignment of P in T at position i. Many of the substrings of P, $P_a \ldots P_b$, $0 \leq a, b < |P|$, $a \leq b$, may not align with $T_{i-|P|+a+1} \ldots T_{i-|P|+b+1}$ but with $T_{X_i[a]} \ldots T_{X_i[b]}$. In our terminology, in an alignment of $P_a \ldots P_b$ at position i, P_a aligns with $T_{X_i[a]}$, P_{a+1} aligns with $T_{X_i[a+1]}$, ..., P_b aligns with $T_{X_i[b]}$.

7.3.2 Basic algorithm

Our algorithm uses a binary tree that we refer to as the distance tree. The algorithm initially fills the leaves of this tree. It then successively fills in the higher level nodes so that eventually the root of the tree contains the information about the optimal alignment of P in T. The distance tree has $|P|$ leaves, one for each character in P. Following the divide and conquer form of our algorithm, the tree is a binary tree with $\log |P|$ levels. The $\frac{|P|}{2^k}$ nodes at level k of the tree represent P broken into $\frac{|P|}{2^k}$ length 2^k non–overlapping, adjacent substrings, e. g.



The initial values to be stored at the nodes of the distance tree are computed in the first stage of computation. These values indicate the best way to align P in the first $|P|$ positions of T. Subsequently, the distance tree will contain the information about the best alignment of P within the first i positions of T. At the $i + 1^{\text{st}}$ stage, the information in the distance tree from the i^{th} stage and information about position $i + 1$ of T are used to recompute the information in the distance tree. At the end of the $i + 1^{\text{st}}$ stage, the distance tree contains information about the best alignment of P within the first $i + 1$ positions of T.

Each node of the distance tree represents a substring of P. The j^{th} node of height k represents the substring

$$p_{j2^k}p_{j2^k+1}\cdots p_{(j+1)2^k-1}$$

of P and each leaf node (height 0) represents a single character of P. We will use V to represent the distance tree rooted at V_{root}. Each of the nodes of V represents a substring of P and can be used to root a distance tree for that substring of P. At stage i, the node associated with the substring $p_{j2^k}p_{j2^k+1}\cdots p_{(j+1)2^k-1}$ contains the values

- score – The maximum alignment score of the substring $p_{j2^k}p_{j2^k+1}\cdots p_{(j+1)2^k-1}$ minus the gap penalty $\mathcal{G}(i - l)$ when its last character is aligned at position l, $l \leq i$, of T.

- max – The maximum alignment score of the substring $p_{j2^k}p_{j2^k+1}\cdots p_{(j+1)2^k-1}$ when its last character is aligned at position l of T, $l \leq i$, such that max - $\mathcal{G}(i - l) \geq$ score.

- max_pos – l, the position which produces max.

```
search $(P, T)$
    $h \leftarrow |T| - |P| + 1$
    $i \leftarrow 0$
    max_score $\leftarrow -1$
    while $(i < h)$
        compare_elements$(P, T_i \ldots T_{i+|P|-1},$ leaves_of$(V), i)$
        filter_up$(V, i)$
        if $(V_{root}$.score $>$ max_score$)$
            max_score $\leftarrow V_{root}$.score
            max_pos $\leftarrow i$
        $i \leftarrow i + 1$
    return (max_score, max_pos)
```

Figure 7.1: A divide and conquer algorithm to find an approximate alignment of P in T.

The basic algorithm starts by initializing the leaves of the distance tree, then this information is filtered up until a score is computed for the root of the tree. With this initialized tree, the algorithm moves on to the next position, recomputes the information at the leaves by comparing the scores of the matches at the new position with the best scores found so far, again this information is filtered up the tree until the root contains the score for the best alignment of the string P found so far. The algorithm repeats these steps until all of the positions in T have been considered.

The procedure search in Figure 7.1 gives the overall structure of the algorithm as a loop that computes the distance tree and records the maximum score found. The procedure compare in Figure 7.2 computes the information stored in the leaves of the distance tree. The first time compare is called, it simply compares the elements of P with the corresponding elements of T and saves the score, initializes max as score, and saves the position as max_pos. On subsequent calls, if compare finds a match of the current character in P with the corresponding character in T, compare saves the information about this match. The procedure compare sets score to 0, and max_pos to the current position if the characters of P and T do not match and the best previous match was too far from the current

```
compare_elements($S1, S2, L, i$)
    — leaf$_j$ is the $j^{\text{th}}$ leaf of the tree $V$ —
    for each j, $0 \leq j < |S1|$
        if ($S1_j = S2_j$)
            leaf$_j$.score $\leftarrow c$
            leaf$_j$.max $\leftarrow c$
            leaf$_j$.max_pos $\leftarrow i$
        else
            leaf$_j$.score $\leftarrow \max(0, \text{leaf}_j.\text{max} - \mathcal{G}(i - \text{leaf}_j.\text{max\_pos}))$
            if (leaf$_j$.max $\leq \mathcal{G}(i -$ leaf$_j$.max_pos))
                leaf$_j$.max $\leftarrow 0$
                leaf$_j$.max_pos $\leftarrow i$
```

Figure 7.2: The characters of the pattern, $S1$, are compared to the corresponding characters of the text, $S2$, and the results stored in the leaves of the distance tree.

```
filter_up(node, $i$)
if node is not a leaf
    filter_up(node.left_child, $i$)
    filter_up(node.right_child, $i$)
    $L \leftarrow$ node.left_child
    $R \leftarrow$ node.right_child
    $Z_l \leftarrow \mathcal{G}(i - L.\text{max\_pos}) + \mathcal{G}(|L.\text{max\_pos} - R.\text{max\_pos}|)$
    $Z_r \leftarrow \mathcal{G}(i - R.\text{max\_pos}) + \mathcal{G}(|R.\text{max\_pos} - L.\text{max\_pos}|)$
    $Z_n \leftarrow \mathcal{G}(i - \text{node.max\_pos})$
    node.score $\leftarrow \mathsf{max}(L.\text{max} + R.\text{max} - \mathsf{min}(Z_l, Z_r), \text{node.max} - Z_n)$
    if node.score $>$ (node.max $- Z_n$)
        node.max $\leftarrow$ node.score
        node.max_pos $\leftarrow i$
```

Figure 7.3: An algorithm to compute the values of the internal nodes of the distance tree.

position as measured by the gap penalty function.

The recursive procedure filter_up in Figure 7.3 computes the best alignment of the substring of P represented by each node for the internal nodes of the distance tree. Specifically, it computes a new score and if this new score is larger than the previous best score minus a gap penalty, the new score and max_pos are remembered. This new score is either the best previous score minus a gap penalty, or the sum of the scores for the best alignments of the two half size substrings minus a gap penalty dependent on the relative positions of the half size substrings.

Z_l, Z_r, and Z_n in the procedure filter_up are gap penalties. Z_n is the penalty incurred if we keep the previous best alignment of the substring of P represented by the node. Z_l is the penalty if we introduce a gap in the middle of the substring of P and hold the left half of the substring in place and allow the right half to go to its maximum scoring position. Z_r is the penalty if we introduce the gap and hold the right half of the substring in place and allow the left half of the substring to go to its maximum scoring position. Note that we do not need to simultaneously let both the left and right halves of P go to their maximum scoring positions since this alignment would have been considered previously when one of the halves was held in place.

7.3.3 A simple example

Figure 7.4 shows a high level execution trace of the algorithm as the alignment of P = ABCD in T = ABXXCD is computed. The figure is divided into three sections, the first section shows the distance tree when ABCD is aligned with ABXX, position 1. Section two shows the distance tree when P is aligned at position 2, and section three shows the distance tree when P is aligned at position 3 of T. For this example we let the scoring function α be $\alpha(p_i, t_{x_i}) = 3$ if $p_i = t_{x_i}$ and 0 otherwise. A gap of length n will be penalized by $n + 1$,

$$\mathcal{G}(n) = \begin{cases} n+1 & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

A node in the distance tree is (score, max, max_pos).

In the first section of Figure 7.4, P is aligned at position 1 of T. The leaves of the distance tree have score $= 3$ if $t_i = p_i$ and 0 otherwise. Since this is the first position that P has been aligned in T, max_score = score and max_pos $= 1$. The internal nodes are simply the sum of their children since there are no previous maximum values.

In the second section of Figure 7.4, P is aligned at position 2 of T. At this position none of the characters of P match the corresponding characters of T. Consider the value of the leftmost leaf node, $(1, 3, 1)$. The score $= 1$ since, at the current position $p_1 \neq t_2$, but $p_1 = t_1$. If we introduce a gap of length one, p_1 will align with t_1. The score, if we introduce the gap of length 1, will be $3 - \mathcal{G}(1)$ or 1. Since 1 is greater than 0, the value of aligning p_1 with t_2, the gap is inserted. Notice that the score in the leftmost node on level 1, $(4, 6, 1)$ is not the sum of the scores of its children because the cost of introducing a single gap of length one for the substring AB is less than introducing two gaps, one for the substring A and another for the substring B. The result is that the substring AB of P is treated as a block and a single gap of length 1 is inserted.

In the third section of Figure 7.4, P is aligned at position 3 of T. The suffix CD of P matches the suffix of T. The root of the tree, $(9, 9, 3)$, tells us that the maximum score of 9 occurred at position 3 of T. The value 9 is the result of each character of P matching a character of T with the insertion of one gap of length two between the characters B and C of the pattern.

7.3.4 Optimal gap positions

In the previous section we used the algorithm to align the string “ABCD” in the string “ABXXCD” and the algorithm placed a gap between the ’B’ and ‘C’ in the string “ABCD”. If we use the algorithm to align the string “ABCDEF” in the string “ABCXXDEF” we will see that the gap is placed between ’B’ and ’C’, not between ’C’ and ‘D’ as we expect. This is a result of the divide and conquer

A		B		X		X
A		B		C		D
(3,3,1)		(3,3,1)		(0,0,1)		(0,0,1)
	(6,6,1)				(0,0,1)	
			(6,6,1)			

B		X		X		C
A		B		C		D
(1,3,1)		(1,3,1)		(0,0,2)		(0,0,2)
	(4,6,1)				(0,0,2)	
			(4,6,1)			

X		X		C		D
A		B		C		D
(0,0,3)		(0,0,3)		(3,3,3)		(3,3,3)
	(3,6,1)				(6,6,3)	
			(9,9,3)			

Figure 7.4: The distance tree at stages one, two, and three while aligning the pattern P = ABCD and the text T = ABXXCD. A node in the distance tree is ($\underline{\text{score}}$, $\underline{\text{max}}$, $\underline{\text{max\_pos}}$).

```
optimal_filter_up($W, i$)

if ($W \neq \Lambda$)
    optimal_filter_up($W$.left_child, $i$)
    optimal_filter_up($W$.right_child, $i$)
    $L \leftarrow W$.left_child
    $R \leftarrow W$.right_child
    $\delta_{GAP} \leftarrow$ MAX(max_gap($P_l, T_l, T_r, |W$.left_child$|, 1$),
                    max_gap($P_l + 1, T_l + 1, T_r + 1, |W$.left_child$|, -1$))
    $Z_l \leftarrow \mathcal{G}(i - L.\text{max\_pos}) + \mathcal{G}(|L.\text{max\_pos} - R.\text{max\_pos}|)$
    $Z_r \leftarrow \mathcal{G}(i - R.\text{max\_pos}) + \mathcal{G}(|R.\text{max\_pos} - L.\text{max\_pos}|)$
    $Z_n \leftarrow \mathcal{G}(i - W.\text{max\_pos})$
    $W.\text{score} \leftarrow \text{max}(L.\text{max} + R.\text{max} - \text{min}(Z_l, Z_r) + \delta_{GAP}, W.\text{max} - Z_n)$
    if $W.\text{score} > (W.\text{max} - Z_n)$
        $W.\text{max} \leftarrow W.\text{score}$
        $W.\text{max\_pos} \leftarrow i$
```

Figure 7.5: An algorithm that places gaps optimally while computing the internal nodes of the distance tree.

```
max_gap($P_l, T_l, T_r, D, v$)
    $\delta_s = \delta'_s = \max_s = 0$
    while $(D > 0)$
        $D \leftarrow D - 1$
        if $(P_l = T_l)$ $\delta'_s \leftarrow \delta'_s + 1$
        if $(P_l = T_r)$ $\delta_s \leftarrow \delta_s + 1$
        if $(\delta_s - \delta'_s > \max_s)$ $\max_s \leftarrow \delta_s - \delta'_s$
        $P_l \leftarrow P_l + v$; $T_l \leftarrow T_l + v$; $T_r \leftarrow T_r + v$
    return $\max_s$
```

Figure 7.6: The procedure max_gap finds the optimal position of a gap.

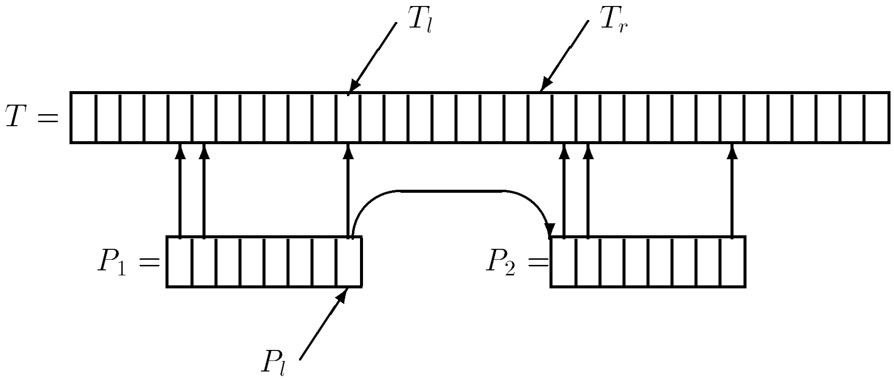


Figure 7.7: The procedure max_gap will move the gap, in this case to the left, and compute the change in alignment score as characters are moved form the tail of P_1 to the head of P_2.

nature of the basic algorithm presented in the previous section. In this section, we will present a modification to the basic algorithm that will allow us to optimally place the gaps in the alignment.

When a gap exists in an alignment, the position of the gap is placed between the substrings of P represented by two adjacent nodes of the distance tree by the basic algorithm. Let N_1 and N_2 be the nodes in V that represent the substrings $P_1 = P_{j2^k}P_{j2^k+1}\cdots P_{(j+1)2^k-1}$ and $P_2 = P_{(j+1)2^k}P_{(j+1)2^k+1}\cdots P_{(j+2)2^k-1}$. The basic algorithm places the gap between $P_{(j+1)2^k-1}$ and $P_{(j+1)2^k}$. This choice may be incorrect, but we will see in section 7.4 that this placement of the gap is a good approximation. To find the optimal placement of the gap we slide the gap back and forth by prepending suffixes of P_1 to P_2 and appending prefixes of P_2 to P_1. The procedure max_gap in Figure 7.6 returns the maximum increase in alignment score found by moving the gap in one direction. This value is computed by moving one character at a time over the gap. If the character being moved contributes a positive score in its current position, that score is subtracted from the overall score of moving the gap. If the character being moved has a positive contribution to the alignment score in its new position that contribution is added to the overall score of moving the gap.

The algorithm in Figure 7.5 redefines the procedure filter_up to compute the optimal placement of gaps in the alignment of P in T. Line 6 of the function optimal_filter_up searchs for the optimal placement of the gap found between the substrings P_1 and P_2. The following definitions of P_l, T_l, and T_r are also shown in Figure 7.7. Let $P_l = P_{(j+1)2^k-1}$, the right most character of the string P_1. Let T_r be the character in the text immediately preceding the character in the text that is aligned by the basic algorithm with $P_{(j+1)2^k}$. T_r is the right most text character of the gap. Let T_l be the character in the text that is aligned by the basic algorithm with P_l. Let D be the maximum distance that we can move the gap. In line 6 of optimal_filter_up, the function max_gap is first called to find the maximal score increase possible by moving the gap to the left. The function max_gap is called

```
construct_filter_up($W, i$)

1   if $W$ is not a leaf node
2       construct_filter_up($W$.left_child, $i$)
3       construct_filter_up($W$.right_child, $i$)
4       $L \leftarrow W$.left_child
5       $R \leftarrow W$.right_child
6       $Z_l \leftarrow \mathcal{G}(i - L.\mathrm{max\_pos}) + \mathcal{G}(|L.\mathrm{max\_pos} - R.\mathrm{max\_pos}|)$
7       $Z_r \leftarrow \mathcal{G}(i - R.\mathrm{max\_pos}) + \mathcal{G}(|R.\mathrm{max\_pos} - L.\mathrm{max\_pos}|)$
8       $Z_n \leftarrow \mathcal{G}(i - W.\mathrm{max\_pos})$
9       $W.\mathrm{score} \leftarrow \mathsf{max}(L.\mathrm{max} + R.\mathrm{max} - \mathsf{min}(Z_l, Z_r), W.\mathrm{max} - Z_n)$
10      if $W.\mathrm{score} > (W.\mathrm{max} - Z_n)$
11          $W.\mathrm{max} \leftarrow W.\mathrm{score}$
12          $W.\mathrm{max\_pos} \leftarrow i$
13          $W.X \leftarrow L.X \circ R.X$
```

Figure 7.8: A divide and conquer version of the algorithm that will construct the alignment.

again to find the maximal score increase possible by moving the gap the the right. The largest increase in score possible by moving the gap is stored in δ_{GAP}. In line 10 of optimal_filter_up, the increase in score that is possible by moving the gap is added to the value used to score the cost of inserting a gap between L and R.

7.3.5 Constructing the alignment

The basic algorithm reports the score and position of the approximate alignment, but does not give enough information to construct the alignment. Using a factor of $\log |P|$ more space we can construct the approximate alignment. The field $\underline{\mathrm{X}}$ is added to each node of the distance tree. For node N of height k, the array $\underline{\mathrm{X}}$ contains the alignment of the string P' represented by N. Element i, $0 < i < 2^k$, of $\underline{\mathrm{X}}$ contains the position in the text that the i^{th} character of the string P' should be aligned with.

The function construct_filter_up, shown in Figure 7.8, was created by adding line 13 to the function filter_up in Figure 7.3. If the score of the alignment of the

```
invert_search ($P, T$)
    $l \leftarrow |T| - |P| + 1$
    $i \leftarrow 0$
    max_score $\leftarrow -1$
    $I \leftarrow$ invert($P$)
    while ($i < l$)
        compare_elements($P, T_i \ldots T_{i+|P|-1}$, leaves_of($V$), $i$)
        compare_elements($I, T_i \ldots T_{i+|P|-1}$, leaves_of($IV$), $i$)
        invert_filter_up($V, IV, i$)
        if ($V_{root}$.score > max_score)
            max_score $\leftarrow V_{root}$.score
            max_pos $\leftarrow i$
        $i \leftarrow i + 1$
    return (max_score, max_pos)
```

Figure 7.9: A Divide and Conquer algorithm to find an approximate alignment of P in T allowing inversions in the alignment.

string P' at position i of T is the maximum in positions 0 to i, then we save the score and the current alignment. Note that we must save the entire alignment and not just pointers to the alignments of the children since the maximum scoring alignments of the children may change without the maximum scoring alignment of P' changing. When the function construct_filter_up completes, the alignment that produces the score $\underline{\mathrm{max}}$ in the root node of the alignment tree will be stored in the $\underline{\mathrm{X}}$ field of the root node. We must also modify the function compare_elements in Figure 7.2 to set V_j.X[0] to i whenever V_j.max_pos is set to i.

7.3.6 Inversions

The basic algorithm can be modified to find alignments when parts of the pattern are inverted. The idea is to invert the pattern and compute a distance tree of the inverted pattern in parallel with the distance tree of the original pattern. Let W be the node in the distance tree representing the substring $P' = P_{j2^k}P_{j2^k+1}\cdots P_{(j+1)2^k-1}$. Let IW be the node in the inverted distance tree representing the substring $\overline{P'} = \overline{P_{(j+1)2^k-1}P_{(j+1)2^k-2}}\cdots\overline{P_{j2^k}}$ where $\overline{P_i}$ is the inverse of P_i. We use IZ as the penalty associated with inverting P'. IZ could be

```
invert_filter_up($W, IW, i$)

  if $W$ is not a leaf node
      invert_filter_up($W$.left_child, $IW$.left_child, $i$)
      invert_filter_up($W$.right_child, $IW$.right_child, $i$)
— Compute score for substring of $P$ —
      $L \leftarrow W$.left_child
      $R \leftarrow W$.right_child
      $Z_l \leftarrow \mathcal{G}(i - L.\text{max\_pos}) + \mathcal{G}(|L.\text{max\_pos} - R.\text{max\_pos}|)$
      $Z_r \leftarrow \mathcal{G}(i - R.\text{max\_pos}) + \mathcal{G}(|R.\text{max\_pos} - L.\text{max\_pos}|)$
      $Z_n \leftarrow \mathcal{G}(i - W.\text{max\_pos})$
      $W.\text{score} \leftarrow \mathsf{max}(L.\text{max} + R.\text{max} - \mathsf{min}(Z_l, Z_r), W.\text{max} - Z_n)$
      if $W.\text{score} > (W.\text{max} - Z_n)$
          $W.\text{max} \leftarrow W.\text{score}$
          $W.\text{max\_pos} \leftarrow i$
— Compute score for the inverse of the substring of $P$ —
      $L \leftarrow IW$.left_child
      $R \leftarrow IW$.right_child
      $Z_l \leftarrow \mathcal{G}(i - L.\text{max\_pos}) + \mathcal{G}(|L.\text{max\_pos} - R.\text{max\_pos}|)$
      $Z_r \leftarrow \mathcal{G}(i - R.\text{max\_pos}) + \mathcal{G}(|R.\text{max\_pos} - L.\text{max\_pos}|)$
      $Z_n \leftarrow \mathcal{G}(i - IW.\text{max\_pos})$
      $IW.\text{score} \leftarrow \mathsf{max}(L.\text{max} + R.\text{max} - \mathsf{min}(Z_l, Z_r), IW.\text{max} - Z_n)$
      if $IW.\text{score} > (IW.\text{max} - Z_n)$
          $IW.\text{max} \leftarrow IW.\text{score}$
          $IW.\text{max\_pos} \leftarrow i$
— Swap inverted scores if they are good enough —
      if $W.\text{score} < (IW.\text{score} - IZ)$
          $W.\text{max\_score} \leftarrow IW.\text{score} - IZ$
          $W.\text{score} \leftarrow W.\text{max\_score}$
      if $IW.\text{score} < (W.\text{score} - IZ)$
          $IW.\text{max\_score} \leftarrow W.\text{score} - IZ$
          $IW.\text{score} \leftarrow IW.\text{max\_score}$
```

Figure 7.10: An algorithm to compute the internal nodes of the distance tree when inversions are allowed in the alignment.

a constant or the result of some function similar to a gap penalty function. If the alignment score of P' is less than the alignment score of $\overline{P'} - IZ$ then the inverted substring $\overline{P'}$ is used in the alignment instead of the substring P'.

The algorithm, modified to find inverted segments, is given in Figures 7.9 and 7.10. The function invert_search uses the variable I to store inverse string of P. The root of the distance tree for the inverted pattern is stored in IV. Since the function invert_filter_up handles the incorporation of inverted segments into the distance tree, the alignment score, including possible inversions, is stored in V.score.

The function invert_filter_up in Figure 7.10 differs from the function filter_up in Figure 7.3 in two ways. First, a score is computed for both nodes W and IW. Second, in lines 22 – 27 of the function invert_filter_up, the larger of the scores W.score and IW.score, if it is more than IZ larger, is stored in both W.score and IW.score. An inversion is included in the alignment score if and only if it increases the alignment score. Note that inversions are incorporated into both the distance tree W and the inverted distance tree IW so that it is possible to have "locally inverted" substrings within a larger inverted substring.

7.3.7 Parallel algorithm

The parallel algorithm presented in Figures 7.11, 7.12, 7.13 and 7.14 will compute the approximate alignment of P in T using a data structure and algorithm similar to the basic algorithm. We use the EREW[1] PRAM[2] model of parallel computation although we do not need to assume the shared memory of the EREW PRAM model. We will show that local memories and limited communication are sufficient for our algorithm to function correctly.

We have added the two fields updated and pos to the data structure that was used in the basic algorithm. The parallel algorithm uses two copies of the

[1] Exclusive Read Exclusive Write

[2] Parallel Random Access Machine

```
search($P$, $T$)

    $l \leftarrow |T| - |P| + \log(|P|) + 1$
    $a \leftarrow 0$
    $i \leftarrow 0$
    max_score $\leftarrow -1$
    while ($i < l$)
        if ($a = 1$) $a \leftarrow 0$ else $a \leftarrow 1$
        compare_elements($P, T_i \ldots T_{i+|P|-1}$, leaves_of($V_a$), $i$)
        filter_up($V_a, V_{\overline{a}}$)
        update($V_0, V_1$)
        if ($V_{a,\text{ root}}$.score > max_score)
            max_score $\leftarrow V_{a,\text{ root}}$.score
            max_pos $\leftarrow V_{a,\text{ root}}$.max_pos
        $i \leftarrow i + 1$
    return (max_score, max_pos)
```

Figure 7.11: A Parallel divide and conquer algorithm to find an alignment of P in T.

```
compare_elements($S1, S2, W, i$)

    for each $0 \leq j < |S2|$ do in parallel
        if $S1_j = S2_j$
            $W_j$.score $\leftarrow c$
            $W_j$.max $\leftarrow c$
            $W_j$.max_pos $\leftarrow i$
            $W_j$.updated $\leftarrow$ true
        else
            $W_j$.score $\leftarrow 0$
            if $W_j$.max $\leq \mathcal{P}(i - W_j$.max_pos)
                $W_j$.max $\leftarrow 0$
                $W_j$.max_pos $\leftarrow i$
                $W_j$.updated $\leftarrow$ true
            else
                $W_j$.updated $\leftarrow$ false
        $W_j$.pos $\leftarrow i$
```

Figure 7.12: An algorithm to compute the leaves of the distance tree in parallel.

```
filter_up($V1, V2$)

for each non–leaf node $j$, do in parallel
    $N_j \leftarrow V1_j$
    $L_j \leftarrow V2_j$.left_child
    $R_j \leftarrow V2_j$.right_child
    $N_j$.pos $\leftarrow L_j$.pos
    $Z_{lj} \leftarrow \mathcal{P}(N_j.\text{pos} - L_j.\text{max\_pos}) + \mathcal{P}(L_j.\text{max\_pos} - R_j.\text{max\_pos})$
    $Z_{rj} \leftarrow \mathcal{P}(N_j.\text{pos} - R_j.\text{max\_pos}) + \mathcal{P}(R_j.\text{max\_pos} - L_j.\text{max\_pos})$
    $Z_{nj} \leftarrow \mathcal{P}(N_j.\text{pos} - N_j.\text{max\_pos})$
    $N_j.\text{score} \leftarrow \mathsf{max}(L_j.\text{max} + R_j.\text{max} - \mathsf{min}(Z_{lj}, Z_{rj}), N_j.\text{max} - Z_{nj})$
     if $N_j.\text{score} > (N_j.\text{max} - Z_{nj})$
         $N_j.\text{max} \leftarrow N_j.\text{score}$
         $N_j.\text{max\_pos} \leftarrow N_j.\text{pos}$
         $N_j$.updated $\leftarrow$ true
     else
         $N_j$.updated $\leftarrow$ false
```

Figure 7.13: An algorithm to compute the internal nodes of the distance tree in parallel.

```
update($V1, V2$)

  for each $0 \leq j < |V2|$ do in parallel
      if $V1_j$.updated
          $V1_j$.updated $\leftarrow$ false
          $V2_j \leftarrow V1_j$
      if $V2_j$.updated
          $V2_j$.updated $\leftarrow$ false
          $V1_j \leftarrow V2_j$
```

Figure 7.14: An algorithm to reconcile the internal nodes of the two distance trees in parallel.

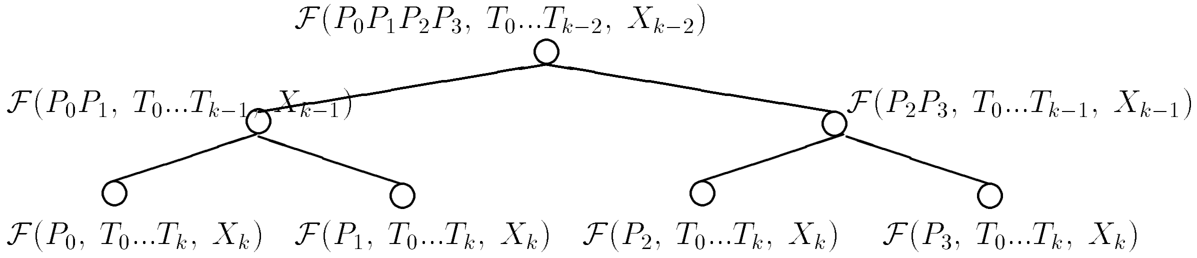

$\mathcal{F}(P_0P_1P_2P_3,\ T_0...T_{k-2},\ X_{k-2})$
$\mathcal{F}(P_0P_1,\ T_0...T_{k-1},\ X_{k-1})$
$\mathcal{F}(P_2P_3,\ T_0...T_{k-1},\ X_{k-1})$
$\mathcal{F}(P_0,\ T_0...T_k,\ X_k)$
$\mathcal{F}(P_1,\ T_0...T_k,\ X_k)$
$\mathcal{F}(P_2,\ T_0...T_k,\ X_k)$
$\mathcal{F}(P_3,\ T_0...T_k,\ X_k)$

Figure 7.15: Tree used by parallel algorithm

distance tree to eliminate read write conflicts. To allow us to keep both copies of the tree current, every time a change is made to a node of one tree, it is marked by setting that node's updated field to true. In the basic algorithm, all nodes in the tree represent substrings of P being aligned at the position in T. In the parallel version of the algorithm, each level of the tree represents an alignment of P at a different position in T. The field pos, associated with each node of the tree, stores the position in T that P was aligned.

The parallel algorithm differs from the basic algorithm in the method used to compute the values of the internal nodes of the tree. In the basic algorithm, all of the internal nodes of the tree are computed immediately following the computation of the leaves. This will not work in the parallel algorithm since the value of an internal node is dependent on the values of all of the node's descendants. In the parallel algorithm, each level of the tree holds the score for a different position of the alignment of P in T. For example, if the leaves of the tree hold the values of the comparison of P with $t_{i-p+1} \ldots t_i$, nodes at height h hold the values computed for the comparison of P with $t_{i-p+1-h} \ldots t_{i-h}$ (see Figure 7.15). Since each level of the tree holds the values for different positions of P in T, the values stored in each node of the tree are dependent only on that nodes children, not on all of its decendents. To avoid simultaneous reads and writes we maintain two copies of the tree, V_0 and V_1, reading from one tree and writing to the other. This allows us to compute the value of each node of the distance tree in parallel with no read or write conflicts.

The procedure search in Figure 7.11 takes a pattern, P, and a text, T, and will compute the alignment score for each of the positions the pattern P can be placed in text T. It will then return the score and position of the approximate best alignment of P in T. The while loop is iterated $|T| - |P| + \log(|P|) + 1$ times, the extra $\log(|P|)$ iterations are needed to insure that the results of the final comparison, P with $t_{|T|-|P|} \cdots t_{|T|-1}$ travel to the root of the distance tree $(V_{a,\, root})$. At the end of each iteration of the while loop the procedure update modifies the trees V_0 and V_1 to make them identicle.

The procedure compare_elements in Figure 7.12 does a character by character comparison of the characters in P and $t_i \cdots t_{i+|P|-1}$ and stores the results in the leaves of the distance tree. If the characters match, the value c is stored as the current and maximum values, the position is saved and this leaf is marked as having been modified. If the characters do not match, the score is set to zero, and the maximum value is checked to see if it is still valid. If the maximum value is less than the gap penalty then the maximum value of the leaf is set to zero, the position of the maximum value is set to the current position, and the leaf is marked as having been modified. Finally, the alignment position of the character represented by each leaf node is set to the current position Note that the values of each leaf node are computed in parallel.

The combination of the leaf values into a score for the alignment is done by the procedure filter_down in Figure 7.13. The variable N_j represents the parent node, L_j and R_j represent the left and right children of N_j. Notice that N_j is read from the $V1$ copy of the tree and the children are read from the $V2$ copy of the tree. Although all of the internal nodes of the tree are being computed in parallel, no node is being written to and read from simultaneously since all of the nodes that are being written to are from the $V1$ copy of the tree and all of the nodes that are being read from are from the $V2$ copy of the tree and each node is written to or read from by one processor. Since each level of the tree contains the partial results of an alignment at different positions in T, the position of the comparison that is being computed is recorded in N.pos. The remainder of the procedure is similar to the procedure given in Figure 7.3.

We maintain two copies of the distance tree so that no two processors ever need to access the same node simultaneously. When a change is made to one of the trees, the other tree needs to be updated to reflect the change before the next iteration of the while loop in the procedure search can be executed. Each time a change is made to a node in the tree, the node is marked as having been updated.

The procedure `update`, shown in Figure 7.14, will examine the corresponding nodes of each tree and modify them, if necessary, to make them identical. Only one of the trees could have been written to, and if node j from that tree was updated, the node j in the other tree is modified to reflect the update. All of the nodes in the trees are updated in parallel. When the procedure `update` completes, the distance trees are identical.

7.4 Resource use

We have given a basic algorithm to find an approximate solution to the string to string rearrangement problem and three extensions to that algorithm. The basic algorithm runs in $O(tp)$ time using $O(p)$ work space. This algorithm can be modified to find alignments with inversions that also runs in $O(tp)$ time and $O(p)$ work space. The basic algorithm can be modified to place the gaps optimally using $O(tp \log p)$ time and $O(p)$ work space. We have also modified the basic algorithm to construct the alignment. This algorithm takes $O(tp \log p)$ time and $O(p \log p)$ work space.

7.4.1 Basic algorithm

In this section we will prove a number of theorems about the resource usage and performance of the basic algorithm.

- The algorithm executes in time proportional to the product of the length of the text and the length of the pattern.

- The work space used by the algorithm is proportional to the length of the pattern. Work space is independent of the length of the text.

- The algorithm underestimates the actual alignment score of a pattern at a given position in the text by, at most, the sum of the gap penalties in the alignment at that position.

Theorem 7.1 *The algorithm presented in Figures 7.1, 7.2, and 7.3, when given T, a text string of length t, and P, a pattern string of length p, finds the score and position of the approximate alignment of P in T using $O(tp)$ operations.*

Proof. The time used by each call to the function compare_elements is $O(p)$ since the for loop is executed once for each character in P and each line in the for loop can be executed in constant time. There are $p - 1$ internal nodes in the distance tree, V, passed to the function filter_up. For each internal node, W, we compute the score and update W.max and W.max_pos. Each of the lines 6 through 14 in the function filter_up can be computed in constant time. The time used by each call to filter_up is $O(p)$. Since the while loop in the function search is iterated $O(t)$ times and the functions compare_elements and filter_up each take $O(p)$ time, the total time used by the algorithm is $O(tp)$. ▮

Theorem 7.2 *The algorithm presented in Figures 7.1, 7.2, and 7.3, when given T, a text string of length t, and P, a pattern string of length p, will find the score and position of the approximate alignment of P in T using $O(p)$ words of working memory.*

Proof. The only data in the function search are P, T, and V. P, by definition, is $O(p)$. Since V is a binary tree, each node of the tree has constant size, and the tree has $2p - 1$ nodes, the size of V will be $O(p)$. During iteration i of the while loop in the function search we only need the length p substring $T_{i-p+1} \cdots T_i$ of T. The function filter_up is recursive and will be called once for each node in the tree V. Each stack frame has constant size so the stack will use $O(p)$ space. ▮

We will use the following two lemmas to show that the score of the approximate alignment of P in T that is found by the basic algorithm underestimates the score of the optimal alignment by at most the sum of the gap penalties in the approximate alignment of P in T. We first show that if the optimal alignment contains no gaps, our algorithm returns the score of the optimal alignment. We then show that if a gap is introduced into an alignment, our algorithm reduces the alignment score by at most two times the penalty for the gap.

Lemma 7.1 *If the optimal alignment of P in T contains no gaps, the algorithm given in Figures 7.1, 7.2, and 7.3 will correctly return i, the position of the optimal alignment and the optimal alignment score.*

Proof. We will use induction on the levels of the distance tree. The leaves, level 0, are the base case. If the optimal score is at position i, then when compare_elements is called at position i, the sum of the scores at the leaves of the tree must be equal to the optimal score. This is because, if the optimal alignment contains no gaps, then all scoring must come from the leaves. If no gaps are introduced, the function filter_up simply finds the sum of the leaf node scores.

The only way the line

$$\text{if } (L_j.\text{max} \leq \mathcal{G}(i - L_j.\text{max\_pos}))$$

of compare_elements could be false is if we introduce a gap of length $(i - L_j.\text{max\_pos})$. This gap would be introduced only if it increased the score, but this can't be since we know the optimal score contains no gaps. For every j that $S1_j \neq S2_j$, the if statement

$$\text{if } (L_j.\text{max} \leq \mathcal{G}(i - L_j.\text{max\_pos}))$$

of compare_elements is true, L_j.max is set to 0, and L_j.max_pos will be set to i. Therefore, for each j, $L_j.\text{score} = L_j.\text{max}$ and $L_j.\text{max\_pos} = i$, the position of the optimal alignment score.

If each node at level $k-1$ of the distance tree has been computed correctly, then the algorithm computes the nodes at level k correctly. For each node at level k we compute W.score in the function filter_up. We know the following

- Neither the alignment represented by the node L nor the alignment represented by the node R contain gaps.

- L.max_pos $= i$ and R.max_pos $= i$ so $Z_l = 0$ and $Z_r = 0$.

- The sum of the scores on level k must equal the optimal score since if the optimal alignment contains no gaps, all scoring must ultimately come from the alignments represented by the nodes on level k. If no gaps are introduced, the function filter_up simply finds the sum of the nodes at level k. It must be true that $L.\text{max} + R.\text{max} \geq W.\text{max} - Z_n$ since, if this were not true, the score at position W.max_pos would be larger than the score at position i, but we assumed the maximum score occurred at position i.

With the above observations, we can see that the algorithm does not introduce any gaps on level k of the tree. Since $W.\text{score} > (W.\text{max} - Z_n)$, the maximum score, W.max, is set to the score at position i and the position of the maximum score, W.max_pos, is set to i. Also note that when the function filter_up completes, the score stored in the root of the tree will be the score of the alignment of P in T at position i. The function search compares the score stored in the root with the previous largest score and then correctly records the score at position i as the maximum. ∎

Lemma 7.2 *Given a pattern P, a text T, an alignment X_i, and a score, s, for the alignment of P in T at position i; adding a block of characters B, of length b, to T (potentially adding a length b gap to the alignment) causes the algorithm in Figures 7.1, 7.2, and 7.3 to reduce the score for the alignment by at most $2\mathcal{G}(b)$.*

Proof. Assume that B was inserted into the substring of T that P aligns with in the alignment X_i, otherwise the alignment score would not be reduced at all. Let $T' = T_0 \cdots T_j B_0 \cdots B_{b-1} T_{j+1} \cdots T_{t-1}$. When the gap is introduced following position j in T we must consider the substring of P, $P_{l \cdot 2^k + 1} \cdots P_{(l+1)2^k}$, that contains P_h where $X_i[h] = j$. If B was inserted into a gap in the alignment X_i then we will increase the length of the gap by b and let P_h be the last character in P before the gap that was aligned with some character in T. Let X_i' be the alignment produced by our algorithm after the insertion of B. We will consider two cases, first when $X_i'[h] = j$ and second, when $X_i'[h] = j + b$. In either case, if the gap were placed optimally, it would be placed following P_h, but our algorithm will place the gap between the characters $P_{l\cdot 2^k + 2^{k-1}}$ and $P_{l\cdot 2^k + 2^{k-1}+1}$.

Case 1: $X_i'[h] = j$ (See Figure 7.16). In this case $h \leq l \cdot 2^k + 2^{k-1}$ and the substring $P_h \cdots P_{l\cdot 2^k + 2^{k-1}}$ is misaligned.

$$\begin{aligned} \text{Let } e &= l \cdot 2^k + 2^{k-2} \\ f &= l \cdot 2^k + 2^{k-1} \end{aligned}$$

$$\begin{aligned} \text{Let } A_1 &= \mathcal{F}(P_{h+1} \cdots P_f,\ T_{X_i[h+1]} \cdots T_{X_i[f]},\ X_i) \\ A_2 &= \mathcal{F}(P_e \cdots P_h,\ T_{X_i[e]} \cdots T_{X_i[h]},\ X_i) \\ A_3 &= \mathcal{F}(P_{l\cdot 2^k+1} \cdots P_h,\ T_{X_i[l\cdot 2^k+1]} \cdots T_{X_i[h]},\ X_i) \end{aligned}$$

First we note that

$$A_1 + A_3 - A_2 - \mathcal{G}(b) < 0$$

otherwise the gap would have been inserted between P_e and P_{e+1}. Rearranging we get

$$A_1 + A_3 < \mathcal{G}(b) + A_2. \tag{7.2}$$

We know that

$$A_3 > \mathcal{G}(b) + A_1 > A_2$$

otherwise the gap would not have been placed following $P_{l\cdot 2^k + 2^{k-1}}$. Since $A_3 > A_2$ we know that $A_1 < \mathcal{G}(b)$. Since A_1 is the only scoring section in X_i that is not in X_i' and the penalty for the gap is $\mathcal{G}(b)$, the score for the alignment X_i' is at most $2\mathcal{G}(b)$ less than the score for the alignment X_i.

Case 2: $X_i'[h] = j + b$ (See Figure 7.17). In this case $X_i[j] \geq l \cdot 2^k + 2^{k-1}$ and the substring $P_{l\cdot 2^k + 2^k + 1} \cdots P_{X_i'[h]}$ is misaligned.

$$\begin{aligned} \text{Let } e &= (l+1) \cdot 2^k - 2^{k-2} \\ f &= l \cdot 2^k + 2^{k-1} \end{aligned}$$

$$\begin{aligned}\text{Let } A_1 &= \mathcal{F}(P_f \cdots P_h,\ T_{X_i[f]} \cdots T_{X_i[h]},\ X_i) \\ A_2 &= \mathcal{F}(P_h \cdots P_e,\ T_{X_i[h]} \cdots T_{X_i[e]},\ X_i) \\ A_3 &= \mathcal{F}(P_h \cdots P_{(l+1)2^k},\ T_{X_i[h]} \cdots T_{X_i[(l+1)2^k]},\ X_i)\end{aligned}$$

First we note that

$$A_1 + (A_3 - A_2) - \mathcal{G}(b) < 0$$

otherwise the gap would have been inserted between P_e and P_{e+1}. rearranging we get

$$A_1 + A_3 < \mathcal{G}(b) + A_2. \tag{7.3}$$

We know that

$$A_3 > \mathcal{G}(b) + A_1 > A_2$$

otherwise the gap would not have been placed following $P_{l \cdot 2^k + 2^{k-1}}$. Since A_1 is the only scoring section in X_i that is not in X_i' and the penalty for the gap is $\mathcal{G}(b)$, the score for the alignment X_i' is at most $2\mathcal{G}(b)$ less than the score for the alignment X_i. ∎

Theorem 7.3 *Given T, P, and an alignment X_i for position i computed by the basic algorithm, the approximate alignment score computed by the algorithm will underestimate the actual alignment score by at most*

$$\sum_{j=1}^{i} \mathcal{G}(|X_i[j-1] - X_i[j] + 1|)$$

if $\mathcal{G}(n)$ is a convex function.

Proof. We will show this by using induction on k, the number of gaps in the alignment X_i. For the base case, $k = 0$, we note that if the optimal alignment contains no gaps, lemma 7.1 shows that our algorithm finds the optimal alignment.

Assume that for all optimal alignments with $k-1$ gaps and a total gap penalty of g, our algorithm computes an alignment with a total gap penalty of at most $2g$. Lemma 7.2 showed that we can add a gap of length n to an alignment and

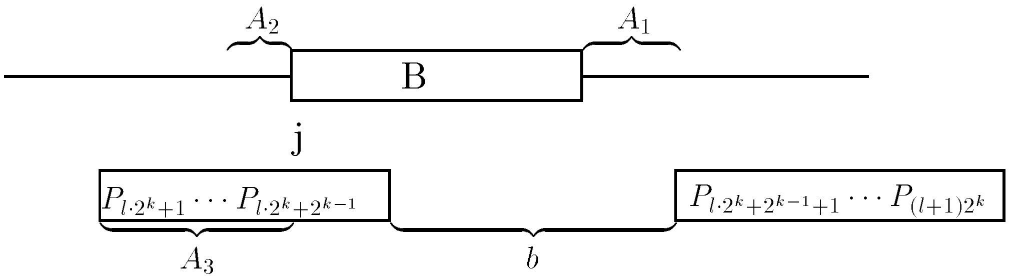

A_2
A_1
B
j
$P_{l\cdot 2^k+1}\cdots P_{l\cdot 2^k+2^{k-1}}$
$P_{l\cdot 2^k+2^{k-1}+1}\cdots P_{(l+1)2^k}$
A_3
b

Figure 7.16: A diagram of a gap insertion for case 1 of Lemma 3.

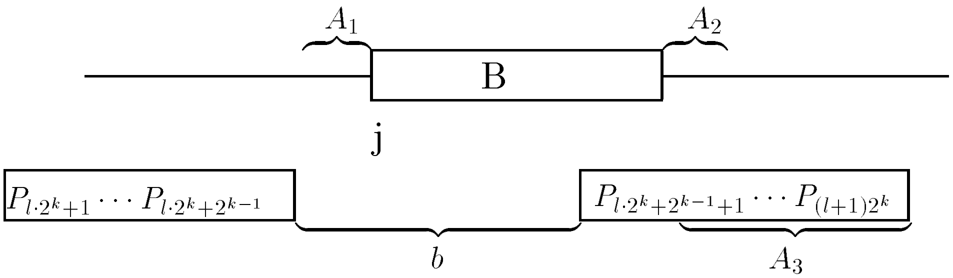

A_1
A_2
B
j
$P_{l\cdot 2^k+1}\cdots P_{l\cdot 2^k+2^{k-1}}$
$P_{l\cdot 2^k+2^{k-1}+1}\cdots P_{(l+1)2^k}$
b
A_3

Figure 7.17: A diagram of a gap insertion for case 2 of Lemma 3.

reduce the score by no more than $2\mathcal{G}(n)$, an underestimate of, at most, $\mathcal{G}(n)$. So, for an alignment with k gaps and an optimal alignment score with gap penalties of $g+\mathcal{G}(n)$, our algorithm finds an alignment with gap penalties of no more than $2g+2\mathcal{G}(n)$. ∎

This bound on the error allows us to report not only an approximate alignment and its score, but also a bound on the error of the score if the gap penalty function is convex.

7.4.2 Optimal gap placement

We saw in the last section that our basic algorithm places gaps close to their proper positions. By searching in the vicinity of the position that the basic algorithm places gaps we can find the position of the optimal placement of gaps. The cost of finding the optimal placement of gaps is a factor of $\log p$ increase in running time.

Theorem 7.4 *The algorithm to compute the position of the alignment with optimal gap positioning runs in $O(tp \log p)$ time when given T, a text string of length t and P, a pattern string of length p.*

Proof. The basic algorithm is modified by using the function optimal_filter_up in place of the function filter_up. The function optimal_filter_up was created by adding line 8 to the function filter_up. Line 8 makes two calls to the function max_gap in Figure 7.6. The function max_gap will use time proportional to the parameter D. Consider the nodes of the distance tree V of height k. These nodes represent the substrings $P_0P_1 \cdots P_{2^k-1}$, $P_{2^k}P_{2^k+1} \cdots P_{2^{k+1}-1}$, $\ldots$, $P_{p-2^k}P_{p-2^k+1} \cdots P_{p-1}$. The sum of the length of the substrings represented by the nodes of height k in the distance tree V is p. The function optimal_filter_up is called once for each node in the distance tree. The calls to max_gap are made with D set to the length of the substring of P represented by the nodes children. The calls to max_gap use $O(p)$ time to operate on the nodes at height k of the distance tree. Since the height of the distance tree is $\log p$, the time used by the calls to max_gap is $O(p \log p)$. The remaining lines in the function optimal_filter_up each require constant time and the function is called $O(p)$ times so the total time required by the function optimal_filter_up when called from search is $O(p \log p)$. We have already seen that the function compare_elements uses $O(p)$ time and the function search is iterated $O(t)$ times giving the running time of $O(tp \log p)$. ▮

The following lemma may be used in place of lemma 7.2 to show that the algorithm given in Figure 7.5 will place gaps optimally.

Lemma 7.3 *Given a pattern P, a text T, an alignment X_i, and a score, s, for the alignment of P in T at position i; inserting a block of characters of length b, to T (potentially adding a length b gap in the alignment) will cause the algorithm to reduce the score of the alignment by at most $\mathcal{G}(b)$.*

Proof. Assume that B was inserted into the substring of T that P aligns with in the alignment X_i, otherwise the alignment score would not be reduced at all. Let $T' = T_0 \cdots T_j B_0 \cdots B_{b-1} T_{j+1} \cdots T_{t-1}$. When the gap is introduced following position j in T we must consider the substring of P, $P_{l \cdot 2^k+1} \cdots P_{(l+1)2^k}$, that contains P_h where $X_i[h] = j$. Let X_i' be the alignment after the insertion of B.

In the function optimal_filter_up, Z_l is the penalty for inserting the gap between $P_{l \cdot 2^k + 2^{k-1} - 1}$ and $P_{l \cdot 2^k + 2^{k-1}}$ and anchoring the substring $P_{l \cdot 2^k + 2^{k-1}} \cdots P_{(l+1) \cdot 2^k - 1}$. The variable Z_r is the penalty for inserting the gap at the same position, but anchoring the substring $P_{l \cdot 2^k} \cdots P_{l \cdot 2^k + 2^{k-1} - 1}$. The first call to the function max_gap on line 8 of the algorithm in Figure 7.5 will compute the change in score when the gap is optimally placed in the substring $P_{l \cdot 2^k} \cdots P_{l \cdot 2^k + 2^{k-1} - 1}$ and the second call to the function max_gap will compute the change in score when the gap is optimally placed in the substring $P_{l \cdot 2^k + 2^{k-1}} \cdots P_{(l+1) \cdot 2^k - 1}$. Since we know the alignment scores of the strings $P_{l \cdot 2^k} \cdots P_{l \cdot 2^k + 2^{k-1} - 1}$ and $P_{l \cdot 2^k + 2^{k-1}} \cdots P_{(l+1) \cdot 2^k - 1}$, the gap penalty when the gap is placed between $P_{l \cdot 2^k + 2^{k-1} - 1}$ and $P_{l \cdot 2^k + 2^{k-1}}$, the change in score when the gap is placed optimally, and the alignment score when no gap is inserted between the two halves of P we can compute the optimal alignment score of P in T when a single gap is inserted into P. ▮

7.4.3 Constructing the alignment

Theorem 7.5 *The algorithm to construct the alignment runs in $O(tp \log p)$ time when given T, a text string of length t and P, a pattern string of length p.*

Proof. The basic algorithm was modified by adding lines 15 and 16 and since these lines use no iteration involving other lines we can add the time to execute lines 15 and 16 to the time of a call to construct_filter_up. In the worst case lines 15 and 16 will be executed once during each call to the function construct_filter_up. Line 15 executes in constant time and line 16 executes in time proportional to the length of the substring represented by the internal node W that construct_filter_up was called with. We saw in the proof of Theorem 7.4 that the sum of the length of the substrings represented by the nodes of height k in the distance tree V is p. Lines 15 and 16 use $O(p)$ time for each level of the distance tree V. Since the height of the distance tree is $O(\log p)$ and the the function construct_filter_up is called $O(t)$ times, the time used to construct an approximate optimal alignment using the algorithm we presented is $O(tp \log p)$.

Theorem 7.6 *The algorithm to construct the alignment will use $O(p \log p)$ work space when given T, a text string of length t and P, a pattern string of length p.*

Proof. We added the field max_array to each node of the distance tree. The length of max_array is proportional to the length of the substring of P represented by the node of the distance tree. We know that the sum of the length of the substrings represented by the nodes of height k in the distance tree V is p and the height of the distance tree is $\log p$ so the size of the distance tree is $O(p \log p)$.

7.4.4 Inversions

By computing a distance tree for both the string and the inverse of the string we have modified the basic algorithm to find alignments with inversions. The order of time and space used by the algorithm to align strings with inversions remains the same. Since we are computing two distance trees the constant of proportionality increases by about a factor of two.

Theorem 7.7 *The algorithm to compute the alignment score allowing inversions will run in $O(tp)$ time when given T, a text string of length t and P, a pattern string of length p.*

Proof. During each of the $O(t)$ iterations of the while loop in the function invert_search, compare_elements will be called twice and invert_filter_up once. Each call to the function compare_elements will use $O(p)$ time so each call to the function invert_filter_up will examine each of the nodes in the distance trees V and IV. There are $O(p)$ nodes in the distance trees V an IV. Each of the lines 6 through

29 in Figure 7.10 can be executed in constant time. Each call to invert_filter_up from invert_search will use $O(p)$ time. The algorithm to compute the alignment score allowing inversions will use $O(tp)$ time. ∎

Theorem 7.8 *The algorithm to compute the alignment score allowing inversions will use $O(p)$ work space when given T, a text string of length t and P, a pattern string of length p.*

Proof. The data structures in the function invert_search are P, T, V, and IV. P, by definition, is $O(p)$. Since V and IV are binary trees, each node of the trees has constant size, and the tree has $2p-1$ nodes, the size of V and IV will be $O(p)$. During iteration i of the while loop in the function search we only need the length p substring $T_{i-p+1} \cdots T_i$ of T. ∎

7.4.5 Parallel algorithm

Lemma 7.4 *The function* filter_up *in figure 7.13, when given two complete binary trees, $V1$ and $V2$, with $2p-1$ nodes each, executes in $O(1)$ time with $O(p)$ EREW PRAM processors.*

Proof. Let processor j perform the computations of the j^{th} iteration of the parallel for loop. Node N_j of $V1$ is accessed only by processor j so the tree $V1$ is never written to or read from by more than one processor at one time. Similarly for the nodes of the tree $V2$. Each of the variables Z_{lj}, Z_{rj}, and Z_{nj} are dependent only on N_j, L_j, and R_j, variables that can be stored locally on processor j. Since all of the information needed to update node j of $V1$ is stored locally on processor j, each iteration of the parallel for loop can be done independently. Each iteration can be computed in constant time since each of the lines in the parallel for loop can be done in constant time. ∎

Theorem 7.9 *The algorithm presented in Figures 7.11, 7.12, 7.13, and 7.14, when given T, a text string of length t, and P, a pattern string of length p, $t > p$, will find i, the position of the approximate alignment of P in T using $O(t)$ time and $O(p)$ words of working memory with $O(p)$ EREW PRAM processors.*

Proof. The function compare_elements can be computed in constant time with p EREW PRAM processors. Processor j will compute the j^{th} leaf value of V and the values stored in the leaves of V are dependent only on the values of $S1_j$ and $S2_j$. By lemma 7.4 we know that the function filter_up can be computed in constant time with $O(p)$ EREW PRAM processors. In the function update, we know that at most one of $V1_j$ and $V2_j$ for each $0 \leq j < |V1|$ has been updated. Therefore only one of the "if" statements in the function update will be true and there will be no time when we are trying to access the same location in memory with more than one processor. The function update can be computed in constant time with $O(p)$ EREW PRAM processors. Since each statement in the while loop of the function search can be done in constant time with $O(p)$ EREW PRAM processors and the number of iterations of the while loop is $|T| + |P| + \log(|P|)$, the function search can be done in $O(t)$ time with $O(p)$ EREW PRAM processors.

▮

The following theorem shows that the EREW PRAM model of computation is stronger than we need. If we can arrange the processors as a tree with communication links from children to parents and an extra link between each adjacent pair of leaves of the tree, then we do not need shared memory.

Theorem 7.10 *The algorithm presented in Figures 7.11, 7.12, 7.13, and 7.14, when given T, a text string of length t, and P, a pattern string of length p, will find i, the position of the approximate string alignment of P in T using $O(1)$ space on each of $O(p)$ EREW PRAM processors.*

Proof. In the function compare_elements each of the $O(p)$ processors needs to access one character from P, one character from T, one node from the tree, and a local copy of i, the position that P is being aligned with in T. In the function update, each processor needs to have access to one node of each of the trees $V1$ and $V2$. The function filter_up requires access to three nodes of the tree $V1$ and $V2$ and uses the local variables N_j, L_j, R_j, Z_{lj}, Z_{rj}, and Z_{nj}. The function search has the local variables l, a, i, max_score, max_pos and also needs to access the roots of the trees $V1$ and $V2$. The locations of the characters of P are static and can be stored in predefined locations associated with the processors that need them.

The nodes of the trees and the text, T, are not static. At iteration i of the while loop in search, the value of T_j is dependent only on the value of T_{j-1} at iteration $i-1$ for $i-p+1 < j \leq i$. At the end of iteration i, each processor associated with at leaf node needs to communitate with the processors associated with the adjacent leaf nodes. This communication is a "left shift" of the text, T. The values of a particular node in one of the distance trees at iteration i is dependent only on the value of its two children nodes stored in the other distance tree at iteration $i-1$. The processor associated with an internal node needs to communicate with the two processors associated with the children. ▮

7.5 Examples and comparisons

We align several pairs of sequences retrieved from GenBank[38] using our divide and conquer algorithm and the dynamic programming algorithm of Myers & Miller [206]. We will see that both algorithms perform well when the alignment requires only the insertion of gaps. The dynamic programming algorithm will fail to find a good alignment if segments of the sequences have been rearranged or inverted. Our algorithm will find a good alignment in both cases.

7.5.1 Triosephosphate isomerase gene

G. gallus			*A. nidulans*		
Exon	Start	End	Exon	Start	End
1	324	435	1	309	346
2	1269	1392	2	468	747
3	1487	1571	3	803	877
4	1707	1839	4	975	1087
5	2070	2155	5	1150	1364
6	2292	2379	6	1425	1453
7	2634	2752			

Table 7.1: The starting and ending position of each exon in the TIM genes of *G. gallus* and *A. nidulans*.

To compare the algorithms when aligning sequences with gaps we have chosen the triosephosphate isomerase (TIM) gene. The TIM gene codes for a well characterized, highly conserved, and ancient protein and has therefore been used by several researchers to study the evolution of gene structure. [259] We have chosen the TIM gene because, although the protein is well conserved, the number and position of the introns vary from species to species. We will use the alignment algorithms to align the TIM genes from *Aspergillus nidulans* (fungus), a 1900 bp sequence with GenBank accession number M13362, and *Gallus gallus* (chicken), a 3900 bp sequence with GenBank accession number M11941. The TIM gene from *G. gallus* is composed of seven exons (coding regions) separated by six introns (non–coding regions). The exons are concatenated to form the specification of the protein. The TIM gene in *A. nidulans* has six exons separated by five introns. Table 7.1 gives the starting and ending positions of the exons in the TIM gene from *G. gallus* and *A. nidulans*. Since the exons code for a protein that must be functional if the organism is to survive, there is selective pressure to conserve the information in the exons. The introns of the gene are not subjected to this selective pressure and may therefore be modified at a higher rate. Using the assumptions that

1. The TIM genes from *G. gallus* and *A. nidulans* evolved from a common ancestor.

2. The introns are evolving at a higher rate than the exons.

we expect the exons from the TIM genes of *G. gallus* and *A. nidulans* to align with one another rather than with introns. Using the TIM genes from these organisms, we will see that both algorithms produce reasonable alignments while introducing gaps.

Both of the algorithms that we investigate penalize gaps. Each algorithm requires a penalty to "break" the sequence, a penalty that is dependent on the length of the gap, and a scoring matrix. Since we know approximately where each exon from *A. nidulans* should align in *G. gallus*, we used these alignments calibrate the gap penalty function for each algorithm. The penalty function that we used in our divide and conquer algorithm was $1 + \log(|\mathrm{gap}|)$. The gap penalty function that we used for the dynamic programming algorithm was $4 + |\mathrm{gap}|$. For both algorithms, we assigned the score of 2 for an exact match and 0 for all mismatches.

Initially, each exon from *A. nidulans* was individually aligned with the entire TIM gene from *G. gallus*. The results of aligning each exon using the dynamic programming algorithm and the divide and conquer algorithm are given in Table 7.2. Since the divide and conquer algorithm is most easily applied to pattern strings of length 2^k, we choose fragments that contained all or nearly all of a particular exon from *A. nidulans*. The first and second columns of the table give the starting and ending position used for each exon in the 1900 bp TIM gene from *A. nidulans*. We aligned each exon from *A. nidulans* with the entire 3900 bp TIM gene from *G. gallus*. Columns three and four of the table give the positions in the complete *G. gallus* TIM gene that fragments of the exon from *A. nidulans* aligned using the divide and conquer algorithm. An exon from *A. nidulans* frequently did not align with a single subsequence in the *G. gallus* sequence, but with a couple of separate subsequences. These are reported as a

		Divide and Conquer		Dynamic Programming	
A. nidulans		*G. gallus*		*G. gallus*	
Exon Start	Exon end	Match Start	Match End	Match Start	Match End
309	346	324	357	330	373
		850	878	1483	1498
468	723	1200	1450	366	421
				718	731
				1268	1400
				1425	1452
				1524	1547
803	866	1707	1770	323	333
				690	712
				1743	1771
975	1102	2024	2154	2090	2120
				2330	2343
1150	1405	2119	2134	2121	2156
		2262	2381	2293	2381
		2637	2700	2636	2720
		2602	2654	2845	2889
1425	1456	1289	1320	2727	2742
				2755	2767

Table 7.2: Positions of the TIM gene from *G. gallus* that align the with exons of the TIM gene from *A. nidulans*. Each exon from *A. nidulans* was aligned seperately in the entire TIM sequence of *G. gallus* using both algorithms.

G. gallus			*A. nidulans*		
Exon Number	Start Position	End Position	Match Start	Match End	Comment
1	324	435	157	217	intron
2	1269	1392	538	663	exon 2
3	1487	1571	700	747	exon 2
4	1707	1839	804	885	exon 3
5	2070	2153	1046	1113	exon 4
6	2292	2379	1187	1273	exon 5
7	2634	2752	1277	1366	exon 5

Table 7.3: Results of aligning the complete TIM gene from *A. nidulans* with the complete TIM gene from *G. galus* using the Myers & Miller dynamic programming algorithm.

set of starting and ending positions in the *G. gallus* columns for the given *A. nidulans* exon. The final two columns of the table give the same information for the dynamic programming algorithm. Examining Table 7.2, we can see that with the exception of the final short exon placed by the divide and conquer algorithm, each of the exons from *A. nidulans* are placed reasonably well in the *G. gallus* sequence by both algorithms.

We used each algorithm to align the complete TIM gene from *A. nidulans* with the complete TIM gene from *G. gallus*. To have some gross measure of the quality of the alignment, we examine how the exons are aligned. Table 7.3 gives the alignment produced using the dynamic programming algorithm of Myers & Miller. The first three columns of the table refer to the exons from the *G. gallus* TIM gene giving the exon number and its starting and ending position. The last three columns of the table refer to the TIM gene from *A. nidulans*. The match start and match end columns give the position in the *A. nidulans* gene that the exon from *G. gallus* aligned with. The comment column lists exon numbers from

A. nidulans			*G. gallus*		
Exon Number	Start Position	End Position	Match Start	Match End	Comment
1	< 309	346	338	364	exon 1
2	468	747	1306	1385	exon 2
			1487	1571	exon 3
3	803	877	2555	2618	intron
4	975	1087	2675	2809	exon 7
			2087	2118	exon 5
5	1150	1364	2321	2382	exon 6
			2639	2702	exon 7
6	1425	> 1453			intron

Table 7.4: Results of aligning the complete TIM gene from *A. nidulans* with the complete TIM gene from *G. galus* using our divide and conquer algorithm.

A. nidulans that correspond with the matched region. Exon 1 from *G. gallus* and exon 6 from *A. nidulans* are not aligned with their corresponding exons. This is not unexpected since the matches between both the first exons and the last exons are short, less than 40 base pairs. A large segment of exon 4 from *G. gallus* is aligned with an intron in *A. nidulans*.

Table 7.4 gives the alignment produced using our divide and conquer algorithm. The first three columns of the table refer to the exons from the *A. nidulans* TIM gene giving the exon number and its starting and ending position. The last three columns of the table refer to the TIM gene from *G. gallus*. The match start and match end columns give the position in the *G. gallus* gene that the exon from *A. nidulans* aligned. The comment column lists exon numbers from *G. gallus* that correspond with the matched region. Exons 3 and 6 of *A. nidulans* are aligned with introns in *G. gallus* and a large segment of exon 4 from *G. gallus* is aligned with exon 7 of *A. nidulans*. The alignments produced by both algorithms clearly

P. miliaris (V01143)			*P. miliaris* (V01144)		
Exon	Start	End	Exon	Start	End
H4	305	616	H1	875	1444
H2B	1609	1980	H4	2280	2591
H3	2598	3008	H2B	3832	4203
H2A	3688	4062	H3	5098	5508
H1	4899	5468	H2A	6416	> 6700

Table 7.5: Exon positions in two alleles of the *Psammechinus miliaris* histone complex.

show that the complete TIM genes from *A. nidulans* and *G. gallus* are similar.

7.5.2 Histone gene cluster of *P. miliaris*

To examine the performance of the algorithms when segments of the sequences are rearranged, we aligned two alleles of the histone gene cluster from two individuals of *Psammechinus miliaris* (sea urchin). The histone genes in *P. miliaris* are in the order {H1 H4 H2B H3 H2A} and are repeated about 450 times [245]. Two alleles for this gene cluster are in the GenBank database with accession numbers V01143 and V01144 [39]. In the 6000 bp sequence with accession number V01143, the genes are in the order {H4 H2B H3 H2A H1}, and in the 6700 bp sequence with accession number V01144, the genes are in the order {H1 H4 H2B H3 H2A}. Table 7.5 gives the beginning and ending position of each exon in the histone gene cluster for both alleles, V01143 and V01144.

We have used the dynamic programming algorithm to align the two *P. miliaris* histone gene clusters and the results of the alignment are presented in Table 7.6. The first three columns give the exon names and positions of the exons in the V01143 allele. The last three columns give the position in the V01144 allele that exons from the V01143 allele aligned. The comment column is the region (exon name) in the V01144 allele that was matched. The dynamic programming

V01143			V01144		
Exon Name	Start Position	End Position	Match Start	Match End	Comment
H4	305	616	909	1190	In H1
H2B	1609	1980	2134	2496	Part in H4
H3	2598	3008	3109	3520	Intron
H2A	3688	4062	4016	4389	Part in H2B
H1	4899	5468	5218	5709	Part in H3

Table 7.6: Results of the alignment of the complete histone complex sequences of *P. miliaris* V01143 and *P. miliaris* V01144 using the dynamic programming algorithm.

algorithm fails to properly align any of the genes, but it should not be surprising since the dynamic programming algorithm was not intended to align sequences that have been rearranged in this way.

We have also used our divide and conquer algorithm to align the two *P. miliaris* histone gene clusters and the results of the alignment are presented in Table 7.7. The divide and conquer algorithm does properly align the exons in the V01143 sequence with the corresponding exons in the V01144 sequence, despite the rearrangement of the exon positions within the gene cluster. The H2B gene was not aligned correctly, but only the first 15 of 370 base pairs of the H2B gene in V01144 are known.

If the dynamic programming algorithm had been used naively to align the two complete histone gene clusters from *P. miliaris* it might be wrongly concluded that the sequences are not similar. Using our divide and conquer algorithm, it is clear that the sequences are similar.

7.5.3 Histone gene cluster of *X. laevis*

V01143			V01143		
Exon Name	Start Position	End Position	Match Start	Match End	Comment
H1	875	1444	4899	5140	H1
			5415	5468	H1
H4	2280	2591	329	399	H4
			440	456	H4
H2B	3832	4203			Intron
H3	5098	5508	2620	3008	H3
H2A	4616	> 6700	3688	3971	H2A

Table 7.7: Results of the alignment of the complete histone complex sequences of *P. miliaris* V01143 and *P. miliaris* V01144 using our divide and conquer algorithm.

X. laevis (X03017)			*X. laevis* (X03018)		
Exon	Start	End	Exon	Start	End
H3	1	228	H1A	615	1244
H4	1422	1730	H2B	1695	2072
H2A	2351	2740	H2A	4372	4761
H2B	3120	3497	H3	5555	5962
H1B	10902	11561	H4	6949	7257
H3	12726	13133			
H4	14290	14598			

Table 7.8: Exon positions in two alleles of the *X. laevis* histone complex.

Dynamic programming				Divide and conquer			
3017	3018			3017	3018		
Exon Name	Match Start	Match End	Comment	Exon Name	Match Start	Match End	Comment
H1B	671	1222	exon H1A	H4	6948	7255	exon H4
H2B	1695	2071	exon H2B	H2A	4760	4372	exon H2A
H2A			fragments	H2B	1694	2070	exon H2B
H3			fragments	H1B	614	1239	exon H1A
H4	6948	7256	H4	H3	5961	5555	exon H3

Table 7.9: Results of aligning the individual exons from the histone gene cluster in *X. laevis* X03017 in the full histone gene cluster sequence from *X. laevis* X03018 using both algorithms.

We use two histone gene clusters from *Xenopus laevis* to demonstrate aligning sequences with inversions. The histone gene cluster from *X. laevis* with GenBank accession number X03017 is 14942 base pairs in length and the histone gene cluster from *X. laevis* with GenBank accession number X03018 is 8592 base pairs long. [220] In these two sequences, the inverse of H2A in X03017 will match H2A in X03018 and the inverse of H3 in X03017 will match H3 in X03018. Table 7.8 shows the starting and ending positions of the genes in the histone gene clusters of the two sequences of *X. laevis* that we are interested in.

The algorithms were used to align individual genes from the X03017 sequence in the complete X03018 sequence. Table 7.9 shows the results of using both algorithms to do the alignments. The dynamic programming algorithm finds the alignments for the genes H1B, H2B, and H4, but the alignments for H2A and H3 are missed. If the genes from the inverse of the X03017 sequence were aligned with the X03018 sequence, the H2A and H3 genes would be aligned properly but the genes H1B, H2B, and H4 would be missed. Using our divide and conquer algorithm to align the individual genes from the X03017 sequence with the complete

X03018			X03017		
Exon Name	Exon Start	Exon End	Match Start	Match End	Comment
H1A	615	1244	2079	2688	partially in H2A
H2B	1695	2072	3117	3493	H2B
H2A	4372	4761	5713	6057	intron
H3	5555	5962	6801	7198	intron
H4	6949	7257	8117	8379	intron

Table 7.10: Positions of the histone gene cluster from *X. laevis* X03017 that match the exons from the histone gene cluster from *X. laevis* X03018. The alignment was done using the complete histone gene cluster from each *X. laevis* sequence using the Myers & Miller dynamic programming algorithm.

X03018 sequence, each of the genes is properly aligned.

Tables 7.10 show the results of aligning the complete X03017 and X03018 sequences using the dynamic programming algorithm. The first three columns show the exons and their starting and ending positions of the gene from the X03018 sequence. The last three columns give the positions in the X03017 sequence that the exons from the X03018 sequence aligned with. The comment column given the name of the region of alignment. Table 7.10 shows that the dynamic programming algorithm fails to align the histone gene clusters from the X03017 and X03018 sequences. Again, this is not surprising since the algorithm was not designed to find alignments with inversions and rearranged segments.

Our divide and conquer algorithm finds the alignment given in Table 7.11. The divide and conquer algorithm aligned the genes from the X03017 sequence with the segment of the X03018 sequence as indicated by the match start and match end columns of Table 7.11. The comment column indicates the gene at the location from match start to match end in the sequence X03018. Each gene from the X03017 sequence is properly aligned with the corresponding gene from

X03017			X03018		
Exon Name	Exon Start	Exon End	Match Start	Match End	Comment
H3	1	128	5774	5553	H3
H4	1422	1730	6949	7256	H4
H2A	2351	2740	4702	4370	H2A
H2B	3120	3497	1697	2072	H2B
H1B	10902	11561	615	1241	H1A

Table 7.11: Positions of the histone gene cluster from *Xenopus laevis* X03017 that match the exons from the histone gene cluster from *X. laevis* X03018. The alignment was done using the complete histone gene cluster from each *X. laevis* using our divide and conquer algorithm.

the X03018 sequence. Notice that the alignments of the genes H3 and H2A start at a position greater than they end, this is because it is actually the inverse of the gene that matched.

The *X. laevis* histone gene cluster example shows that the dynamic programming algorithm can fail to properly align sequences that contain inverted segments. The divide and conquer algorithm properly aligns the histone gene clusters from *X. laevis* since it is capable of inverting subsequences in the alignment that it creates.

7.5.4 Running Time

The basic algorithm and its variants that have been developed in this paper are simple to implement and can be used to solve problems that are large enough to be of practical interest. The basic algorithm can be used alone, or in combination with one or more of the extensions that are discussed in section 7.3. In this section we have seen the algorithm used to align the TIM genes from *G. gallus* and *A. nidulans*, two alleles for the histone gene cluster from *P. miliaris*, and two alleles

Optimal	Constructive	Inversions	Time
			213
		√	397
	√		297
	√	√	556
√			517
√		√	1021
√	√		603
√	√	√	1153

Table 7.12: Running time in CPU seconds to compute an alignment of *A. nidulans* with *G. gallus* using several variants of our divide and conquer algorithm.

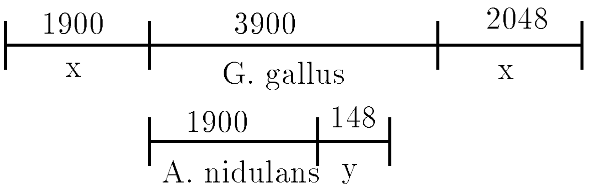


Figure 7.18: Schematic representation of input used to time the divide and conquer algorithm.

for the histone gene cluster from *X. laevis*.

We have measured the CPU time used to compute the alignment of the TIM genes from *G. gallus* and *A. nidulans* using the basic divide and conquer algorithm. Each of the extensions to the basic algorithm can be used individually or in combination and the CPU time used by each of the combinations was also measured. Table 7.12 gives the running time of the basic algorithm with all possible combination of the optimal, constructive, and inversion extensions. All times were measured on a Sun IPC with 8 Mb of memory.

The current implementation of the algorithm requires that the length of

Number of Processors	Run Time	Speedup	Percent Utilization
20	140	16.7	83.5
16	172	13.6	85.0
12	224	10.5	87.5
8	323	7.26	90.8
4	612	3.83	95.8
2	1191	1.97	98.5
1	2344	1.0	100.0

Table 7.13: Speedup of the parallel divide and conquer algorithm to align a string of lengh 4096 within a string of length 8192.

one of the sequences be a power of two. This is easily accomplished by padding one of the sequences. Since our algorithm allows rearrangement of the sequences in the alignment, the sequences need to be padded to allow the prefixes of one string to be aligned with the suffixes of the other. Figure 7.18 represents the input that we used to time the algorithm. The *G. gallus* sequence was padded with a 1900 character prefix and and 2048 character suffix and the A. nidulans sequence was padded with a 148 character suffix. The characters used to pad the *G. gallus* sequence are distinct from the characters used to pad the *A. nidulans* sequence and none of the characters used to pad the sequences appear in the sequences.

The dynamic programming algorithm of Myers & Miller was used to align the TIM genes from *G.gallus* and *A. nidulans*. Their algorithm used 148 CPU seconds to align the sequences and used 298 CPU seconds to align the input given to our divide and conquer algorithm.

A simple version of the divide and conquer approximate string matching algorithm has been implemented on a sequent balance 21000 parallel computer with 28 processors. The program was asked to align a string of length 4096 within a string of length 8192 using a varying number of processors from one to twenty. Figure 7.13 gives the number of processors used to align the sequences, the time in CPU seconds, the speedup achieved by using multiple processors, and the percent utilization of the processors. Using twenty processors decreases the time required to align the strings by a factor of 16.7 from the time used by one processors.

7.6 Summary and future work

There is evidence that evolution may proceed by moving and inverting segments of a genome in addition to changing, inserting, and deleting individual bases in the genome. The algorithms that are currently in use to align genetic sequences frequently fail to consider the move and invert operations. We have introduced the string to string correction problem to allow the comparison of gene sequences using both types of operations; the change, insert, and delete operations on individual bases, and the move and invert operations on segments of the sequences.

We have described a simple algorithm to find an approximate solution to the string to string correction problem. This is followed by several variations of this algorithm including a version to construct the alignment, a version to place the gaps optimally, and a parallel version of the algorithm. The algorithm runs quickly, in time proportional to the product of the length of the sequences being aligned and uses a very small amount of work space, proportional to the length of the shorter sequence.

It may be possible to construct a "smart" disk controller based on our parallel divide and conquer algorithm. The parallel algorithm uses only a few simple operations and never needs to back up in the text. Each processor would need a small, constant sized memory, and would need to communicate with at most four other processors. Such a disk controller would allow database search algorithm to start with only the sequences that are similar to the query sequence.

The divide and conquer approach should be easy to apply to multi–dimensional approximate matching. The idea of inverted substrings in the one dimensional case can be extended to transformations of sub–patterns in the multidimensional case.

We wish to thank Gene Myers and Webb Miller for kindly providing the source for their excellent implementation of the dynamic programming algorithm to find optimal alignments. We also thank Charlotte Rasmussen for suggesting to us that alignments should include inversions.

Chapter 8

Ideas for Future Research

During the course of completing the research presented in this thesis, a number of interesting problems arose that I did not pursue. In this chapter I describe a few of the problems and present my ideas for possible solutions to the problem.

8.1 Nucleotide / amino acid alignment with frame shifts

Sequencing a specific gene that has previously been sequenced in several other organisms can be done more efficiently by using the homologous sequences to help select the specific gene of interest in the new organism. Once the DNA coding for homologous proteins has been sequenced from several organisms, it may be possible to find regions of the DNA that are conserved in most or all of the organisms. Using these conserved regions it may be possible to construct a piece of tagged DNA that is used as a probe to mark similar sequences in the new organism. The DNA segment of interest is isolated, cloned, and sequenced. When reconstructing the original DNA sequence from the sequence data, we not only have the overlap information, but also the homologous sequences to guide us.

Once a preliminary sequence has been reconstructed from the sequence fragment data, this preliminary sequence can be aligned with a homologous protein sequence. Since three consecutive bases code for a single amino acid in the protein, the nucleotides are divided into triples, translated into amino acids, and aligned with the amino acid sequence. The translation can begin at position one, two, or three of the nucleotide sequence, each generating a different amino acid sequence (the three reading frames). If there are insertion or deletion errors in the nucleotide sequence, the reading frame will shift at that position. Knowing the position of insertions or deletions in the nucleotide sequence relative to the amino acid sequence can allow the researcher to reexamine the data in the proximity of the insertion or deletion to insure that no nucleotides were missed or added while reading and entering the data.

The standard dynamic programming algorithm to align sequences can be used to align nucleotide sequences with amino acid sequences, but the standard operations of insert a character, delete a character, and substitute a character do not allow the reading frame to shift. In 1986, H. Peltola, H. Soderlund, & E. Ukkonen [219], and later in 1992, M. Masreliez & J. Holloway [192], described a modification to the dynamic programming algorithm that allows the reading frame to shift. M. Masreliez & I are further modifying the algorithm to "adjust" the reading frame as the alignment is constructed so that a reading frame shift is penalized only once, not each time a nucleotide triplet in a shifted segment is aligned with the corresponding amino acid.

The algorithms of H. Peltola, H. Soderlund, & E. Ukkonen [219] and M. Masreliez & J. Holloway [192] run in $O(mn)$ time and use $O(mn)$ space. Since the sequences being aligned can be several thousand bases long, these algorithms can quickly exhaust available memory. The method of D. Hirschberg[132] and E. Myers & W. Miller [206] to reduce the space used by the standard dynamic programming algorithms to solve the edit distance problem from $O(mn)$ to $O(m+n)$ may be applicable to the problem of aligning nucleotide sequences with amino acid sequences with frame shifts.

The algorithms of H. Peltola, H. Soderlund, & E. Ukkonen [219] and M. Masreliez & J. Holloway [192] align a nucleotide sequence with a single amino acid sequence. In the process of finding conserved regions of the amino acid sequence, the amino acid sequences are frequently aligned. It could be more informative to align the nucleotide sequence with the group of aligned amino acid sequences instead of an individual amino acid sequence.

8.2 Divide and conquer multi–dimensional matching

The ideas that were developed in chapter 7 allowed us to find an alignment of a pattern string within a text string when substrings of the pattern string could be rearranged, inverted, and gaps inserted. These ideas can be extended to two or more dimensional approximate pattern matching. In the one dimensional case, the alignment of a string is found by aligning each half of the string and then constructing an alignment of the entire string by possibly placing a gap between the halves. In the case of d dimensions, the d–dimensional pattern is broken into 2^d parts by dividing the pattern in half along each of the d axes. Once each of the parts is aligned using this same procedure, a score for the whole pattern is computed by comparing the score for the pattern with no gaps between the parts and the score of the optimal alignment of the parts minus any gap penalties.

Once the basic d–dimensional approximate alignment algorithm has been completed, the idea of an inverted pattern can be considered. Any transformation that can be applied and results in another pattern with the same size and shape can be used as the inverted strings were used in the one dimensional case. There is no reason to limit the number of transformations to one, so we could align patterns with different pieces transformed in different manners. In the two dimensional case, this might be a pattern with two parts that have been rotated independently.

8.3 Updating suffix arrays

Suffix arrays are a space efficient data structure that enables questions of the form "Is the pattern string P a substring of the text string T?" to be answered in $O(|P| + \log |T|)$ time. Storing a large text that needs to be searched frequently in a suffix array is one way of providing very quick response to substring queries.

If the text changes frequently we need some way of quickly updating the suffix array. The algorithm given by U. Manber [189] to construct a suffix array of T will take $O(|T|)$ expected time. If T is large and updated frequently, the cost of recomputing the suffix array after each update of T may be prohibitive.

One possible alternative to recomputing the suffix array for T after each update is to modify the suffix array or some similar data structure to reflect the changes. The updates to T could be restricted to additions of text, no deletions or changes, since adding text is the only operation that happens to the genetic sequence databases that I am interested in.

Another alternative is to recompute the suffix array in parallel. Parallel algorithms for constructing suffix trees, a data structure similar to suffix arrays have been given [21]. Although I have started working on parallelizing the sequential algorithm of U. Manber [189], it is not clear yet if this algorithm can be efficiently parallelized.

8.4 Chaos game theory and approximate string matching

In 1990, H. Jeffrey published the paper "Chaos game representation of gene structure" that described an algorithm to construct a visual representation of a nucleotide sequence. Label the four vertices of a square with A, C, G, and T and set the current position to be the center of the square. Moving from left to right in the nucleotide sequence, for each nucleotide in the sequence, move from the current position in the square halfway to the vertex labeled with the nucleotide that is being examined and place a dot at this new position. Using this algorithm to view nucleotide sequences reveals recurring patterns and regions of self similarity.

Each point in the pattern is the result of a particular prefix of the sequence. If we think of each point in the pattern as a substring of the sequence, in some sense similar substrings will cluster in similar positions in the square. For example, if the upper right corner of the square is labeled with 'G', all substrings that end with "GG" appear in the upper right quadrant of the upper right quadrant (the upper right $1/16^{th}$) of the square. Similarly, any length three substring that has “GG” as a prefix appears in the upper right quadrant of the upper right quadrant of whatever quadrant is associated with the third character of the substring. By using these regions of similar strings, it should be possible to search sequences or databases for approximate matches with short sequences. Studying the one dimensional case with two vertices and the characters ’0’ and ’1’ in the sequence might facilitate the development of algorithms.

It may be possible to apply the ideas of Jeffrey to visualizing amino acid sequences. Since there are 20 amino acids commonly used to construct proteins, using a dodecahedron (20 vertices) instead of a square and applying a similar algorithm may produce interesting representations of amino acid sequences. One immediate problem with this approach is that results are in three dimensions and must be viewed in two dimensions.

8.5 Comparing a sequence against a database of motifs

In 1975, A. Aho & M. Corasick [4] described an algorithm that searches a text string for the occurrence of any pattern in a set of pattern strings in time proportional to the length of the text string. The algorithm constructs a pattern matching machine from the set of patterns and then uses the text as input to the pattern matching machine. The construction of the pattern matching machine can be done in time proportional the sum of the lengths of the patterns in the set of patterns.

PROSITE is a database of several hundred amino acid motifs that is maintained by A. Bairoch [29]. Each of the motifs is expressed as a set of amino acids that may occur at each of the positions in the motif. For example, the motif

$$\text{C–X–\{P\}–X}_5\text{–[GS]–X}_2\text{–C}$$

specifies that a cysteine (C) be in the first position, any amino acid may be in the second position, any amino acid except a proline (P) may be in the third position, the fourth through eighth positions may contain any amino acids, the ninth position may contain either a glycine (G) or a serine (S) followed by any two amino acids and then a final cysteine (C).

It may be easy to apply the Aho–Corasick key word searching algorithm to the problem of locating motifs from the PROSITE database in an amino acid sequence.

Bibliography

[1] R. M. Abarbanel, P. R. Wieneke, E. Mansfield, D. A. Jaffe, and D. L. Brutlag. Rapid searches for complex patterns in biological molecules. *Nucleic Acids Research*, 12:263–280, 1984.

[2] K. Abrahamson. Generalized string matching. *SIAM Journal of Computing*, 16:1039–1051, 1987.

[3] K. Abremski, K. Sirotkin, and A. Lapedes. Application of neural networks and information theory to the identification of *e. coli* transcriptional promoters. Technical Report LA–UR–91–729, Los Alamos National Laboratory, 1991.

[4] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[5] A. V. Aho, D. S. Hirschberg, and J. D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the Association for Computing Machinery*, 23:1–12, 1976.

[6] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, MA, 1974.

[7] L. Allison, C. S. Wallace, and C. N. Yee. Induction inference over macro–molecules. Technical Report 90/148, Monash University, Clayton, Victoria, Australia, 3168, 1990.

[8] L. Allison and C. N. Yee. Minimum message length encoding and the comparison of macromolecules. *Bulletin of Mathematical Biology*, 52:431–453, 1990.

[9] H. Almagor. A Markov analysis of DNA sequences. *Journal of Theoretical Biology*, 104:633–645, 1983.

[10] S. F. Altschul. Gap costs for multiple sequence alignments. *Journal of Theoretical Biology*, 138:297–309, 1989.

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

[12] S. F. Altschul and B. W. Erickson. Locally optimal subalignments using nonlinear similarity functions. *Bulletin of Mathematical Biology*, 48:633–660, 1986.

[13] S. F. Altschul and B. W. Erickson. A nonlinear measure of subalignment similarity and its significance levels. *Bulletin of Mathematical Biology*, 48:617–632, 1986.

[14] S. F. Altschul and B. W. Erickson. Optimal sequence alignments using affine gap costs. *Bulletin of Mathematical Biology*, 48:606–616, 1986.

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

[16] S. F. Altschul and D. J. Lipman. Trees, stars and multiple biological sequence alignment. *SIAM Journal of Applied Mathematics*, 49:197–209, 1989.

[17] S. F. Altschul and D. J. Lipman. Protein database searches for multiple alignments. *Proceedings of the National Academy of Science*, 87:5509–5513, 1990.

[18] A. Amir and G. M. Landau. Fast parallel and serial multidimensional approximate array matching. In R. M. Capocelli, editor, *Sequences, combinatorics, compression, security and transmission*, pages 3–24. Springer–Verlag, 1990.

[19] A. Amir and G. M. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:97–115, 1991.

[20] J. Aoe. An efficient implementation of static string pattern matching machines. *IEEE Transactions on Software Engineering*, 15:1010–1016, 1989.

[21] A. Apostolico. Parallel log–time construction of suffix trees. Technical Report CSD–TR–632, Purdue, 1986.

[22] A. Apostolico, S. Browne, and C. Guerra. Fast linear–space computations of longest common subsequences. *Theoretical Computer Science*, 92:3–17, 1992.

[23] P. Argos. A sensitive procedure to compare amino acid sequences. *Journal of Molecular Biology*, 193:385–396, 1987.

[24] I. E. Auger and C. E. Lawrence. Algorithms for the optimal identification of segment neighborhoods. *Bulletin of Mathematical Biology*, 51:39–54, 1989.

[25] D. J. Bacon and W. F. Anderson. Multiple sequence alignment. *Journal of Molecular Biology*, 191:153–161, 1986.

[26] R. A. Baeza-Yates. Improved string searching. *Software Practice and Experience*, 19:257–271, 1989.

[27] R. A. Baeza-Yates and M. Regnier. Average running time of the Boyer–Moore–Horspool algorithm. *Theoretical Computer Science*, 92:19–31, 1992.

[28] A. Bairoch. Prosite: a dictionary of protein sites and patterns. *Nucleic Acids Research*, 19:2241–2245, 1991.

[29] A. Bairoch. SEQANALREF: a sequence analysis bibliographic reference databank. *Computer Applications in the Biosciences*, 7(2):268, 1991.

[30] L. J. Barnett. Probabilistic analysis of random clone restriction mapping. Master's thesis, Washington University in St. Louis, 1990.

[31] S. Barron, M. Witten, R. Harkness, and J. Driver. A bibliography on computational algorithms in molecular biology and genetics. *Computer Applications in the Biosciences*, 7(2):269, 1991.

[32] G. J. Barton. Scanning protein sequence databanks using a distributed processing workstation network. *Computer Applications in the Biosciences*, 7(1):85–88, 1991.

[33] G. J. Barton and M. J. E. Sternberg. Flexible protein sequence patterns: a sensitive method to detect weak structural similarities. *Journal of Molecular Biology*, 212:389–402, 1990.

[34] D. C. Benson. Digital signal processing methods for biological sequence comparison. *Nucleic Acids Research*, 18:3001–3006, 1990.

[35] J. C. Beran-Koehn and W. D. Gillett. Information theoretic estimation of clone overlap probabilities. Technical Report WUCS–90–27, Washington University in St. Louis, 1990.

[36] M. P. Berger and P. J. Munson. A novel randomized iterative strategy for aligning multiple protein sequences. *Computer Applications in the Biosciences*, 7(4):479–484, 1991.

[37] A. A. Bertossi, E. Lodi, F. Luccio, and L. Pagli. Context–dependent string matching. In R. M. Capocelli, editor, *Sequences, combinatorics, compression, security and transmission*, pages 25–40. Springer–Verlag, 1990.

[38] H. S. Bilofsky and C. Burks. The genbank genetic sequence data bank. *Nucleic Acids Research*, 16:1861–1863, 1988.

[39] M. Birnstiel, R. Portmann, M. Busslinger, W. Schaffner, E. Probst, and A. Kressmann. Functional organization of the histone genes in the sea urchin psammechinus: a progress report. In *Alfred Benzon Symposium*, volume 13, pages 117–132, 1979.

[40] Bishop and Rawlings. *Nucleic Acid and Protein Sequence Analysis. A Practical Approach.* IRL Press, Oxford, 1987.

[41] M. J. Bishop and E. A. Thompson. Maximum likelihood alignment of DNA sequences. *Journal of Molecular Biology*, 190:159–165, 1986.

[42] B. E. Blaisdell. A measure of the similarity of sets of sequences not requiring sequence alignment. *Proceedings of the National Academy of Science*, 83:5155–5159, 1986.

[43] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 328–336, Baltimore, MD, 1991. ACM press.

[44] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ–tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[45] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[46] J. H. Bradford. Sequence matching with binary codes. *Information Processing Letters*, 34:193–196, 1990.

[47] D. Breslauer and Z. Galil. An optimal $O(\log \log n)$ time parallel string matching algorithm. *SIAM Journal of Computing*, 19:1051–1058, 1990.

[48] D. L. Brutlag, J. P. Dautricourt, S. Maulik, and J. Relph. Sensitive similarity searches of biological sequence databases. *Computer Applications in the Biosciences*, 6:237–245, 1990.

[49] G. N. Cameron. The EMBL data library. *Nucleic Acids Research*, 16:1865–1867, 1988.

[50] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal of Applied Mathematics*, 48:1073–1082, 1988.

[51] W. I. Chang and E. L. Lawler. Approximate string matching in sublinear expected time. In *IEEE Symposium on the Foundations of Computer Science*, pages 116–124, 1990.

[52] E. A. Cheever, G. C. Overton, and D. B. Searls. Fast Fourier transform–based correlation of DNA sequences using complex plane encoding. *Computer Applications in the Biosciences*, 7(2):143–154, 1991.

[53] J. Clayton and L. Kedes. Gel, a DNA sequencing project management system. *Nucleic Acids Research*, 10:305–321, 1982.

[54] L. Colussi, Z. Galil, and R. Giancarlo. On the exact complexity of string matching. In *IEEE Symposium on the Foundations of Computer Science*, pages 135–143, 1990.

[55] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, 1990.

[56] N. G. Core, E. W. Edmiston, J. H. Saltz, and R. M. Smith. Supercomputers and biological sequence comparison algorithms. *Computers and Biomedical Research*, 22:497–515, 1989.

[57] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw–Hill, New York, 1990.

[58] F. Corpet. Multiple sequence alignment with hierarchical clustering. *Nucleic Acids Research*, 16(22):10881–10890, 1988.

[59] A. F. W. Coulson, J. F. Collins, and A. Lyall. Protein and nucleic acid sequence database searching: a suitable case for parallel processing. *Computer Journal*, 30:420–424, 1987.

[60] M. Crochemore. Constant–space string–matching. In R. M. Capocelli, editor, *Sequences, combinatorics, compression, security and transmission*, pages 25–40. Springer–Verlag, 1990.

[61] M. Crochemore. String–matching on ordered alphabets. *Theoretical Computer Science*, 92:33–47, 1992.

[62] M. Crochemore and D. Perrin. Two–way string matching. *Journal of the Association for Computing Machinery*, 38:651–675, 1991.

[63] P. Cull and J. Holloway. A divide and conquer approach to approximate string matching. Technical Report TR–91–50–1, Oregon State University, Department of Computer Science, Corvallis, OR. 97331, 1991.

[64] P. Cull and J. L. Holloway. Algorithms for constructing a consensus sequence. Technical Report TR–91–20–1, Oregon State University, Department of Computer Science, 1991.

[65] P. Cull and J. L. Holloway. Reconstructing sequences from shotgun data. In *Sequences: Combinatorics, Compression, Security, and Transmission*. Springer–Verlag, 1991.

[66] P. Cull and J. L. Holloway. Optimistically building a consensus sequence using {–inexact matches. In *Proceedings of the Hawaii International Conference on System Sciences*, volume 1, pages 643–652, 1992.

[67] A. Danckaert, C. Chappey, and S. Hazout. 'size leap' algorithm: an efficient extraction of the longest common motifs from a molecular sequence set. Application to DNA sequence reconstruction. *Computer Applications in the Biosciences*, 7(4):509–513, 1991.

[68] D. Davison. Sequence similarity (homology) searching for molecular biologists. *Bulletin of Mathematical Biology*, 47:437–474, 1985.

[69] D. Davison and K. H. Thompson. A non–metric sequence alignment program. *Bulletin of Mathematical Biology*, 46:579–590, 1984.

[70] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. In M. O. Dayhoff, editor, *Atlas of Protein Structure*, volume 5(Suppl. 3), pages 345–352. National Biomedical Research Foundation, Silver Spring, Md., 1979.

[71] C. DeLisi. Computers in molecular biology: Current applications and emerging trends. *Science*, 240:47–52, 1988.

[72] A. S. Deshpande, D. S. Richards, and W. R. Pearson. A platform for biological sequence comparison on parallel computers. *Computer Applications in the Biosciences*, 7(2):237–247, 1991.

[73] R. F. Doolittle. *Of URFs and ORFs*. University Science Books, Mill Valley, CA, 1986.

[74] J. Dumas and J. Ninio. Efficient algorithms for folding and comparing nucleic acid sequences. *Nucleic Acids Research*, 10:197–206, 1982.

[75] T. Eilam-Tzoreff and U. Vishkin. Matching patterns in strings subject to multi–linear transformations. *Theoretical Computer Science*, 60:231–254, 1988.

[76] D. Eppstein. Sequence comparison with mixed convex and concave costs. preprint.

[77] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Annual Review of Computer Science*, 3:233–283, 1988.

[78] D. Eppstein, Z. Galil, and R. Giancarlo. Speeding up dynamic programming with application to the computation of RNA structure. preprint.

[79] D. Eppstein, Z. Galil, and R. Giancarlo. Speeding up dynamic programming. In *IEEE Symposium on the Foundations of Computer Science*, pages 488–496, 1988.

[80] D. Eppstein, Z. Galil, and R. Giancarlo. Efficient algorithms with applications to molecular biology. In R. M. Capocelli, editor, *Sequences, combinatorics, compression, security and transmission*, pages 59–74. Springer–Verlag, 1990.

[81] D. Eppstein, R. Giancarlo, Z. Galil, and F. Italiano. Sparse dynamic programming. preprint.

[82] D. Eppstein, R. Giancarlo, Z. Galil, and F. Italiano. Sparse dynamic programming I: linear cost functions. preprint.

[83] D. Eppstein, R. Giancarlo, Z. Galil, and F. Italiano. Sparse dynamic programming II: convex and concave cost functions. preprint.

[84] B. W. Erickson and P. H. Sellers. *Recognition of patterns in genetic sequences*, pages 55–91. Addison–Wesley, Reading, MA, 1983.

[85] R. Farber, A. Lapedes, and K. Sirotkin. Determination of eukaryotic protein coding regions using neural networks and information theory. Technical Report LA–UR–90–4014, Los Alamos National Laboratory, Theoretical Division, MS B213, Los Alamos National Laboratory, Los Alamos, NM, 87545, 1991.

[86] J. Felsenstein. Numerical methods for inferring evolutionary trees. *The Quarterly Review of Biology*, 57:379–404, 1982.

[87] J. Felsenstein, S. Sawyer, and R. Kochin. An efficient method for matching nucleic acid sequences. *Nucleic Acids Research*, 10:133–139, 1982.

[88] D. F. Feng, M. S. Johnson, and R. F. Doolittle. Alignment of amino acid sequences: Comparison of commonly used methods. *Journal of Molecular Evolution*, 21:112–125, 1985.

[89] J. W. Fickette. Fast optimal alignment. *Nucleic Acids Research*, 12:175–179, 1984.

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

[91] W. M. Fitch and T. F. Smith. Optimal sequence alignments. *Proceedings of the National Academy of Science*, 80:1382–1386, 1983.

[92] W. M. Fitch, T. F. Smith, and W. W. Ralph. Mapping the order of DNA restriction fragments. *Gene*, 22:19–29, 1983.

[93] D. E. Foulser. A linear time algorithm for DNA sequencing. Technical Report YALEU/DCS/RR-812, Yale University, Department of Computer Science, 1990.

[94] M. L. Fredman. Algorithms for computing evolutionary similarity measures with length independent gap penalties. *Bulletin of Mathematical Biology*, 46:553–566, 1984.

[95] T. Friedemann. Alignment of multiple DNA and protein sequence data. *Computer Applications in the Biosciences*, 4:213–214, 1988.

[96] P. Friedland and L. H. Kedes. Discovering the secrets of DNA. *Communications of the ACM*, 28:1164–1186, 1985.

[97] D. J. Galas, M. Eggert, and M. S. Waterman. Rigorous pattern–recognition methods for DNA sequences. *Journal of Molecular Biology*, 186:117–128, 1985.

[98] Z. Galil. Optimal parallel algorithms for string matching. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 240–248, 1984.

[99] Z. Galil. On the exact complexity of string matching: lower bounds. *SIAM Journal of Computing*, 20:1008–1020, 1991.

[100] Z. Galil. A constant time optimal parallel string matching algorithm. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 69–76, Baltimore, MD, 1992. ACM press.

[101] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. Technical Report 110–87, Columbia University Department of Computer Science, 1987.

[102] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 65:107–118, 1989.

[103] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM Journal of Computing*, 19:989–999, 1990.

[104] Z. Galil and K. Park. A linear–time algorithm for concave one–dimensional dynamic programming. *Information Processing Letters*, 33:309–311, 1990.

[105] Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92:49–76, 1992.

[106] Z. Galil and J. Seiferas. Time–space–optimal string matching. *Journal of Computer and System Sciences*, 26:280–294, 1983.

[107] J. Gallant, D. Maier, and J. Storer. On finding minimal length superstrings. *Journal of Computer and System Science*, 20:50–58, 1980.

[108] J. K. Gallant. The complexity of the overlap method for sequencing biopolymers. *Journal of Theoretical Biology*, 101:1–17, 1983.

[109] P. W. Garden. Markov analysis of viral DNA/RNA sequences. *Journal of Theoretical Biology*, 82:679–684, 1980.

[110] M. R. Garey and D. S. Johnson. *Computer and Intractability.* W. H. Freeman and Company, New York, 1979.

[111] J. Garnier and J. M. Levin. The protein structure code: what is its present status? *Computer Applications in the Biosciences*, 7:133–142, 1991.

[112] M. A. Gates. A simple way to look at DNA. *Journal of Theoretical Biology*, 119:319–328, 1986.

[113] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, New York, 1988.

[114] W. Gilbert. Genome sequencing: Creating a new biology for the twenty–first century. *Issues in Science and Technology*, 3:26–35, 1987.

[115] T. R. Gingeras. Computers and DNA sequences: A natural combination. In B. S. Weir, editor, *Statistical analysis of DNA sequence data*, pages 15–43. Marcel–Dekker, New York, 1983.

[116] W. B. Goad. Computational analysis of genetic sequences. *Annual Review of biophysics and biophysical chemistry*, 15:79–95, 1986.

[117] W. B. Goad and M. I. Kanehisa. Pattern recognition in nucleic acid sequences. I. a general method for finding local homologies and symmetries. *Nucleic Acids Research*, 10:247–263, 1982.

[118] G. H. Gonnet and R. A. Baeza-Yates. An analysis of the Karp–Rabin string matching algorithm. *Information Processing Letters*, 34:271–274, 1990.

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

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

[121] O. Gotoh. Consistency of optimal sequence alignments. *Bulletin of Mathematical Biology*, 52:509–525, 1990.

[122] O. Gotoh. Optimal sequence alignment allowing for long gaps. *Bulletin of Mathematical Biology*, 52:359–373, 1990.

[123] O. Gotoh and Y. Tagashira. Sequence search on a supercomputer. *Nucleic Acids Research*, 14:57–64, 1986.

[124] M. Gribskov and M. Devereux. *Sequence analysis primer*. Stockton Press, New York, 1991.

[125] M. Gribskov, M. McLachlan, and D. Eisenberg. Profile analysis: detection of distantly related proteins. *Proceedings of the National Academy of Science*, 84:4355–4358, 1987.

[126] R. Grossi and R. Luccio. Simple and efficient string matching with k mismatches. *Information Processing Letters*, 33:113–120, 1989.

[127] R. A. Grymes, P. Travers, and A. Engelberg. Gel – a computer tool for DNA sequencing projects. *Nucleic Acids Research*, 14:87–99, 1986.

[128] D. Gusfield, G. M. Landau, and B. Schieber. An efficient algorithm for the all pairs suffix–prefix problem. To appear in Sequences: Combinatorics, Compression, Security and Transmission, 1991.

[129] J. E. Haber and D. E. Koshland Jr. An evaluation of the relatedness of proteins based on comparison of amino acid sequences. *Journal of Molecular Biology*, 50:617–639, 1970.

[130] S. Hahn, S. Buratowski, P. A. Sharp, and L. Guarente. Isolation of the gene encoding the yeast TATA binding protein TFIID: A gene identical to the SPT15 suppressor of ty element insertions. *Cell*, 58:1173–1181, 1989.

[131] R. Harr, H. Haggstrom, and P. Gustafsson. Search algorithm for pattern match analysis of nucleic acid sequences. *Nucleic Acids Research*, 11:2943–2957, 1983.

[132] D. S. Hirschberg. A linear space algorithm for computing longest common subsequences. *Communications of the ACM*, 18:341–343, 1975.

[133] D. S. Hirschberg. Fast parallel sorting algorithms. *Communications of the ACM*, 21:657–661, 1978.

[134] J. L. Holloway. An annotated bibliography of algorithms applicable to molecular biology. Technical Report 92–50–01, Oregon State University, Corvallis, OR. 97331, 1992.

[135] L. Hood and L. Smith. Genome sequencing: How to proceed. *Issues in Science and Technology*, 3:36–46, 1987.

[136] M. Horikoshi, C. K. Wang, H. Fujii, J. A. Cromlish, P. A. Weil, and R. G. Roeder. Cloning and structure of a yeast gene encoding a general transcription initiation factor TFIID that binds to the TATA box. *Nature*, 341:299–303, 1989.

[137] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(3):501–506, 1980.

[138] W. J Hsu and M. W. Du. New algorithms for the LCS problem. *Journal of Computer and System Sciences*, 29:133–152, 1984.

[139] X. Huang. A lower bound for the edit–distance problem under an arbitrary cost function. *Information Processing Letters*, 27:319–321, 1988.

[140] X. Huang. A space–efficient parallel sequence comparison algorithm for a message–passing multiprocessor. *International Journal of Parallel Programming*, 18:223–239, 1989.

[141] X. Huang, W. Miller, S. Schwartz, and R. C. Hardison. Parallelization of a local similarity algorithm. *Computer Applications in the Biosciences*, 8(2):155–165, 1992.

[142] A. Hume and D. Sunday. Fast string searching. Technical Report Computing Science Technical Report Number 156, AT&T Bell Labs, 1991.

[143] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[144] O. H. Ibarra, T. Jiang, and H. Wang. String editing on a one–way linear array of finite state machines. *IEEE Transactions on Computers*, 41(1):112–118, 1992.

[145] O. H. Ibarra, T. Pong, and S. M. Sohn. Hypercube algorithms for some string comparison problems. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume 3, pages 190–193, 1988.

[146] O. H. Ibarra, T. Pong, and S. M. Sohn. String processing on the hypercube. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 38:160–164, 1990.

[147] M. E. Isenman and D. E. Shasha. Performance and architectural issues for string matching. *IEEE Transactions on Computers*, 39:238–250, 1990.

[148] M. S. Johnson and R. F. Doolittle. A method for the simultaneous alignment of three or more amino acid sequences. *Journal of Molecular Evolution*, 23:267–278, 1986.

[149] R. E. Johnston, J. M. Mackenzie Jr., and W. G. Dougherty. Assembly of overlapping DNA sequences by a program written in BASIC for 64k CP/M and MS-DOS IBM–compatible microcomputers. *Nucleic Acids Research*, 14:517–527, 1986.

[150] L. P. Jones. Portrep: A portable repeated string finder. *Software Practice and Experience*, 19:63–77, 1980.

[151] J. R. Jungck and R. M. Friedman. Mathematical tools for molecular genetics data: An annotated bibliography. *Bulletin of Mathematical Biology*, 46:699–744, 1984.

[152] M. I. Kanehisa and W. B. Goad. Pattern recognition in nucleic acid sequences. II. an efficient method for finding locally stable secondary structures. *Nucleic Acids Research*, 10:265–278, 1982.

[153] S. Karlin and S.F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Science*, 87:2264–2268, 1990.

[154] S. Karlin and G. Ghandour. Comparative statistics for DNA and protein sequences: single sequence statistics. *Proceedings of the National Academy of Science*, 82:5800–5804, 1985.

[155] S. Karlin, G. Ghandour, F. Ost, S. Tavare, and L. J. Korn. New approaches for computer analysis of nucleic acid sequences. *Proceedings of the National Academy of Science*, 80:5660–5664, 1983.

[156] S. Karlin, M. Morris, G. Ghandour, and M. Leung. Algorithms for identifying local molecular sequence features. *Computer Applications in the Biosciences*, 4(1):41–51, 1988.

[157] S. Karlin, M. Morris, G. Ghandour, and M. Y. Leung. Efficient algorithms for molecular sequence analysis. *Proceedings of the National Academy of Science*, 85:841–845, 1988.

[158] R. M. Karp and M. O. Rabin. Efficient randomized pattern–matching algorithms. *IBM Journal of Research and Development*, 32:249–260, 1987.

[159] J. D. Kececioglu. *Exact and Approximation algorithms for DNA sequence reconstruction*. PhD thesis, The University of Arizona, Tucson, Arizona, 1991.

[160] D. E. Knuth. *The art of computer programming: fundamental algorithms*, volume 1. Addison–Wesley, 1973.

[161] D. E. Knuth. *The art of computer programming: searching and sorting*, volume 3. Addison–Wesley, 1973.

[162] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:323–350, 1977.

[163] M. Krawczak. Algorithms for the restriction–site mapping of DNA molecules. *Proceedings of the National Academy of Science*, 85:7298–7301, 1988.

[164] S. Kuo and G. R. Cross. An improved algorithm to find the length of the longest common subsequence of two strings. *SIGIR Forum*, 23:89–99, 1989.

[165] G. M. Landau and U. Vishkin. Efficient string matching in the presence of errors. In *IEEE Symposium on the Foundations of Computer Science*, pages 126–136, 1985.

[166] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[167] G. M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 220–230, 1986.

[168] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989.

[169] G. M. Landau, U. Vishkin, and R. Nussinov. An efficient string matching algorithm with k differences for nucleotide and amino acid sequences. *Nucleic Acids Research*, 14:31–46, 1986.

[170] G. M. Landau, U. Vishkin, and R. Nussinov. An efficient string matching algorithm with k substitutions for nucleotide and amino acid sequences. *Journal of Theoretical Biology*, 126:483–490, 1987.

[171] G. M. Landau, U. Vishkin, and R. Nussinov. Fast alignment of DNA and protein sequences. *Methods in Enzymology*, 183:487–502, 1990.

[172] A. M. Landraud, J. F. Avril, and P. Chretienne. An algorithm for finding a common structure shared by a family of strings. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:890–895, 1989.

[173] L. L. Larmore. On–line dynamic programming with applications to the prediction of RNA secondary structure. *Journal of Algorithms*, 12:490–515, 1991.

[174] R. H. Lathrop, T. A. Webster, and T. F. Smith. Ariadne: Pattern–directed inference and hierarchical abstraction in protein structure recognition. *Communications of the ACM*, 30:909–921, 1987.

[175] C. B. Lawrence. Use of homology domains in sequence similarity detection. *Methods in Enzymology*, 183:133–146, 1990.

[176] C. B. Lawrence, D. A. Goldman, and R. T. Hood. Optimized homology searches of the gene and protein sequence data banks. *Bulletin of Mathematical Biology*, 48:569–583, 1986.

[177] S. Y. Le, R. Nussinov, and J. V. Maizel. Tree graphs of RNA secondary structures and their comparisons. *Computers and Biomedical Research*, 22:461–473, 1989.

[178] T. Lecroq. A variation on the Boyer–Moore algorithm. *Theoretical Computer Science*, 92:119–144, 1992.

[179] A. M. Lesk, editor. *Computational Molecular Biology.* Oxford, New York, 1988.

[180] B. M. Lewin. *Genes II.* John Wiley & Sons, New York, 1985.

[181] M. Li. Towards a DNA sequencing theory. In *IEEE Symposium on the Foundations of Computer Science*, pages 125–134, 1990.

[182] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A tool for multiple sequence alignment. *Proceedings of the National Academy of Science*, 86:4412–4415, 1989.

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

[184] R. Lowrance and R. A. Wagner. An extension of the string to string correction problem. *Journal of the Association for Computing Machinery*, 23(2):177–183, 1975.

[185] M. Maes. On a cyclic string–to–string correction problem. *Information Processing Letters*, pages 73–78, 1990.

[186] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the Association for Computing Machinery*, 25:322–336, 1978.

[187] M. G. Main and R. J. Lorentz. An $\theta(n \log n)$ algorithm for finding all repetitions in a string. Technical Report CU–CS–241–82, University of Colorado, 1982.

[188] U. Manber and R. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 37:133–136, 1991.

[189] U. Manber and G. Myers. Suffix arrays: A new method for on–line string searches. In *Proceedings of the First Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 319–327. SIAM, 1990.

[190] H. M. Martinez. An efficient method for finding repeats in molecular sequences. *Nucleic Acids Research*, 11:4629–4634, 1983.

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

[192] A. M. Masreliez and J. L. Holloway. Calculation of the minimal–cost path seperating a DNA strand and a protein sequence. Class project, 1992.

[193] E. M. McCreight. A space–economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23:262–272, 1976.

[194] P. L. Miller, P. M. Nadkarni, and N. M. Carriero. Parallel computation and FASTA: confronting the problem of parallel database search for a fast sequence comparison algorithm. *Computer Applications in the Biosciences*, 7(1):71–78, 1991.

[195] W. Miller, J. Barr, and K. E. Rudd. Improved algorithms for searching restriction maps. *Computer Applications in the Biosciences*, 7:447–456, 1991.

[196] W. Miller and E. W. Myers. A file comparison program. *Software–Practice and Experience*, 15:1025–1040, 1985.

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

[198] A. D. Milosavljevic. *Categorization of macromolecular sequences by minimal length encoding.* PhD thesis, University of California at Santa Cruz, 1991.

[199] J. F. Moore and C. Burks. The genbank nucleic acid data bank. *Focus*, 11(4), 1991.

[200] R. Mott. Maximum–likelihood estimation of the statistical distribution of Smith–Waterman local sequence similarity scores. *Bulletin of Mathematical Biology*, 54:59–75, 1992.

[201] R. F. Mott, T. B. L. Kirkwood, and R. N. Curnow. An accurate approximation to the distribution of the length of the longest matching word between two random DNA sequences. *Bulletin of Mathematical Biology*, 52:773–784, 1990.

[202] A. Mukherjee. Hardware algorithms for determining similarity between two strings. *IEEE Transactions on Computers*, 38:600–603, 1989.

[203] M. Murata, J. S. Richardson, and J. L. Sussman. Simultaneous comparison of three protein sequences. *Proceedings of the National Academy of Science*, 82:3073–3077, 1985.

[204] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–256, 1986.

[205] E. W. Myers. Incremental alignment algorithms and their applications. *SIAM Journal of Computing*, 1989. to be published.

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

[207] E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51:5–37, 1989.

[208] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the Association for Computing Machinery*, 29:642–667, 1982.

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

[210] R. Nussinov. Efficient algorithms for searching for exact repetition of nucleotide sequences. *Journal of Molecular Evolution*, 19:283–285, 1983.

[211] R. Nussinov. Theoretical molecular biology: prospectives and perspectives. *Journal of Theoretical Biology*, 125:219–235, 1987.

[212] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman. Algorithm for loop matchings. *SIAM Journal of Applied Mathematics*, 35:68–82, 1978.

[213] S. G. Oliver and J. G. Sgouros et. al. The complete DNA sequence of yeast chromosome III. *Nature*, 357:38–46, 7 May 1992.

[214] B. C. Orcutt and W. C. Barker. Searching the protein sequence database. *Bulletin of Mathematical Biology*, 46:545–552, 1984.

[215] W. R. Pearson. Rapid and sensitive sequence comparison with fastp and fasta. *Methods in Enzymology*, 188:63–98, 1990.

[216] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Science*, 85:2444–2448, 1988.

[217] H. Peltola, H. Soderlund, J. Tarhio, and E. Ukkonen. Algorithms for some string matching problems arising in molecular genetics. In *Information Processing 83*, pages 53–64, 1983.

[218] H. Peltola, H. Soderlund, and E. Ukkonen. SEQAID: a DNA sequence assembling program based on a mathematical model. *Nucleic Acids Research*, 12:307–321, 1984.

[219] H. Peltola, H. Soderlund, and E. Ukkonen. Algorithms for the search of amino acid patterns in nucleic acid sequences. *Nucleic Acids Research*, 14:99–107, 1986.

[220] M. Perry, G. H. Thomsen, and R. G. Roeder. Genomic organization and nucleotide sequence of two distinct histone gene clusters from *xenopus laevis*. *Journal of Molecular Biology*, 185:479–499, 1985.

[221] J. J. Pollock. Spelling error detection and correction by computer: some notes and a bibliography. *Journal of documentation*, 38(4):282–291, 1982.

[222] S. Pramanik and C. T. King. A hardware pattern matching algorithm on a data flow. *The Computer Journal*, 38:264–269, 1985.

[223] R. W. Quong. Fast average–case pattern matching by multiplexing sparse tables. *Theoretical Computer Science*, 92:165–179, 1992.

[224] S. Ranka and T. Heywood. Two–dimensional pattern matching with k mismatches. *Pattern Recognition*, 24:31–40, 1991.

[225] J. H. Reif. An optimal parallel algorithm for integer sorting. In *IEEE Symposium on the Foundations of Computer Science*, pages 496–504, 1985.

[226] I. Rinsma, M. Hendy, and D. Penny. Distribution of the number of matches between nucleotide sequences. *Bulletin of Mathematical Biology*, 52:349–358, 1990.

[227] L. Roberts. New chip may speed genome analysis. *Science*, 244:655–656, 1989.

[228] L. Roberts. The worm project. *Science*, 248:655–656, 1989.

[229] K. H. Rosen. *Elementary Number Theory and Its Applications.* Addison–Wesley, Reading, MA, 1984.

[230] F. Sanger, A. R. Coulson, B. G. Barrell, A. J. H. Smith, and B. A. Roe. Cloning in single–stranded bacteriophage as an aid to rapid DNA sequencing. *Journal of Molecular Biology*, pages 161–178, 1980.

[231] D. Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Science*, 69:4–6, 1972.

[232] D. Sankoff and R. J. Cedergren. A test for nucleotide sequence homology. *Journal of Molecular Biology*, 77:159–164, 1973.

[233] D. Sankoff, R. J. Cedergren, and Y. Abel. Genomic divergence through gene rearrangement. *Methods in Enzymology*, 183:428–438, 1990.

[234] D. Sankoff and M. Goldstein. Probabilistic models of genome shuffling. *Bulletin of Mathematical Biology*, 51:117–124, 1989.

[235] D. Sankoff and J. B. Kruskal. *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison.* Addison–Wesley, Reading, MA, 1983.

[236] R. Schaback. On the expected sublinearity of the boyer–moore algorithm. *SIAM Journal of Computing*, 17:648–658, 1988.

[237] M. C. Schmidt, C. Kao, R. Pei, and A. J. Berk. Yeast TATA-box transcription factor gene. *Proceedings of the National Academy of Science*, 86:7785–7789, 1989.

[238] M. Schoniger and M. S. Waterman. A local algorithm for DNA sequence alignment with inversions. *Bulletin of Mathematical Biology*, 54(4):521–536, 1992.

[239] R. Sedgewick. *Algorithms.* Addison–Wesley, Reading, MA, 1983.

[240] P. H. Sellers. An algorithm for the distance between two finite sequences. *Journal of Combinatorial Theory (A)*, 16:253–258, 1974.

[241] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal of Applied Mathematics*, 26:787–793, 1974.

[242] P. H. Sellers. Pattern recognition in genetic sequences. *Proceedings of the National Academy of Science*, 76:3041–3041, 1979.

[243] P. H. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.

[244] P. H. Sellers. Pattern recognition in genetic sequences by mismatch density. *Bulletin of Mathematical Biology*, 46:501–514, 1984.

[245] D. Sellos, S. A. Krawetz, and G. H. Dixon. Organization and complete nucleotide sequence of the core–histone–gene cluster of the annelid *platynereis dumerilii*. *European Journal of Biochemistry*, 190:21–29, 1990.

[246] M. B. Shapiro. An algorithm for reconstructing protein and RNA sequences. *Journal of the Association for Computing Machinery*, 14:720–731, 1967.

[247] B. D. Silverman and R. Linsker. A measure of DNA periodicity. *Journal of Theoretical Biology*, 118:295–300, 1986.

[248] A. K. Singh and R. Overbeek. Derivation of efficient parallel programs: An example from genetic sequence analysis. *International Journal of Parallel Programming*, 18:447–484, 1989.

[249] D. F. Sittig, D. Foulser, N. Carriero, G. McCorkle, and P. L. Miller. A parallel computing approach to genetic sequence comparison: The master–worker paradigm with interworker communication. *Computers and Biomedical Research*, 24:152–169, 1991.

[250] J. K. Smith. Seqwarp: A low–cost linear systolic array for biological sequence comparison. Master's thesis, The University of Utah, 1991.

[251] P. D. Smith. Experiments with a very fast substring search algorithm. *Software–Practice and Experience*, 21:1065–1074, 1991.

[252] R. F. Smith and T. S. Smith. Automatic generation of primary sequence patterns from sets of related protein sequences. *Proceedings of the National Academy of Science*, 87:118–122, 1990.

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

[254] E. Sobel and H. M. Martinez. A multiple sequence alignment program. *Nucleic Acids Research*, 14:363–374, 1986.

[255] J. L. Spouge. Improving sequence–matching algorithms by working from both ends. *Journal of Molecular Biology*, 181:137–138, 1985.

[256] J. L. Spouge. Fast optimal alignment. *Computer Applications in the Biosciences*, 7(1):1–7, 1991.

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

[258] P. Stolorz, A. Lapedes, and Y. Xia. Predicting protein secondary structure using neural net and statistical methods. Technical Report LA–UR–91–15, Los Alamos National Laboratory, Theoretical Division, MS B213, Los Alamos National Laboratory, Los Alamos, NM, 87545, 1991.

[259] D. Straus and W. Gilbert. Genetic engineering in the precambrian: Structure of the chicken triosephosphate isomerase gene. *Molecular and Cellular Biology*, 5(12):3497–3506, 1985.

[260] S. Subbiah and S. C. Harrison. A method for multiple sequence alignment with gaps. *Journal of Molecular Biology*, 209:539–548, 1989.

[261] D. M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33:132–142, 1990.

[262] J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57:131–145, 1988.

[263] P. Taylor, P. Rosenberg, and M. G. Samsonova. A new method for finding long consensus patterns in nucleic acid sequences. *Computer Applications in the Biosciences*, 7:495–500, 1991.

[264] N. Tewari and M. D. Wagh. Bit-sequential array for pattern matching. *Proceedings of the IEEE*, 74:1465–1466, 1986.

[265] J. L. Thorne, H. Kishino, and J. Felsenstein. Inching toward reality: An improved likelihood model of sequence evolution. submitted to the Journal of Molecular Evolution.

[266] J. L. Thorne, H. Kishino, and J. Felsenstein. An evolutionary model for maximum likelihood alignment of DNA sequences. *Journal of Molecular Evolution*, 33:114–124, 1991.

[267] W. F. Tichy. The string to string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[268] D. C. Torney, C. Burks, D. Davison, and K. M. Sirotkin. Computation of d^2: A measure of sequence dissimilarity. In G. Bell and R. Marr, editors, *Computers and DNA*, pages 109–125, New York, 1990. Sante Fe Institute studies in the sciences of complexity, vol. VII, Addison–Wesley.

[269] J. Turner. Approximation algorithms for the shortest common superstring problem. *Information and Computation*, 83:1–20, 1989.

[270] E. C. Tyler, M. R. Horton, and P. R. Krause. A review of algorithms for molecular sequence comparison. *Computers and Biomedical Research*, 24:72–96, 1991.

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

[272] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[273] E. Ukkonen. A linear–time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5:313–323, 1990.

[274] S. M. Ulam. Some ideas and prospects in biomathematics. In M. F. Morales, editor, *Annual Review of Biophysics and Bioengineering.* Annual reviews Inc., Palo Alto, CA, 1972.

[275] U.S. Department of Health and Human Services and U.S. Department of Energy. Understanding our genetic inheritance. the u.s. human genome project: The frist five years, fy 1991–1995. Technical Report NIH 90–1590, U.S. Department of Health and Human Services and U.S. Department of Energy, 1990.

[276] M. Veldhorst. Parallel dynamic programming algorithms. In *CONPAR 86. Conference on Algorithms and Hardware for Parallel Processing*, pages 393–402, Berlin, 1986. Springer–Verlag.

[277] U. Vishkin. Optimal parallel pattern matching in strings. In *Proceedings of the* 12*th* *ICALP, Lecture Notes in CS 194*, pages 497–508, New York, 1985. Springer–Verlag.

[278] G. von Heijne. *Sequence analysis in molecular biology.* Academic Press, New York, 1987.

[279] R. A. Wagner and M. J. Fischer. The string–to–string correction problem. *Journal of the Association for Computing Machinery*, 21:168–173, 1974.

[280] M. S. Waterman. Sequence alignments in the neighborhood of the optimum with general application to dynamic programming. *Proceedings of the National Academy of Science*, 80:3123–3124, 1983.

[281] M. S. Waterman. Efficient sequence alignment algorithms. *Journal of Theoretical Biology*, 108:333–337, 1984.

[282] M. S. Waterman. General methods of sequence comparison. *Bulletin of Mathematical Biology*, 46:473–500, 1984.

[283] M. S. Waterman. Multiple sequence alignment by consensus. *Nucleic Acids Research*, 14:9095–9102, 1986.

[284] M. S. Waterman. Computer analysis of nucleic acid sequences. *Methods in Enzymology*, 164:765–795, 1988.

[285] M. S. Waterman. Forward to 1989 special issue on molecular sequence analysis. *Bulletin of Mathematical Biology*, 51:1–4, 1989.

[286] M. S. Waterman, R. Arratia, and D. J. Galas. Pattern recognition in several sequences: consensus and alignment. *Bulletin of Mathematical Biology*, 46:515–527, 1984.

[287] M. S. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA–rRNA comparisons. *Journal of Molecular Biology*, 197:723–728, 1987.

[288] M. S. Waterman and J. R. Griggs. Interval graphs and maps of DNA. *Bulletin of Mathematical Biology*, 48:189–195, 1986.

[289] M. S. Waterman, T. F. Smith, and W. A. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20:367–387, 1976.

[290] P. Weiner. Linear pattern matching algorithms. In *Conference Record, IEEE* 14^{th} *Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[291] W. J. Wilbur. On the PAM matrix model of protein evolution. *Molecular and Biological Evolution*, 2:434–447, 1985.

[292] W. J. Wilbur and D. J. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Science*, 80:726–730, 1983.

[293] W. J. Wilbur and D. J. Lipman. The context dependent comparison of biological sequences. *SIAM Journal of Applied Mathematics*, 44:557–567, 1984.

[294] C. K. Wong and A. K. Chandra. Bounds for the string editing problem. *Journal of the Association for Computing Machinery*, 23:13–16, 1976.

[295] S. Wu, U. Manber, G. Myers, and W. Miller. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35:317–323, 1990.

[296] M. Zuker. On finding all suboptimal foldings of an RNA molecule. *Science*, 244:48–52, 1989.

[297] M. Zuker and R. L. Somorjai. The alignment of protein structures in three dimensions. *Bulletin of Mathematical Biology*, 51:55–78, 1989.

[298] M. Zuker and P. Stiegler. Optimal computer folding of large rna sequence using thermodynamics and auxililary information. *Nucleic Acids Research*, 9:133–148, 1981.

Appendix

Appendix A

Glossary

Base One of four molecules that make up the backbone of DNA. They are adenine and guanine (the purines) and cytosine and thymine (the pyrimidines), often represented as A, G, C, and T respectively.

Cloning The process of inserting a segment of foreign DNA into a plasmid and then into a host organism, usually for the purpose of highly expressing (making many copies of) the segment of foreign DNA.

Codon Three nucleotides that represent an amino acid or a stop signal.

Dicodon Two adjacent codons.

DNase I An enzyme that randomly cleaves DNA, independent of the position or composition of the site being cleaved.

Eukaryote Organism that have cells with nuclear membranes.

Exon Any segment of a gene that is represented in the final RNA copy of the gene.

Fingerprint See motif.

GenBank Database of DNA and RNA sequences.

Homologous A group of organisms or molecules are homologous if they have evolved from a common ancestor. Homology is frequently inferred from similarity.

Intron A segment of a gene that is transcribed into the initial copy of the RNA, but removed from the final RNA copy by splicing together the exons on either side of the intron.

Mapping, Genetic Determining the order of genetic markers by data gathered from the recombination of the DNA. The markers are frequently genes or alleles of the same gene. The unit of measure is the centimorgan. A distance of one centimorgan indicates that two markers are separated by recombination one percent of the time when the traits are passed from parent to children.

Mapping, Physical Determining the physical position of regions of interest in the DNA such as genes. The unit of measure if frequently the number of base pairs between points of interest.

Messenger RNA (mRNA) A transcription of the DNA used to move the specification for one or more proteins from the nucleus to the cytoplasm.

Motif A pattern of nucleic acids or amino acids that is associated with a specific structure or function of the molecule.

Nucleotide A subunit of DNA composed of a base, a sugar, and at least one phosphate group.

Phylogenetic tree A bifurcating tree that describes the evolutionary history of a group of organisms. In some phylogenetic trees the distance from a branch point represents the time since the two groups diverged, in other phylogenetic trees only the branching order is significant.

Polymerase Chain Reaction (PCR) A method of amplifying specific regions of DNA. Sequence specific primers are created that bind to the ends of the region of DNA to be amplified. By repeating a cycle that causes the the region of DNA between the primers to be duplicated, the number of copies of the region of DNA that is of interest increases exponentially.

Primer A short sequence that pairs with a strand of DNA to initiate the synthesis of a copy of the DNA.

Reading Frame One of the three possible ways of reading a nucleotide sequence as a series of triplets representing amino acids. Each reading frame will produce a different sequence of amino acids from the same nucleotide sequence.

Restriction Enzyme A molecule that recognizes a short specific sequence of DNA and cleaves the DNA at that or nearby site. Also referred to as a restriction endonuclease.

Ribosomal RNA (rRNA) RNA that is an integral part of ribosomes, particles that translate Messenger RNA to produce a protein.

Sequence A string of characters representing the nucleic acids or amino acids of a strand of DNA, RNA, or protein.

Sequencing Gel A material that when an electric current is run through, allows DNA fragments to travel at different rates depending on the length of the fragment. Used to determine the order of bases in DNA.

Sequencing The process of determining the order of bases in a DNA or RNA sequence.

Shotgun sequencing A method of determining the order of bases in a DNA sequence. The DNA of interest is fragmented into random subsequences, the subsequences are sequenced, and the overlaps in the subsequences are used to reconstruct the original sequence.

Structure, Primary The order of the nucleotides in DNA and RNA, or the order of the amino acids in a protein.

Structure, Secondary In RNA, the pattern of stems and loops (see Figure 1.3). In protein, the basic structures such as helix, β sheet, and turns.

Structure, Tertiary In protein, the three dimensional structure of the molecule.

Template See motif.

Transfer RNA (tRNA) A small RNA molecule that binds an amino acid on one end and recognizes an mRNA triplet on the other end. Used to transport amino acids to construct a protein.

Watson–Crick Base Pairing In DNA, adenine will pair with thymine and guanine will pair with cytosine to form base pairs.