Efficient Homology Search for Genomic Sequence Databases

A thesis submitted for the degree of
Doctor of Philosophy

Michael Cameron B.CompSci. (Hons.),
School of Computer Science and Information Technology,
Science, Engineering, and Technology Portfolio,
RMIT University,
Melbourne, Victoria, Australia.

November 27, 2006

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Michael Maxwell Cameron
School of Computer Science and Information Technology
RMIT University
November 27, 2006

Acknowledgments

My deepest thanks go to Dr Hugh Williams for his guidance, advice and enthusiasm throughout my candidature. This research experience would certainly not have been as enjoyable or as rewarding without your support, and I greatly appreciate that you continued to advise me after your departure from RMIT.

I am indebted to many others, who helped me to complete my PhD. I am grateful for the guidance, helpful discussions and proof reading of my secondary supervisors Dr Andrew Turpin and Dr Adam Cannane. I would also like to thank Dr Peter Smooker for valuable discussions and the chance to give lectures on topics relating to genomic search. I am especially thankful to Dr Stephen Altschul for giving me the opportunity to visit the NCBI, which was most rewarding, and to Dr Tom Madden and Dr David Lipman for providing BLAST usage statistics.

I would also like to thank members of the Search Engine Group at RMIT for their support, and providing an enjoyable atmosphere to conduct my research. In particular, I thank fellow student Yaniv Bernstein for a very rewarding and enjoyable collaboration that was certainly the highlight of this project. I also thank Lily Chen for valuable discussions and Dr Bodo von Billerbeck for welcoming me into the group.

Finally, I am most grateful to my gorgeous wife, Michelle Chow, for supporting me, proof reading my work and educating me on the fundamentals of biochemistry and molecular biology, and to my loving parents Sonya and Max Cameron.

This project was supported by an Australian Postgraduate Award, a scholarship from the School of Computer Science and Information Technology, and the Australian Research Council.

Credits

Portions of the material in this thesis have previously appeared in the following publications:

- M. Cameron, H. E. Williams, and A. Cannane. Improved gapped alignment in BLAST. *IEEE Transactions on Computational Biology and Bioinformatics*, 1(3):116–129, 2004.
- M. Cameron, H. E. Williams, and A. Cannane. A deterministic finite automaton for faster protein hit detection in BLAST. *Journal of Computational Biology*, 13(4): 965–978, 2006.
- M. Cameron, Y. Bernstein, and H. E. Williams. Clustering near-identical sequences for fast homology search. In A. Apostolico, C. Guerra, S. Istrail, P. A. Pevzner, and M. S. Waterman, editors, *Proceedings of the International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 175–189, Venice, Italy, 2006. Springer.
- Y. Bernstein and M. Cameron. Fast discovery of similar sequences in large genomic collections. In M. Lalmas, A. MacFarlane, S. M. Rüger, A. Tombros, T. Tsikrika, and A. Yavlinsky, editors, *28th European Conference on IR Research, ECIR 2006, London, UK, April 10-12, 2006, Proceedings*, pages 432–443, London, UK, 2006. Springer.
- M. Cameron and H. E. Williams. Comparing compressed sequences for faster nucleotide blast searches. *IEEE Transactions on Computational Biology and Bioinformatics*, 2006. To appear.
- M. Cameron, Y. Bernstein, and H. E. Williams. Clustered sequence representation for fast homology search. *Journal of Computational Biology*, 2006. To appear.

Contents

List of Figures

List of Tables

Summary

Genomic search tools can provide valuable insights into the chemical structure, evolutionary origin and biochemical function of genetic material. A homology search algorithm compares a protein or nucleotide query sequence to each entry in a large sequence database and reports alignments with highly similar sequences. The exponential growth of public data banks such as GenBank has necessitated the development of fast, heuristic approaches to homology search. The versatile and popular BLAST algorithm, developed by researchers at the US National Center for Biotechnology Information (NCBI), uses a four-stage heuristic approach to efficiently search large collections for analogous sequences while retaining a high degree of accuracy. Despite an abundance of alternative approaches to homology search, BLAST remains the only method to offer fast, sensitive search of large genomic collections on modern desktop hardware. As a result, the tool has found widespread use with millions of queries posed each day. A significant investment of computing resources is required to process this large volume of genomic searches and a cluster of over 200 workstations is employed by the NCBI to handle queries posed through the organisation's website. As the growth of sequence databases continues to outpace improvements in modern hardware, BLAST searches are becoming slower each year and novel, faster methods for sequence comparison are required.

In this thesis we propose new techniques for fast yet accurate homology search that result in significantly faster BLAST searches. First, we describe improvements to the final, gapped alignment stages where the query and sequences from the collection are aligned to provide a fine-grain measure of similarity. We describe three new methods for aligning sequences that roughly halve the time required to perform this computationally expensive stage.

Next, we investigate improvements to the first stage of search, where short regions of similarity between a pair of sequences are identified. We propose a novel deterministic finite automaton data structure that is significantly smaller than the codeword lookup table employed by NCBI-BLAST, resulting in improved cache performance and faster search times.

We also discuss fast methods for nucleotide sequence comparison. We describe novel approaches for processing sequences that are compressed using the *byte packed* format already utilised by BLAST, where four nucleotide bases from a strand of DNA are stored in a single byte. Rather than decompress sequences to perform pairwise comparisons, our innovations permit sequences to be processed in their compressed form, four bases at a time. Our techniques roughly halve average query evaluation times for nucleotide searches with no effect on the sensitivity of BLAST.

Finally, we present a new scheme for managing the high degree of redundancy that is prevalent in genomic collections. Near-duplicate entries in sequence data banks are highly detrimental to retrieval performance, however existing methods for managing redundancy are both slow, requiring almost ten hours to process the GenBank database, and crude, because they simply purge highly-similar sequences to reduce the level of internal redundancy. We describe a new approach for identifying near-duplicate entries that is roughly six times faster than the most successful existing approaches, and a novel approach to managing redundancy that reduces collection size and search times but still provides accurate and comprehensive search results.

Our improvements to BLAST have been integrated into our own version of the tool. We find that our innovations more than halve average search times for nucleotide and protein searches, and have no significant effect on search accuracy. Given the enormous popularity of BLAST, this represents a very significant advance in computational methods to aid life science research.

Chapter 1

Introduction

Modern, high-throughput sequencing technologies and initiatives such as the Human Genome Project have resulted in an explosion of genomic data in recent years. As a result, publicly available data banks such as GenBank [Benson et al., 2005], EMBL [Kanz et al., 2005] and DDBJ [Miyazaki et al., 2004] contain over 50 gigabytes of uncompressed sequence data and continue to exhibit exponential growth. Mining such collections can provide valuable insights into the characteristics of proteins and DNA and lead to important discoveries in the fields of biology, medicine and agriculture. For example, an homology search is often the first step towards understanding the function of a newly sequenced protein or strand of DNA. Homology search tools compare a query sequence to every sequence in a collection with the aim of identifying highly-similar and possibly related entries. A poorly-annotated or unknown sequence may be used to query a large database of biological sequences with the aim of identifying a related and well-annotated sequence in the collection. Similar sequences often share a common three-dimensional structure, perform the same role in an organism and have a common evolutionary origin. As a result, homology search can provide an insight into the structure, function and evolutionary origin of a newly sequenced or poorly annotated protein or strand of DNA. This can lead to discoveries in the fields of biochemistry and molecular biology, resulting in new methods for crop production, and new medicines to combat diseases such as cancer and HIV infection.

The degree of similarity between biological sequences is measured using a sequence alignment algorithm, a comparison method that models the process through which proteins and DNA evolve and measures the number of elementary changes required to transform one sequence into another. The Smith-Waterman algorithm [1981] is a dynamic programming

technique commonly employed to compute the optimal alignment between two sequences. However, a database search using the exhaustive and highly-sensitive Smith-Waterman approach is not feasible for any collection of significant size — our tests show that a search of GenBank can take hours or even days on a modern workstation. As a result, several heuristic approaches have been developed that sacrifice a small amount of accuracy for substantially faster search. The most successful heuristic approach is BLAST, which was first described by Altschul et al. [1990] and later refined by Altschul et al. [1997].

BLAST provides fast yet sensitive search of large genomic databases and is the most popular homology search tool in widespread use. The BLAST tool is ubiquitous with copies of the software installed in almost every medium- to large-scale molecular biology research facility, and it has been widely-adapted to different hardware, operating systems, and tasks. The online interface to the tool at the popular National Center for Biotechnology Information (NCBI) website[1] is used to evaluate over 120,000 queries each day [McGinnis and Madden, 2004], and the 1997 paper describing the algorithm [Altschul et al., 1997] has been cited more than 10,000 times[2]. BLAST remains the most successful approach to homology search despite a plethora of more recent methods such as index-based approaches [Kent, 2002; Williams and Zobel, 2002] and discontinuous seeds [Ma et al., 2002; Li et al., 2004].

BLAST uses a four-stage approach to search that efficiently identifies collection sequences with a high degree of similarity to the query [Altschul et al., 1997]. A filtering strategy is employed, where each stage involves more computation per collection sequence than the previous, but fewer collection sequences are considered in later stages. In the first stage, BLAST identifies hits: short, fixed-length regions of similarity between the query sequence and sequences in the collection. A compact lookup structure, which is derived from the query sequence, is used to scan the collection and identity matching regions efficiently. In the second stage, an *ungapped alignment* is performed, where the regions from the query and collection sequences immediately surrounding a hit are aligned. An ungapped alignment provides a coarse measure of similarity and does not consider the more complex gap events. Collection sequences that produce a high-scoring ungapped alignment are passed on to the third stage, where a more fine-grain *gapped alignment* is performed. The computationally expensive gapped alignment stage compares the query to the remaining collection sequences using a dynamic programming algorithm that is similar to Smith-Waterman [Zhang et al., 1998a]. Statistically significant alignments are displayed to the user in the fourth stage.

[1] See `http://www.ncbi.nlm.nih.gov/`

[2] See: `http://scholar.google.com/`

Despite the success of the BLAST approach, it is not perfect. Searches with BLAST are becoming slower each year because of the well-known exponential growth in genomic collections, and despite improvements in hardware [Chen, 2004; Attwood and Higgs, 2004]. Chen [2004] reports that searches of the entire GenBank nucleotide database are becoming roughly 64% slower each year. With millions of queries posed each day, BLAST searches represent a very significant investment in computing resources; a cluster of 200 workstations is required just to handle the volume of queries posed through the NCBI website [McGinnis and Madden, 2004]. It is therefore imperative that the fundamental algorithms in BLAST continue to be improved, and that new heuristics to improve search speed are discovered.

Despite the widespread appeal of BLAST, there has been little work since 1997 — other than that of the original authors — into improving the fundamental algorithmic steps that it uses to efficiently and accurately search genomic collections. We believe that further improvements to the algorithm have been hindered by three crucial factors. First, the algorithm has already been carefully refined and optimised by a large team of developers at the NCBI over many years, leaving few obvious yet significant improvements. Second, BLAST is an incredibly complex algorithm that is difficult to understand in detail, let alone improve upon. A detailed description of the underlying algorithm has never been published, and we have discovered that important aspects of the BLAST approach have not been described at all in existing literature. Further, no in-depth analysis of the performance characteristics of each stage has been carried out. Finally, we believe that many researchers are adverse to improving existing approaches, and prefer to develop their own, novel algorithms for homology search that often fail to find widespread use for one reason or another.

In this thesis we present a range of improvements to the fundamental BLAST algorithm. Our innovations include new methods for aligning sequences, additional stages for filtering collection sequences, novel data structures and algorithms for faster comparison, and new methods for representing and storing sequence data. When combined, our improvements increase the search speed of BLAST by a factor of two. Importantly, none of our improvements have a significant effect on overall search accuracy; we believe that faster approaches to homology search will be welcome by biologists, but not at the expense of accuracy. Indeed, several novel search strategies have been proposed that provide significantly faster search but reduced search accuracy, none of which have achieved the same level of success as BLAST [Zhang et al., 2000; Ning et al., 2001; Kent, 2002]. Many of the concepts introduced in this thesis are applicable to homology search in general, and can be trivially adapted for use with other genomic search tools; we have chosen BLAST as our testbed due to its popu-

larity and wide-spread use. Our improvements to BLAST are embodied in a new, open-source version of the tool that is freely available for download at `http://www.fsa-blast.org/`.

1.1 Thesis Structure

We begin our motivation for fast and accurate genomic search techniques with an introduction to existing sequence comparison and search methods in Chapters 2 and 3. In Chapter 2, we provide an overview of genomics, proteomics, and sequence comparison methods that are widely used in computational biology. We describe proteins and DNA, and the roles that these two molecules play in living organisms. A vast quantity of protein and DNA sequence data has been made available through public data banks such as GenBank, and the analysis of such data can lead to important breakthroughs and discoveries by biologists.

The degree of similarity between a pair of sequences is commonly measured using a pairwise sequence alignment algorithm that computes the minimum number of elementary changes or mutations required to transform one sequence into another. Dynamic programming techniques such as the Smith-Waterman algorithm [1981] are used to compute the optimal or highest-scoring alignment between two sequences, and in Chapter 2 we review a selection of sequence alignment algorithms that support varying alignment types and scoring schemes. We also describe methods for recording the optimal alignment through traceback and approaches to recording locally optimal alignments.

Algorithms for computing the optimal alignment between a pair of sequences are computationally expensive in both space and time. In Chapter 2, we also describe two heuristic approaches, called *banded alignment* and *dropoff alignment*, that aim to minimise the computational cost of aligning sequences by dismissing unlikely paths through the dynamic programming matrix. We also explain how *substitution matrices* are constructed and how they are employed to score matching pairs of amino acids in protein sequence alignments.

In Chapter 3 we survey existing methods for searching genomic collections, also known as homology search. We begin this chapter by introducing three classic approaches; Smith-Waterman search, FASTA [Pearson and Lipman, 1988] and BLAST [Altschul et al., 1990; 1997]. The Smith-Waterman search approach compares a single query to each sequence in a collection using the exhaustive Smith-Waterman alignment algorithm, and is impractical for any collection of significant size because it is too slow; we show that a search of GenBank with a typical query takes hours if not days. FASTA and BLAST both employ heuristic approaches that dismiss the majority of collection sequences using more coarse-grain comparisons before

aligning only a small number of remaining sequences using more time consuming methods. Of these two methods, BLAST is substantially faster than FASTA — at the expense of a small loss in sensitivity — and is used to conduct millions of searches worldwide each day. In Chapter 3, we describe the BLAST algorithm in detail, based on existing published descriptions [Altschul et al., 1990; 1997] and our own analysis of the popular NCBI-BLAST software. We also present our own in-depth analysis of the performance characteristics of each stage of the algorithm, and an analysis of usage data for BLAST searches conducted through the NCBI website[3].

In Chapter 3, we also discuss a range of alternative approaches to homology search, each of which provides advantages as well as disadvantages over the traditional approaches such as BLAST. Indexed-based approaches such as CAFE [Williams and Zobel, 2002], BLAT [Kent, 2002], PATTERNHUNTER [Ma et al., 2002; Li et al., 2004], and SSAHA [Ning et al., 2001] rely on an index structure such as those commonly employed in text retrieval [Witten et al., 1999] to efficiently search large collections. Discontinuous seeds have received considerable attention at late [Brown et al., 2004] and we survey existing literature concerning this approach to the first, hit detection stage of homology search. We discuss existing distributed methods for dividing the search task amongst a cluster of processors, and explain iterative methods such as PSI-BLAST [Altschul et al., 1997] and SAM-T98 [Karplus et al., 1998] that use sequence profiles.

Several important issues must be considered in the design of a homology search tool, which we also discuss in Chapter 3. We survey methods for calculating an alignment E-value that represents the likelihood that an alignment score is due to a chance similarity between unrelated sequences. Measures of the statistical significance of an alignment are useful for distinguishing true-positive matches between related sequences from false-positive matches between unrelated sequences. We describe in detail how BLAST calculates alignment E-values with respect to a scoring model and query and collection characteristics.

Low complexity regions in sequences adversely affect measures of the statistical significance of alignments, and the quality of alignments themselves, thus we also discuss approaches for identifying and removing low complexity regions during homology search in Chapter 3. We also survey existing approaches for assessing the retrieval effectiveness of genomic search tools that employ the sequence classifications found in databases such as SCOP [Murzin et al., 1995; Andreeva et al., 2004]. Finally, we discuss the pernicious effects of internal redundancy

[3]See `http://www.ncbi.nlm.nih.gov/BLAST/`

on search performance, in the form of near-duplicate entries, that is often found in genomic collections such as GenBank. We review existing approaches for managing redundancy that typically rely upon an all-against-all comparison of the collection to identify highly-similar sequences, and then prune entries to reduce the amount of duplication.

We begin describing our improvements to the BLAST homology search algorithm in Chapter 4. In this chapter, we describe three new approaches to the final stages of BLAST that compute gapped alignments. Our first contribution is an optimisation to the dynamic programming recurrence relations used to align sequences that reduces the amount of computation per cell with no effect on the result. The optimisation is based on a rearrangement of the recurrence relations that was used to align protein sequences and DNA sequences in Zhang et al. [1997], but has since received little attention. We show that our implementation of the gapped alignment stages of BLAST that incorporates the optimisation is around 20% faster than NCBI-BLAST for protein sequence alignments.

Our most significant contribution in Chapter 4 is a new step in the BLAST algorithm that reduces the computational cost of searching with negligible effect on accuracy. This new step — *semi-gapped alignment* — compromises between the efficiency of ungapped alignment and the accuracy of gapped alignment, allowing BLAST to accurately filter sequences between the second and third stages with lower computational cost. Semi-gapped alignment is a dynamic programming technique that permits gaps only at certain residue positions in the alignment, and closely approximates the more computationally expensive gapped alignment. Our experiments reveal that when carefully tuned, the semi-gapped alignment approach reduces average search times by 40% with no significant effect on accuracy.

In Chapter 4, we also propose an heuristic — *restricted insertion alignment* — that avoids unlikely evolutionary paths through the alignment matrix with the aim of reducing computational costs. The approach does not consider *adjacent gaps* during alignment, where an insertion in one sequence is followed immediately by an insertion in the other; we show that adjacent gaps are sufficiently rare that this has negligible effect on accuracy. Restricted insertion can be applied either to the gapped alignment or the semi-gapped alignment stages of BLAST, and provides a further 8% reduction in search time. When combined, our three improvements more than double the speed of the gapped alignment stages in BLAST, and we conclude that our techniques are important improvements to the algorithm. The results and discussions presented in this chapter are based on work published in Cameron et al. [2004].

In Chapter 5, we focus on the first stage of BLAST that identifies short, fixed length regions or *words* that match between the query sequence and sequences in the collection. We refer to

a single matching region between two sequences as a *hit*. BLAST uses a lookup structure to identify the location of words in the query that match each of the words extracted from the collection. We consider the effect of varying the word length and neighbourhood threshold parameters that are used for hit detection, and demonstrate that a larger word length can be used to achieve comparable search accuracy while reducing the amount of computation required to detect hits. Unfortunately, larger word lengths also increase the size of the lookup structure used to identify hits efficiently, resulting in poor cache performance and slower search times in practice.

NCBI-BLAST uses a codeword lookup approach to identify hits. Each word from the collection is converted into a special codeword that provides a unique integer representation of that word. The codeword is used to access an entry in a large lookup table that specifies an offset into the query for each hit. Our main contribution in Chapter 5 is a new data structure for identifying hits that is specifically designed for protein search. Our approach employs a deterministic finite automaton (DFA) that is optimised for modern hardware, making careful use of cache-conscious approaches to improve speed. For BLAST searches with default parameters, the DFA is around 94% smaller than the codeword lookup table used by NCBI-BLAST, and produces the same results as NCBI-BLAST but takes around 59% of the time on Intel-based platforms; we also present results for other popular architectures. Further, our method is practical for a word length of four, a parameter setting that is not currently supported by NCBI-BLAST. We also perform a detailed comparison of the two different search modes supported by BLAST, where either one hit or two hits are required to trigger an ungapped extension in the second stage. Our analysis shows that the two-hit mode provides faster search times than the one-hit mode with comparable search accuracy, in agreement with Altschul et al. [1997]. A preliminary version of the results and discussions presented in this chapter appeared in Cameron et al. [2006c].

In Chapter 6, we focus on the BLAST approach to searching nucleotide collections that is embodied in the BLASTN tool. Surprisingly, BLASTN has had very little attention paid to its algorithms, optimisations, and innovation, and the algorithms it uses do not follow those described in the 1997 BLAST paper [Altschul et al., 1997]. It is important that BLASTN is state-of-the-art, especially since nucleotide collections such as GenBank dwarf their protein counterparts in size, and our analysis of the NCBI usage statistics reveal that the majority of BLAST search conducted through their website are BLASTN searches. We propose several significant improvements to the BLASTN algorithm. Each of our schemes is based on compressed *byte packed* formats [Williams and Zobel, 1997], where four nucleotide bases are represented

by a single byte. Our approach compares query and collection sequences four bases at a time, permitting very fast query evaluation using lookup tables and numeric comparisons.

In Chapter 6, we describe improvements to each stage of BLASTN that utilise the byte packed compression scheme. We present new methods for hit detection and ungapped alignment that compare collection sequences in their compressed form, leading to a 43% reduction in average search time with no effect on the result. Our most significant innovations are two new, fast gapped alignment schemes for accurate sequence alignment without decompressing collection sequences. The first scheme, *bytepacked alignment*, places restrictions on the location of gaps allowing collection sequences to be processed one byte, or four nucleotide bases, at a time. This heuristic approach reduces the time taken to align sequences by 78% with negligible effect on accuracy. The second scheme, *table-driven alignment*, uses a specially designed lookup table to align four nucleotide bases at a time, with consideration for gaps, using an approach inspired by the Four Russians technique of Wu et al. [1996]. This approach leads to 72% faster gapped alignment times and guarantees no loss in accuracy. When combined, our innovations more than double the speed of BLASTN with no detectable effect on search accuracy. The results and discussions presented in this chapter are based on Cameron and Williams [2006].

Internal redundancy, in the form of near-duplicate entries, has several detrimental effects on search performance including slower query evaluation times, more repetitive results, less accurate measures of alignment significance and an increased likelihood of profile saturation in iterative search tools such as PSI-BLAST [Altschul et al., 1997]. In Chapter 7, we present new methods for managing redundancy in genomic databases. Our first contribution is a new approach for detecting highly similar sequences within large genomic collections. Near-duplicate detection has several important applications such as the assembly of expressed sequence tag data [Burke et al., 1999; Malde et al., 2003] and the comparison of entire genomes [Delcher et al., 1999; Miller, 2001], in addition to managing redundancy. While several approaches exist for this task, they are becoming infeasible — either in space or in time — as genomic collections continue to grow at a rapid pace. The most successful existing approaches use a form of all-against-all comparison that is quadratic in the number of sequences in the database, and scales poorly with the exponential grown of collections such as GenBank.

In Chapter 7, we present a novel approach for identifying highly similar sequences based on *document fingerprinting*, an technique that has been successfully applied to collections of text and web documents in Information Retrieval [Manber, 1994; Heintze, 1996; Broder

et al., 1997; Shivakumar and Garcia-Molina, 1999]. Our approach is based on the lossless SPEX algorithm [Bernstein and Zobel, 2004] that has been successfully employed to identify near-duplicate entries in large text collections [Bernstein and Zobel, 2005]. We find that genomic data has vastly differing characteristics to English text, rendering the original SPEX approach unsuitable for sequence data. We address this problem with a variation of the original approach, called slotted SPEX, that efficiently and accurately identifies pairs of highly-similar sequences in large genomic collections. Further, our approach uses a modest amount of memory and executes in a time roughly proportional to the size of the collection. We demonstrate substantial speed improvements compared to the CD-HIT algorithm, the most successful existing approach for clustering large protein sequence databases. Further, there is no significant change in sensitivity between CD-HIT and our own approach based on fingerprinting methods.

Our second contribution in Chapter 7 is a new approach for managing redundancy once it has been identified. Existing schemes create a representative sequence database by removing sequences so that no two entries share more than a certain level of similarity [Holm and Sander, 1998; Park et al., 2000b; Li et al., 2001b]. This leads to less accurate and less authoritative search results because the representative collection is not comprehensive. We present a new approach for managing redundancy that identifies clusters of near-identical sequences and uses a representative *union-sequence* to describe all members of a cluster. The union-sequence contains special wildcard characters to denote residue positions where the cluster members differ, so that it can represent all of the member sequences simultaneously. During search, the query is compared to the union-sequence representing each cluster only; cluster members are then aligned to the query if the union-sequence achieves a sufficiently high score. This reduces the number of sequence comparisons, leading to significantly faster search.

In addition to faster search times, our approach provides a form of compression. We store each member of a cluster as the difference between that member and the corresponding union-sequence. This edit information specifies how to transform the union-sequence representing a cluster into each member of that cluster, and reduces on-disk collection size significantly. We have applied our clustering strategy to BLAST and found that it affords a 27% reduction is collection size and a corresponding 22% decrease in search time with no significant change in overall search accuracy. When combined with our improvements to protein search described in Chapters 4 and 5, our schemes more than halve average query evaluation times when compared to NCBI-BLAST. The discussion and results presented in this chapter are based on

the published papers Cameron et al. [2006b], Bernstein and Cameron [2006], and Cameron et al. [2006a].

In Chapter 8, we discuss possible future extensions of the work presented in this thesis. We consider applying our new methods for fast homology search to the iterative PSI-BLAST algorithm [Altschul et al., 1997]. We expect that our improvements to BLAST will deliver a similar two-fold speed gain when applied to PSI-BLAST, and our novel scheme for managing redundancy will improve search accuracy by reducing the likelihood of profile saturation. We also propose optimisations that are specific to the iterative process. Next, we consider integrating the efficient comparison and filtering techniques employed by BLAST to index-based approaches such as CAFE [Williams and Zobel, 2002], and propose extending our work on duplicate detection to nucleotide data. Finally, we discuss the benefits of applying our novel fingerprinting algorithm described in Chapter 7 to English text, and consider tighter integration of the individual stages of BLAST. We also present concluding remarks in Chapter 8, and reiterate our contributions to genomic search that halve average query evaluation times for the popular BLAST homology search tool.

Chapter 2

Background

The study of biology at the molecular level has led to an in-depth understanding of life and the biological processes embodied in living organisms. This has resulted in a range of medical advances, including new medicines to combat a broad range of diseases such as cancer, neurodegenerative conditions, heart disease and HIV infection. Further, research into molecular biology has provided benefits to agriculture, with new methods for crop production and raising livestock.

The field of bioinformatics provides computational methods that are an integral part of research into genomics, biochemistry and molecular biology. A myriad of new approaches have been proposed to assist in obtaining, analysing and comparing biological data. In this chapter we describe some of the fundamental principles of genomics and molecular biology. We describe DNA and protein sequences, and explain how such sequence data is recorded. We also introduce basic methods for comparing genomic sequences through *sequence alignment* that are commonly employed. Sequence analysis can provide valuable insights into the biological role, chemical structure, and evolutionary origin of proteins and related DNA, and a sequence alignment provides a measure of similarity between two sequences by modelling the process through which proteins and DNA evolve. We also describe existing methods for reducing the computation involve in aligning sequences by dismissing unlikely evolutionary events.

We begin in Section 2.1 with an overview of DNA, RNA, and proteins and their functions in living organisms. We describe the universal genetic code and the process by which DNA encodes proteins. We also present a brief overview of the publicly available sequence data banks and methods for sequencing biological material.

Genomic sequences are commonly compared using a sequence alignment algorithm that models the process through which proteins and DNA evolve [Sankoff and Kruskal, 1983; Durbin, 1998]. In Section 2.2 we describe methods for aligning sequences using dynamic programming, and present commonly employed scoring schemes and alignment algorithms.

To reduce the time taken to align sequences, two heuristic approaches have been proposed that dismiss unlikely regions of the search space. In Section 2.3 we describe these two approaches, called *banded alignment* and *dropoff alignment*. We explain how these methods reduce the computation involved in aligning sequences with only minor effect on accuracy.

Finally, in Section 2.4 we describe substitution matrices, which are commonly used to score protein sequence alignments. We describe the general theory of substitution matrices and how the popular PAM [Dayhoff et al., 1978] and BLOSUM [Henikoff and Henikoff, 1992] matrices are constructed.

2.1 Genomic sequence data

The two types of genomic sequence data are nucleotide and protein sequences. A nucleotide sequence is the series of nucleotide bases that form a strand of deoxyribonucleic acid (DNA) or ribonucleic acid (RNA). There are four nucleotide bases. A protein sequence is an ordered series of amino acids that form a protein, with each of the twenty commonly occurring amino acids represented by a distinct character. We provide a brief introduction to proteins and protein sequence data in Section 2.1.1. We describe nucleotide sequence data and the role of DNA and RNA in living organisms in Section 2.1.2. Finally, in Section 2.1.3 we describe some of the publicly available collections of sequence data and related biological information. For more detailed introductions to genomics and the properties of proteins and DNA see Lehninger et al. [1993] and Alberts et al. [1994].

2.1.1 Proteins

Proteins perform a wide range of functions in living organisms, including catalyzing reactions, intra- and extracellular communication, and providing strength and protection to biological structures. A protein is formed by a chain of naturally occurring molecules called *amino acids*. An amino acid that forms part of the protein chain is often referred to as a *residue*. There are twenty commonly occurring amino acids that are notated by the three-letter or one-letter abbreviations listed in Table 2.1. Proteins can vary widely in length, however most proteins are a few hundred residues long. Once assembled as a chain of residues, a

Amino acid	Abbreviations		Amino acid	Abbreviations	
Alanine	Ala	A	Methionine	Met	M
Cysteine	Cys	C	Asparagine	Asn	N
Aspartate	Asp	D	Proline	Pro	P
Glutamate	Glu	E	Glutamine	Gln	Q
Phenylalanine	Phe	F	Arginine	Arg	R
Glycine	Gly	G	Serine	Ser	S
Histidine	His	H	Threonine	Thr	T
Isoleucine	Ile	I	Valine	Val	V
Lysine	Lys	K	Tryptophan	Trp	W
Leucine	Leu	L	Tyrosine	Tyr	Y

Table 2.1: The twenty standard amino acids and their associated three-letter and one-letter abbreviations.

protein forms a three-dimensional structure often referred to as the *tertiary* structure of the protein. The three-dimensional structure of a human protein α_1-antitrypsin is illustrated in Figure 2.1. The tertiary structure can be broken down into commonly occurring *secondary* structural elements that span several adjacent residues. Examples of secondary structural elements include α-helices, β-sheets, coils, and turns. The example protein in Figure 2.1 primarily contains α-helices, that are shown in light grey, and β-sheets, that are shown in dark grey.

At the simplest level, a protein can be represented as an amino acid sequence, which is called the *primary* structure. A protein sequence is typically represented by a text string using the one-letter amino acid abbreviations in Table 2.1. In addition to the twenty commonly occurring amino acids, the International Union of Pure and Applied Chemistry — International Union of Biochemistry and Molecular Biology (IUPAC-IUBMB) define four additional character codes that are used to represent ambiguous residues [Liébecq, 1992]: the letter `B` represents aspartic acid (`D`) or asparagine (`N`), the letter `Z` represents glutamate (`E`) or glutamine (`Q`), and the letters `X` and `U` represent any of the twenty amino acids. These extra codes are used to represent positions in a protein sequence where the exact residue is unknown, possibly due to limitations when determining protein sequences in laboratory experiments.

Two techniques are currently available to biologists for determining the precise tertiary

Figure 2.1: Human α_1-antitrypsin (PDB accession 1HP7) three-dimensional protein structure.

structure of a protein: X-ray crystallography and Nuclear Magnetic Resonance (NMR). Both techniques are time consuming and costly, and currently available technologies cannot be successfully applied to all proteins. Although more than three million protein sequences have been deposited into the publicly available databases over the past fifty years[1], the tertiary structure of most proteins remains unknown. A publicly available database of protein structures, the Protein Data Bank (PDB) [Berman et al., 2000], currently contains fewer than 40,000 entries[2], equating to roughly 1% of all sequenced proteins.

Proteins with a common evolutionary origin are referred to as being *homologous* to one another — a term that is often misused to refer to sequence similarity [Reeck et al., 1987]. Proteins with a shared evolutionary origin often have a similar tertiary structure and perform similar biological functions. As a result, information about the structure of a protein can provide useful insights into its origin and role in the organism. Further, proteins that share a common structure tend to be similar at the sequence level. Therefore, similarity between protein sequences can be used to infer common structure, function, and evolutionary origin.

[1]Based on number of sequences in the GenBank non-redundant protein database on November 23, 2005
[2]Based on statistics available at http://www.rcsb.org/pdb/ on November 23, 2005

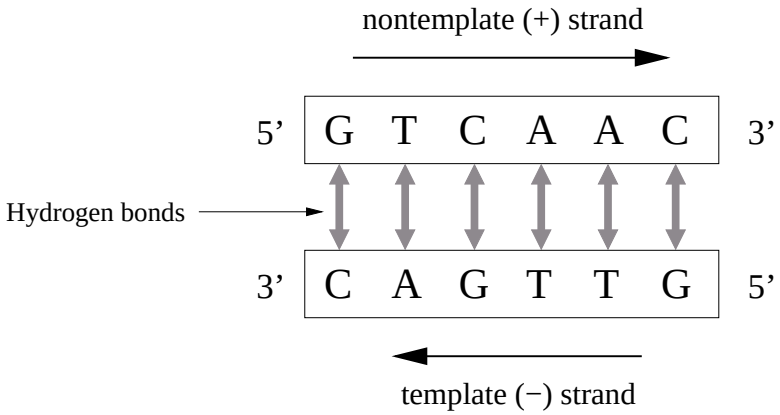

Figure 2.2: A pair of nucleotide strands form the DNA double helix structure. Each base in the nontemplate (+) strand is linked to its complement in the template (-) strand via a hydrogen bond. The two strands twist around each other to form a helix. A nucleotide sequence is read from the 5’ terminus to the 3’ terminus.

2.1.2 Nucleic acids

Deoxyribonucleic acid (DNA) is a molecular repository for genetic information and is responsible for encoding protein sequences in living organisms. A strand of DNA is a chemically linked chain of nucleotide bases that include the pyrimidines, *thymine* (T) and *cytosine* (C), and the purines, *guanine* (G) and *adenine* (A). The start of a nucleotide strand is denoted as the 5’ terminus and the end of the strand is the 3’ terminus. A pair of anti-parallel nucleotide strands form the DNA double helix, as illustrated in Figure 2.2. Each base in the plus (+) or nontemplate strand is linked via a hydrogen bond to a base in the minus (−) or template strand to form a *base pair*. Each pair of linked bases complement one another such that T is always paired with A and C is always paired with G.

A DNA or nucleotide sequence is represented by a string composed of the letters A, C, G, and T that are used to represent each of the four nucleotide bases. Nucleotide sequences range widely in length and can be up to several million bases long. An example of a DNA sequence is shown in Figure 2.3. As with protein sequences, the IUPAC-IUBMB define eleven ambiguity codes that are used in nucleotide sequences at positions where the exact base is unknown [Liébecq, 1992]. The complete set of nucleotide ambiguity codes is given in Table 2.2. The frequency of wildcards in collections such as GenBank is extremely low: over 99% of character occurrences are one of the four nucleotide bases, and almost 98% of the wildcard occurrences are N [Williams and Zobel, 1997].

```
tgcccaaatc gccaccggct tatttttggc tatacactat acagccgata cctccctcgc
attctcatct atcgcccaca tctgtcggga tgttaataac ggctgactcc ttcgtaatct
tcatgccaac ggtgcatcat ttttctttat ctgtatctac ttccacatcg gacggggcct
ttattacggc tcatacctct acaaagagac atgaaacatc ggagtaatcc tcttattctt
agtgatagcc acagcttttg tcggctacgt tcttccgtga gggcaaatgt cattttgagg
tgccacagta attactaacc ttctctcagc cgccccctac attggctacg acctcgtcca
atggatctga ggaggattct cagtagacaa tgctactctt acccgattct tcacattcca
ctttattctt ccgtttatta
```

Figure 2.3: A mitochondrial DNA sequence from the R. temporaria *(the common frog). GenBank accession number AY619564.*

Code	Meaning
R	G or A (purine)
Y	T or C (pyrimidine)
K	G or T
M	A or C
S	G or C
W	A or T

Code	Meaning
B	C, G or T
D	A, G or T
H	A, C or T
V	A, C or G
N	A, C, G, or T (any)

Table 2.2: Nucleotide ambiguity codes and their meanings.

During sequencing, the linear sequence of bases in DNA is read from the 5' terminus to the 3' terminus. Either strand may be sequenced, although the nontemplate strand is more commonly processed. Depending on which strand is processed, the resulting sequence may have one of two different possible orientations. The minus strand sequence is the reverse complement (also known as the Watson-Crick complement) of the plus strand sequence, that is, equivalent to the former after the sequence has been reversed and each of the bases has been replaced with its complement. For example, the plus strand sequence in Figure 2.2 is `GTCAAC` and the minus strand sequence is its reverse complement `GTTGAC`.

Almost every cell in a living organism contains a copy of the entire DNA or *genome* for that organism. Genomes vary in length between species; the human genome is roughly three billion base pairs in length yet the genome for *Haemophilus influenzae* is less than two million base pairs [Myers, 1999]. Short sections of the genome, referred to as *exons*, are used

Amino acid	Codon(s)	Amino acid	Codon(s)
A	GCT,GCC,GCA,GCG	M	ATG
C	TGC,TGT	N	AAT,AAC
D	GAT,GAC	P	CCT,CCC,CCA,CCG
E	GAA,GAG	Q	CAA,CAG
F	TTT,TTC	R	CGT,CGC,CGA,CGG,AGA,AGG
G	GGT,GGC,GGA,GGG	S	TCT,TCC,TCA,TCG,AGT,AGC
H	CAT,CAC	T	ACT,ACC,ACA,ACG
I	ATT,ATC,ATA	V	GTT,GTC,GTA,GTG
K	AAA,AAG	W	TGG
L	TTA,TTG,CTT,CTC,CTA,CTG	Y	TAT,TAC

Table 2.3: List of codons that encode each amino acid.

to encode protein sequences. The remaining sections of DNA are non-coding regions and are not responsible for encoding protein sequences.

DNA encodes for proteins through a two-stage process. First, the DNA is *transcribed* to create Ribonucleic acid (RNA), and second, the coding regions of the RNA is *translated* to produce a protein sequence. In an abstract sense, RNA has a similar structure to DNA and is also composed of a series of nucleotide bases or base pairs. The only significant differences are that RNA is generally composed of a single strand and that the base *uracil* (U) occurs in RNA instead of *thymine* (T); in the context of nucleotide sequence analysis U and T are interchangeable. During transcription, the encoding portion of the DNA is transcribed into *messenger* RNA. The RNA transcript is a copy of the nontemplate strand of DNA. The messenger RNA is then translated into a protein sequence, where each codon (sequence of three consecutive nucleotide bases) maps to a single amino acid using the universal genetic code.

The mapping between triplets of bases in DNA and amino acids in proteins that applies to living organisms is listed in Table 2.3. The translation process is lossy due to degeneracy in the genetic code, where multiple codons encode the same amino acid. This is because the number of distinct nucleotide codons is $4^3 = 64$ which is considerable larger than 20, the number of amino acids that they encode. As a result, several different codons may translate to the same amino acid; for example the codons, `ATT`, `ATC`, and `ATA` all encode for isoleucine.

2.1.3 Genomic collections

In this section we briefly describe some of the frequently used genomic collections. A more detailed list of publicly available genomic databases can be found elsewhere [Galperin, 2004].

Three publicly available services provide a major comprehensive collection of nucleotide data from repositories worldwide:

- the GenBank nucleotide database that is maintained by the National Center for Biotechnology Information (NCBI) in the USA [Benson et al., 1993; 2005]; and
- the DNA Database of Japan (DDBJ) that is maintained by the National Institute of Genetics in Japan [Miyazaki et al., 2004]; and
- the European Molecular Biology Laboratory (EMBL) that is based in the United Kingdom also provides an exhaustive collection of sequences [Kanz et al., 2005].

The three collections are regularly updated and mirror the contents of one another daily, so that recently deposited sequences in one database will soon appear in all three collections [Rapp and Wheeler, 2005]. We use the GenBank database for the majority of experiments in this thesis.

The GenBank nucleotide database currently contains more than 53 billion nucleotide base pairs stored in almost 50 million sequences from over 150,000 different organisms[3]. The collection has roughly doubled in size every 1.4 years, which is faster than improvements in the processing power of modern workstations [Attwood and Higgs, 2004]. Further, this trend is expected to continue with new technologies for high-throughput sequencing and support from scientific journals, many of which now require new sequences to be submitted to GenBank before the related work is published [Rapp and Wheeler, 2005]. The resulting exponential growth of data in GenBank between 1982 and 2004 is illustrated in Figure 2.4.

The NCBI also distributes a compact version of the GenBank nucleotide collection that is roughly one-quarter of the original size and does not include "high-throughput, patent, genomic or sequence tagged cite (STS) sequences" [McGinnis and Madden, 2004] that are typically of less interest to researchers. The pruned collection is the default database for BLAST [Altschul et al., 1990; 1997] searches via the NCBI website[4], with the complete GenBank database no longer supported by the online tool. We refer to the compact version of the collection as the GenBank non-redundant (NR) nucleotide database throughout this

[3] GenBank Release 150, October 2005

[4] See: `http://www.ncbi.nlm.nih.gov/BLAST/`

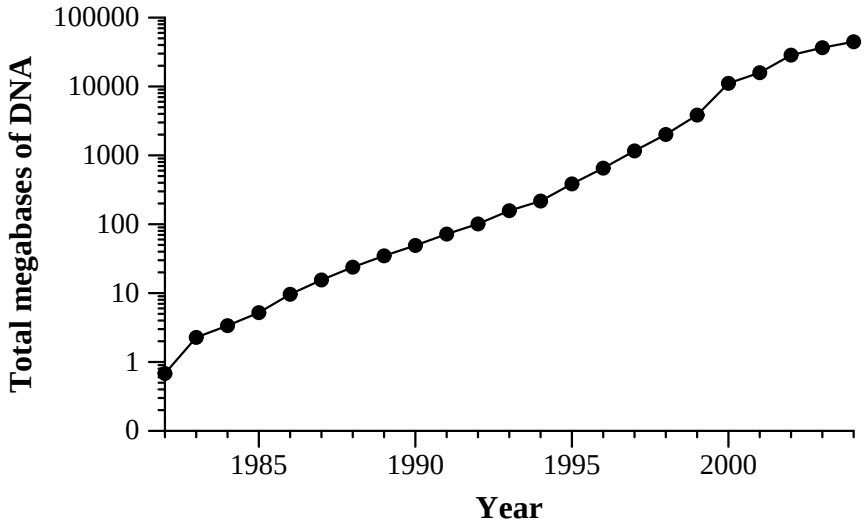

Figure 2.4: Size of GenBank between 1982 and 2004.

thesis and use a copy of the NR nucleotide database downloaded 4 April 2005 and containing 13,626,473,199 basepairs in 3,023,092 sequences for our experiments.

Several publicly available resources provide comprehensive collections of protein sequence data and related functional and structural information, including:

- the SWISS-PROT protein database [Boeckmann et al., 2003] and Protein Information Resource (PIR) [Wu et al., 2003] non-redundant (NREF) database, which are comprehensive protein sequence databases; and,
- the Protein Sequence Database (PSD) component of the PIR, which provides functional annotation of proteins and contains over 280,000 entries[5]; and,
- the Protein Data Bank (PDB) [Berman et al., 2000], which is a collection of structural data for over 33,000 proteins[6].

Further, the GenBank non-redundant (NR) protein database [Wheeler et al., 2002] maintained by NCBI contains sequences from all of the above collections. For our experiments in this thesis, we use a recent copy of the NR protein database that was downloaded 18 August 2005 and contains 2,739,666 sequences in around 938,574,629 characters of sequence data unless reported otherwise. Although the quantity of protein sequence information in

[5]Release 80.0, December 2004

[6]Based on statistics from http://www.rcsb.org/pdb, November 2005

GenBank is considerably less than the quantity of nucleotide data, protein sequences still represents a sizeable portion of the data bank.

In addition to the primary genomic collections that we have described, several secondary collections have been constructed that classify proteins into related families based on structural, functional, or sequence similarities. The SCOP database [Murzin et al., 1995; Andreeva et al., 2004] uses structural information to classify proteins. Each protein in SCOP is hierarchically classified into a class, fold, superfamily, and family. The classifications are based on structures of over 25,000 proteins from the PDB database[7]. The CATH database [Orengo et al., 1997] also provides a hierarchical classification for proteins that is based on four levels: protein class (C), architecture (A), topology (T), and homologous superfamily (H). Unlike SCOP, the CATH database includes entries for proteins where the precise structure is unknown. This is possible because the classification process is based on sequence as well as structural similarities, leading to arguably less reliable classifications [Brenner et al., 1998].

In addition to shorter protein and nucleotide sequences, several complete genomes have been sequenced and assembled as the result of large-scale genome sequencing projects. The first organism to be completely sequenced was *Haemophilus influenzae* with the entire 1.83 million base pair genome recorded in 1995 [Myers, 1999]. Since then, over 300 genomes have been sequenced and made publicly available through services such as the Genomes OnLine Database [Bernal et al., 2001]. The number of sequenced genomes is roughly doubling every 1.3 years [Attwood and Higgs, 2004].

An increasing number of genomes have been sequenced, but technical limitations continue to hamper the process. Current sequencing technologies are limited to reads of a few hundred nucleotide bases at a time, so entire genomes cannot be sequenced directly [Myers, 1999; Pop et al., 2002]. Instead, an approach called *shotgun sequencing* is used to acquire short fragments of the longer sequence, which are then assembled into a larger sequence. The task of fragment assembly is complicated by several issues including sequencing errors, incomplete coverage, and unknown orientation of fragments. For a more detailed description of genome sequencing see Myers [1999] and Pop et al. [2002].

2.1.4 Summary

In this section we have provided an overview of genomic sequence data. We have described amino acid sequences and nucleotide sequences that are the primary structure of proteins and

[7] Version 1.69, July 2005

DNA. Proteins perform a range of functions in biological organisms and are represented as a sequence of the twenty commonly occurring amino acids. A strand of DNA is composed of a sequence over an alphabet of four nucleotide bases. DNA is responsible for encoding proteins through a two-stage process of transcription and translation, where a nucleotide triplet or codon encodes for a single amino acid. Hence, the two forms of sequence data are closely related and can provide valuable insights into the function, structure, and evolutionary origin of proteins and DNA.

We have surveyed some of the important public repositories of genomic data. Comprehensive sequence databanks such as GenBank contain more than fifty millions DNA sequences and over three million protein sequences. Smaller collections such as the PDB, PIR, SCOP, and CATH provide structural or functional annotations for a smaller number of proteins; the precise structure, function, and evolutionary relationship of the vast majority of sequences remains unknown. Recently, the sequencing of complete genomes has become possible with several hundred genomes sequenced to date.

In the next section, we discuss methods for comparing sequences through sequence alignment, including methods for scoring alignments and algorithms for finding the optimal or highest-scoring alignment between two sequences.

2.2 Sequence alignment

Homologous relationships between proteins are more easily detected using structures, rather than sequences [Sierk and Pearson, 2004]. Unfortunately, the precise three-dimensional structure of most proteins remains unknown and biologists must often rely upon sequence data for most computational or *in silico* studies. Homologous proteins typically share at least 20-25% identity at the sequence level [Pearson, 1996]. As a result, sequence similarity can be used to infer similar structure, function and shared evolutionary origin of genetic material. Sequence comparison is therefore an invaluable tool for directing research by providing information about newly sequenced or poorly annotated proteins. For example, a biologist may identify a high degree of sequence similarity between an unannotated protein found in a human, and a protein that has been shown experimentally to perform a specific function in a mouse. The sequence similarity suggests that the unannotated protein has a similar chemical structure and evolutionary history, and is responsible for a similar function in humans. The relationship can be used as the basis of experimental work to test this hypothesis. Sequence comparison can also be used for a range of bioinformatic studies and can lead to the discovery

of new families of proteins [Pearson, 1990].

The degree of similarity between genomic sequences is commonly measured using a sequence alignment score, a measure of similarity that is closely related to other measures for approximate string matching such as the Levenshtein distance [Levenshtein, 1966] or Longest Common Subsequence (LCS) [Needleman and Wunsch, 1970; Apostolico and Guerra, 1987]. (For a more detailed overview of approximate string matching algorithms see Hall and Dowling [1980] and Navarro [2001].) A sequence alignment provides a measure of the number of point mutations, or elementary changes, required to transform one sequence into another and has been shown to be an effective model of the evolutionary process [Pearson and Miller, 1992; Crooks et al., 2005].

Three basic types of changes are common in the evolution of biological sequences. First, one nucleotide base or amino acid maybe be substituted for another. Second, a base or residue may be inserted into the sequence. Last, a base or residue maybe be deleted from the sequence. These three elementary changes are the basis of sequence alignment [Sankoff and Kruskal, 1983; Durbin, 1998].

Consider the short pairwise alignment of the two nucleotide sequences in Figure 2.5. The alignment contains five *matches*, where a base in sequence A matches the corresponding base in sequence B, as denoted by the symbol `|` in the middle line. The alignment also contains a single mismatch or substitution, where the third base `G` from sequence A does not match the third base `A` from sequence B. Finally, the alignment contains two insertions in sequence A that are represented by dashes (`-`). An insertion in one sequence is equivalent to a deletion in the other; in this example, the insertions in sequence A can also be regarded as deletions with respect to sequence B. Insertions and deletions are collectively referred to as *indels*, and a series of adjacent insertions in an alignment is referred to as a gap. The alignment in Figure 2.5 contains two indels, or a gap of length two. We discuss methods for scoring each component of an alignment in Section 2.2.1.

Given an alignment scoring scheme, the highest scoring alignment between two sequences is called the *optimal alignment*. To compute an optimal alignment, all possible evolutionary pathways between two sequences are computed with respect to a scoring scheme [Sankoff and Kruskal, 1983]. Two types of alignment algorithms are used to computing an optimal alignment: global and local. A global pairwise alignment spans the entire length of both sequences, starting at the beginning and finishing at the end of each sequence. We describe methods for computing the optimal global alignment using dynamic programming in Section 2.2.3. A local alignment does not need to cover the entire length of both sequences.

```
Sequence A:      A C G T - - C C
                 | |   |     | |
Sequence B:      A C A T G G C C

Alignment score: +2+2−1+2−4−1+2+2 = 4
```

Figure 2.5: Pairwise alignment of nucleotide sequences `ACGTCC` *and* `ACATGGCC`*. Insertion are represented by a dash and matches are represented by a vertical line. The alignment contains five matches, one mismatch, and a gap of length two. Using a scoring scheme of +2 for a match, -1 for a mismatch and affine gap penalties of -3 for opening a gap and -1 for each insertion the alignment score is 4.*

Instead, a local alignment may span only a region from each sequence. We discuss methods for computing optimal local alignments in Section 2.2.4.

2.2.1 Scoring sequence alignments

The optimal alignment score of two sequences provides a measure of their similarity. (Global alignment also provides a measure of the distance between two sequences.) An alignment scoring scheme consist of two main components: a method for scoring matches and mismatches between pairs of aligned bases or residues, and a method for scoring gaps. For nucleotide alignments, a constant reward for each match and a constant penalty for each mismatch are commonly applied [Altschul et al., 1990; Nicholas et al., 2000]. Matching bases are rewarded and increase the alignment score by r while mismatching bases are penalised and decrease the score by p, where r and p are positive integers. For protein sequences, a *substitution matrix* is commonly employed to score the alignment of amino acid pairs [Dayhoff et al., 1978; Henikoff and Henikoff, 1992]. The likelihood of substitution is greater for some amino acids pairs then others and the substitution matrix encapsulates this information. The matrix specifies an alignment score for every possible pair of residues. In general, pairs of identical residues and residues with similar properties produce a positive alignment score, while dissimilar residues produce a negative score. We discuss substitution matrices and methods for their construction further in Section 2.4.

The two most common methods for scoring gaps in the alignment of biological sequences are *linear gap costs* [Needleman and Wunsch, 1970] and *affine gap costs* [Gotoh, 1982]. For linear gap costs, the cost c of a gap of length k is defined by $c(k) = k \times e$, where e is

the cost of each insertion in the gap and $e > 0,\ k \geq 1$. The affine gap model applies an additional penalty o for opening a gap as well as the penalty e for each insertion in that gap. For affine gap costs, the cost c of a gap of length k is defined as $c(k) = o + k \times e$. The penalty for beginning a gap is typically high but the penalty for continuing the gap is usually low. The affine gap model has been shown to be more effective than the linear gap model because adjacent insertions are more likely to occur in the evolutionary process than non-adjacent insertions [Fitch and Smith, 1983]. Further, we show in Section 2.2.5 that optimal alignments can be computed relatively efficiently using affine gap costs. The choice of gap open and extension penalties are typically made through empirical studies due to the lack of solid statistical theory for penalising gaps [Pearson, 1996; Reese and Pearson, 2002].

To illustrate the popular methods for scoring sequence alignments let us return to the example in Figure 2.5 and compute the score for this alignment using a scoring scheme with a match reward of $r = 2$, a mismatch penalty of $p = 1$ and affine gap costs of $o = 3$ and $e = 1$. The five matches, one mismatch, and gap of length two result in a cumulative alignment score of $(5 \times 2) - (1 \times 1) - (3 + 2 \times 1) = 4$.

2.2.2 Protein versus nucleotide alignment

Because proteins are encoded by DNA through the genetic code, DNA sequence similarities can also provide an insight into the relationship between proteins and their role in various biological functions. However, protein sequence comparisons are generally more informative and are preferred by biologists to DNA sequence comparisons [Pearson, 1990; 1996; Nicholas et al., 2000]; protein sequence comparison can be used to identify a common ancestory from more than 2.5 billion years ago, however it is difficult to identify similarities between nucleotide sequences that diverged more than 200–600 million years ago [Pearson, 1996]. There are two main reasons why protein comparison is more sensitive to distantly related sequences:

- Changes in DNA do not necessarily translate to changes in the encoded protein, due to degeneracy in the genetic code [Pearson, 1996]. Nucleotide comparisons fail to distinguish between the more probable base mutations that do not affect the encoded protein sequence and the less probable mutations that do. For example, we would expect that the nucleotide codon `TTT` that encodes phenylalanine is more likely to mutate into the codon `TTC` that also encodes phenylalanine, than to the codon `TTA` that encodes leucine, despite only one base change in both cases.

- Protein sequence comparisons use a substitution matrix that reflects the varying likelihood of substitutions between differing amino acid pairs. Nucleotide alignments are generally scored using match and mismatch penalties that do not take the properties of the encoded amino acids into consideration [Pearson, 1996]. For example, the codon `AGA`, which encodes for arginine, is more likely to mutate into `AAA`, which encodes for lysine, than to mutate into `GGA`, which encodes for glycine, because arginine and lysine are physico-chemically similar but arginine and glycine are not [Taylor, 1986]. In both cases, only one base differs between the codons.

Further, a large quantity of available nucleotide data represents non-coding regions that does not encode for proteins, and alignments with non-coding regions are probably less likely to be of interest to biologists; nucleotide alignments do not distinguish between coding and non-coding regions.

However, despite a preference for protein comparisons, nucleotide comparisons are sometimes necessary. DNA databases are generally more up-to-date due to annotation bottlenecks and delays in translating coding regions in DNA to protein sequences [Anderson and Brass, 1998]. Further, errors sometimes appear in protein sequences due to errors when translating from a DNA sequence, and the non-coding regions of DNA may be of interest to some biologists.

In addition to protein-protein and nucleotide-nucleotide comparisons, methods for aligning a nucleotide sequence with a protein sequence are also available [Zhang et al., 1997; Ko et al., 2004]. A DNA sequence can be aligned with a protein sequence by translating codons in the DNA and aligning the resulting protein sequence. Unfortunately, the translation process is ambiguous for two reasons. First, the exact start position of the coding region of DNA is often unknown; because each nucleotide triplet encodes for a single amino acid, three completely different sequences will result from the translation of a nucleotide sequence at different offsets. Second, the orientation of the nucleotide strand from which a sequence is derived is often unknown. If the sequence is derived from the template strand then the sequence must be reversed and complemented before the encoding scheme in Table 2.3 is applied. As a result of these ambiguities, a total of six different *reading frames* (three possible frames for each of the two possible strands) must be considered when decoding a nucleotide sequence, and nucleotide-protein comparisons are therefore computationally more expensive. To further complicate the task of aligning translated sequences, insertion and deletions in DNA can cause frameshift errors (changes in the reading frame) [Ko et al., 2004].

2.2.3 Global Alignment

The optimal alignment between a pair of sequences is computed using dynamic programming, an algorithm design technique that finds solutions by decomposing a problem into a series of tabular operations [Levitin, 2002]. A dynamic programming algorithm consists of three main components: a recurrence relation that defines how to score a solution, a series of tabular operations that compute the highest-scoring or optimal solution, and a traceback method for discovering the solution or solutions with an highest score. In the context of sequence alignment, the traceback provides the optimal sequence alignment that produces the highest score.

Methods for aligning sequences using dynamic programming were independently discovered by several researchers [Sankoff and Kruskal, 1983]. The original algorithms for aligning biological sequences were described by Needleman and Wunsch [1970] and Sellers [1974]. In this section, we describe the global alignment algorithm for aligning two sequences with linear gap costs. Our description follows the efficient approach by Sellers that requires O(mn) time. For a more detailed review of sequence alignment algorithms see Waterman [1984].

When aligning a pair of sequences x and y, three possible evolutionary events are considered with respect to each possible pair of residues drawn from the two sequences: first, the residues are aligned (meaning the residues are conserved because they are the same, or one is substituted for the other); second, an insertion is made with respect to y; and, last, an insertion is made with respect to x. As discussed in Section 2.2.1, the first class of events is scored using an amino acid substitution matrix, or simple nucleotide match and mismatch penalties.

The alignment of two sequences of lengths l_x and l_y requires the tabulation of scores in an *alignment matrix* of size $(l_x+1)\times(l_y+1)$ [Sankoff and Kruskal, 1983; Durbin, 1998]. Each cell $[i,j]$ in the matrix represents the highest scoring alignment between x and y that ends with the i^{th} residue in x and the j^{th} residue in y. The value in each cell $[i,j]$ is dependent on three of the immediate neighbours of the cell at coordinates $[i-1,j-1]$, $[i-1,j]$, and $[i,j-1]$, and these map to the three possible evolutionary events that can affect the alignment of the i^{th} residue in x and the j^{th} residue in y. This dependence is illustrated in Figure 2.6, where the alignment of a pair of amino acids or bases is represented by a diagonal arrow and insertions in y and x are represented by horizontal and vertical arrows respectively.

The Sellers algorithm for globally aligning sequences with linear gap costs records a single value $B(i,j)$ for each cell in the matrix; this value is the highest score for an alignment

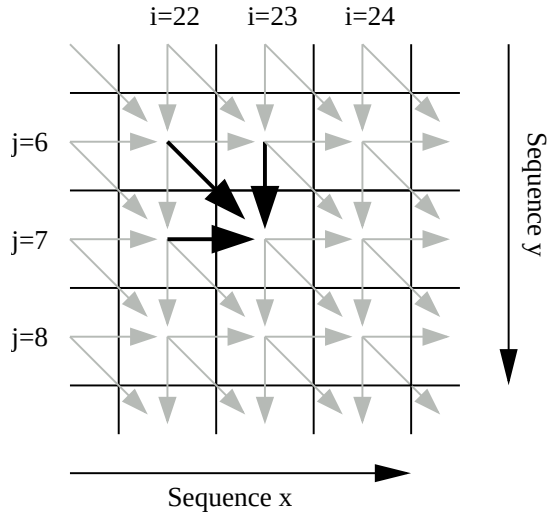

Figure 2.6: Portion of the dynamic programming matrix used to align sequences x and y. Diagonal arrows represent the alignment of a pair of amino acids or bases, and vertical and horizontal arrows represent insertions in x and y respectively. Values for the cell at $i = 23, j = 7$ are dependent on the three immediate neighbours at coordinates $[22, 7]$, $[22, 6]$ and $[23, 6]$.

ending at $[i, j]$, that is, the alignment of the subsequences $x_1...x_i$ and $y_1...y_j$. The following recurrence relations are employed to compute $B(i, j)$ for each cell and find the score of the optimal global alignment between x and y using linear gap costs:

$$B(i,j) = \max \begin{cases} B(i-1, j-1) + s(x_i, y_j) \\ B(i-1, j) - e \\ B(i, j-1) - e \end{cases}$$

where $s(x_i, y_j)$ is the score resulting from the alignment of the i^{th} character of x and the j^{th} character of y, and e is the insertion penalty. In addition to these recurrence relations, initialisation rules are required to handle boundary conditions; for global alignment, all cells where $i = 0$ or $j = 0$ are initialised to $-\infty$, except for the alignment starting point $[0, 0]$ that is initialised to zero. These initialisation rules restrict sequence alignments to start at the beginning of both sequences.

Let us consider globally aligning the sequences `ACATGGCC` and `ACGTCC` with a match score of 2, a mismatch penalty of -1, and an insertion penalty of -1, as shown in Figure 2.7. First, the starting point for the alignment at $[0, 0]$ is initialised to zero, and the remaining cells in

the first row and first column are initialised to $-\infty$. Each cell in the alignment matrix is then processed sequentially using the recurrence relations above; this is typically done by scanning from top to bottom, and left to right, starting at $[1, 1]$ and finishing at $[l_x, l_y]$. The value $B(i, j)$ is calculated for each cell, and an arrow pointing into the cell indicates which of the three evolutionary events was considered when calculating the cell value. Consider, for example, the processing of the cell at position $[2, 2]$ that represents the intersection of base `C` from the first sequence and `C` from the second sequence. Three events are considered using the recurrence relations above: a match event that produces a score of $B(1, 1)+2 = 2+2 = 4$, an insertion in the first (top) sequence that produces a score of $B(1, 2) - 1 = 1 - 1 = 0$ and an insertion in the second (left) sequence that produces a score of $B(2, 1) - 1 = 1 - 1 = 0$. The largest of the three scores is recorded; in this case the largest value results from the match event that is represented by a diagonal arrow from the cell at $[1, 1]$ to the cell at $[2, 2]$. Once every cell in the alignment matrix has been processed in this manner, the final value $B(l_x, l_y) = 7$ is the optimal global alignment score for the two sequences and corresponds to the alignment shown on the right-hand side of the figure.

The algorithm we have described is suitable for calculating the score of the optimal alignment only, and not the optimal alignment itself. We discuss methods for computing optimal alignments by performing a traceback in Section 2.2.6. The score-only method we have described requires $O(l_x l_y)$ time but only $O(l_x)$ space, because each row in the alignment matrix is dependent on the previous row only. Consequently, only one row needs to be retained to compute the optimal alignment score.

2.2.4 Local alignment

During the evolutionary process, portions of DNA or protein that are central to its role or function in an organism, such as active or binding sites, often remain relatively unchanged while other parts of the sequence vary considerably [Myers, 1991]. As a result, related sequences often share a partial region of localised similarity even though they are not similar globally. Hence, local alignment algorithms that align regions of biological sequences are more commonly used to perform sequence similarity searches than global alignment methods [Pearson, 1996].

To illustrate local alignment, let us consider aligning the sequences `AAAACDEFGGGGG` and `HHHHCDEFIIIII` that share a region of local similarity; both sequences contain the motif `CDEF`. Figure 2.8 shows the optimal global and local alignments for these two sequences using

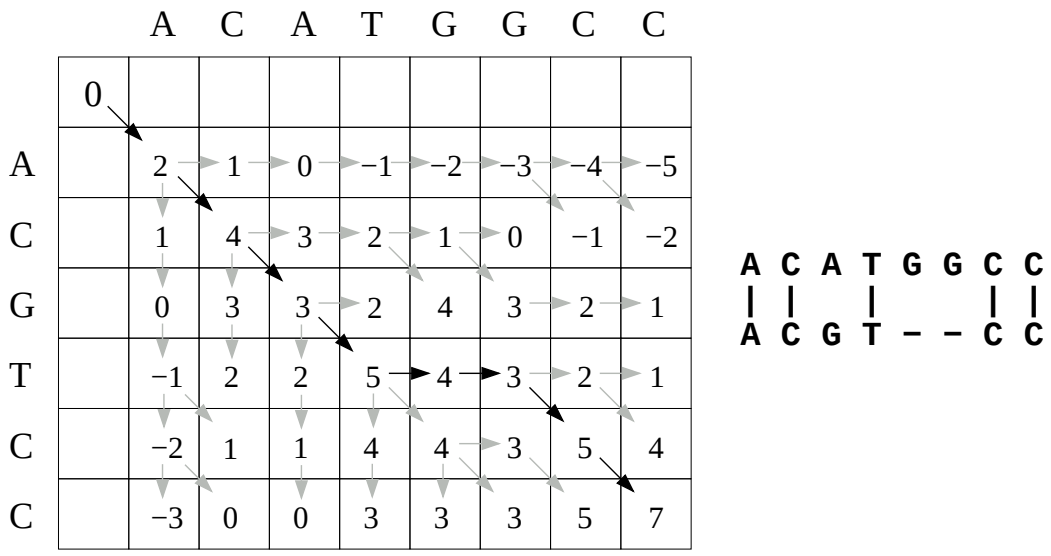

Figure 2.7: The dynamic programming matrix used to globally align sequences `ACATGGCC` *and* `ACGTCC` *with a match score of 2, a mismatch score of -1, and an insertion penalty of -1. Arrows are used to indicate how the value for each cell is derived. Blank cells have an initialised value of* $-\infty$*. The dark arrows represent the traceback path corresponding to the optimal alignment shown on the right. The optimal alignment score of 7 is recorded in the bottom right cell.*

```
A A A A C D E F G G G G G
        | | | |
H H H H C D E F I I I I I
```

Global alignment with four matches, nine mismatches

```
C D E F
| | | |
C D E F
```

Local alignment with four matches

Figure 2.8: Optimal global and local alignments of the sequences `AAAACDEFGGGGG` *and* `HHHHCDEFIIIII` *using scoring scheme with +2 for a match and -1 for a mismatch.*

a simple scoring scheme of $+2$ for a match and -1 for a mismatch. Despite the presence of a locally conserved region in both sequences, the optimal global alignment contains four matches and nine mismatches with an overall alignment score of -1. The local alignment considers only the conserved region and contains four matches with an alignment score of 8. Unlike global alignment, optimal local alignment scores for pairs of sequences cannot be negative.

The Smith-Waterman algorithm [1981] performs a local alignment between two sequences. Local alignment differs from global alignment in two ways: first, the alignment may start

at any residue pair from the two sequences, and second, the highest scoring cell in the alignment matrix provides an end-point for the optimal alignment. To consider alignments starting at any point, negative cell values are disallowed. This involves minor adjustment to the recurrence relations described in Section 2.2.3. The following recurrence relations are used to find the score of the optimal local alignment between sequences x and y using linear gap costs:

$$B(i,j) = \max \begin{cases} B(i-1,j-1) + s(x_i, y_j) \\ B(i-1,j) - e \\ B(i,j-1) - e \\ 0 \end{cases}$$

The largest value $B(i,j)$ in the alignment matrix is the optimal local alignment score. The local alignment algorithm requires different initialisation rules to handle boundary conditions then global alignment: all cells where $i = 0$ or $j = 0$ are initialised to zero instead of $-\infty$.

A handful of optimisations to the Smith-Waterman algorithm have been proposed. The most commonly used optimisation was first used by Green in his SWAT implementation of the algorithm[8], but was never published. The optimisation is based on the observation that most cells in the alignment matrix do not score above the open gap penalty o when popular substitution matrices and affine gap costs are employed. This can be exploited to avoid considering an open gap event when processing subsequent adjacent cells, resulting in a reduction in computation for some cells. Rognes and Seeberg [2000] report that Green's optimisation roughly doubles the speed of Smith-Waterman when compared to the naïve approach.

Myers and Durbin [2003] describe another approach to local alignment that uses a lookup table to perform faster alignment. The authors report a speed increase of up to 1.5 times when using the popular BLOSUM62 data mutation matrix [Henikoff and Henikoff, 1992] and up to twice as fast with other matrices. Their approach also relies on the sparsity of high scoring cells in the alignment matrix.

2.2.5 Alignment with affine gap penalties

The approaches described in Sections 2.2.3 and 2.2.4 align two sequences using linear gap costs where each insertion in a gap is penalised using a fixed insertion penalty e. In this section

[8] http://www.phrap.org/phredphrap/swat.html

we describe the algorithm by Gotoh [1982] for aligning sequences with affine gap costs. The affine gap penalty system applies a fixed cost o for opening a gap and an extension penalty e for each insertion in the gap. Specifically, the cost c of a gap of length k is defined by $c(k) = k \times e + o$, where $e > 0,\ o > 0,\ k \geq 1$. We add to this definition the pre-computed cost d of opening a gap and making the first insertion in that gap, that is, $d = o + e$. Affine gap penalties provide a better approximation of the evolutionary process than linear penalties and are widely used in sequence alignment [Fitch and Smith, 1983].

Gotoh's algorithm uses dynamic programming and is similar to the Needleman-Wunsch and Smith-Waterman approaches. Gotoh's approach differs in two ways: first, three values are recorded for each cell in the alignment matrix instead of one, and second, a different set of recurrence relations are used to calculated the optimal alignment score. Gotoh's algorithm records the following values for each cell in the matrix when aligning sequences x and y; $B(i,j)$ is the best score for an alignment ending at $[i,j]$, $I_x(i,j)$ is the best score for an alignment ending at $[i,j]$ with an insertion in x and $I_y(i,j)$ is the best score for an alignment ending at $[i,j]$ with an insertion in y. Using these values, the following recurrence relations are employed to compute the score of the optimal alignment between x and y:

$$
\begin{aligned}
M(i,j) = &\quad B(i-1,j-1) + s(x_i, y_j) \\
I_x(i,j) = &\quad \max \begin{cases} B(i-1,j) - d \\ I_x(i-1,j) - e \end{cases} \\
I_y(i,j) = &\quad \max \begin{cases} B(i,j-1) - d \\ I_y(i,j-1) - e \end{cases} \\
B(i,j) = &\quad \max \begin{cases} I_x(i,j) \\ I_y(i,j) \\ M(i,j) \end{cases}
\end{aligned}
$$

where the scalar value $M(i,j)$ represents the best score for an alignment ending at $[i,j]$ with a match. These recurrence relations are suitable for globally aligning two sequences; minor modifications are required to dismiss negative values and perform local alignment with affine gap costs [Durbin, 1998]. Gotoh's algorithm requires $O(l_x l_y)$ time and $O(l_x)$ space and has roughly three times the memory usage of the linear gap penalty approaches.

Two-state variation

Durbin [1998] describes a variation of Gotoh's algorithm that records only two values for each cell in the alignment matrix. The two-state variation combines $I_x(i,j)$ and $I_y(i,j)$ into

```
AGCATT------AGGACCTTGAACATT
||  |||          |||  |||  ||
AGAATTCCATCT----CCTCGAAACTT
```

Figure 2.9: An example of the infrequent case where an alignment contains two adjacent gaps.

a single maximum value, $I(i,j)$, that represents the best score for an alignment ending at $[i,j]$ with an insertion in either direction. The following recurrence relations are employed by the two-state approach:

$$
\begin{aligned}
M(i,j) = &\quad B(i-1,j-1) + s(x_i, y_j) \\
I(i,j) = &\quad \max \begin{cases} B(i-1,j) - d \\ I(i-1,j) - e \\ B(i,j-1) - d \\ I(i,j-1) - e \end{cases} \\
B(i,j) = &\quad \max \begin{cases} I(i,j) \\ M(i,j) \end{cases}
\end{aligned}
$$

The two-state variation is easier to implement and requires less memory than Gotoh's original algorithm. However, the approach is not guaranteed to provide the correct optimal alignment score when aligning certain sequences with some scoring schemes. Specifically, the result for the two-state variation differs when scoring *adjacent gaps*, that is, where a gap in one sequence is followed immediately by a gap in the other, as illustrated in Figure 2.9. In the two-state variation, a single open gap penalty is incurred for the pair of adjacent gaps, unlike the original algorithm where two open gap penalties are applied. As a result, the two-state variation may over-estimate the optimal alignment score. However, Durbin [1998] report that adjacent gaps rarely appear in optimal alignments, and do not occur when the lowest mismatch score is greater than or equal to $-2e$.

2.2.6 Recording traceback

The alignment algorithms we have described so far in this chapter calculate the score of the optimal alignment, but do not provide the optimal alignment itself. In this section we describe methods for finding the optimal alignment between two sequences using the traceback approach. See Durbin [1998] for a more in-depth introduction to traceback methods.

To record the optimal alignment using traceback, additional information is recorded about how scores are derived for each cell in the matrix. Once the entire matrix has been processed, a traceback is performed to find the path resulting in the optimal score. When linear gap costs are employed, one of three possible values are recorded for each cell to indicate whether the score $B(i, j)$ was derived through a match or mismatch, an insertion in x, or an insertion in y. For affine gap costs, additional information is recorded indicating how the values $I_x(i, j)$ and $I_y(i, j)$ were derived. To find the optimal global alignment, the series of events that led to the optimal alignment score are followed back to the origin of the alignment; for global alignment the origin is the cell $[0, 0]$. This process is illustrated in Figure 2.7 on page 29, where an arrow is used to indicate how each cell value is derived. To find the optimal alignment on the right-hand side of the figure, the series of arrows are followed backwards from the bottom-right cell — the end-point of the alignment — to the origin at $[0, 0]$.

For local alignment, the traceback process starts at the highest scoring cell in the alignment matrix and terminates at the first cell to be encountered with a zero value. In some cases, the alignment matrix may contain several equally high-scoring cells that may be traced to find several different, equally high-scoring alignments. Further, the traceback process may be used to find locally optimal alignments, in different regions of the matrix with a score less than the optimal alignment score, that may also be of interest. We discuss methods for finding locally optimal alignments in Section 2.2.7.

Linear space methods

The basic approach for performing a pairwise alignment with traceback requires $O(l_x l_y)$ space, because a matrix with l_x columns and l_y rows is required to store the traceback information [Durbin, 1998]. Therefore, on a modern workstation the basic approach is unsuitable for aligning very long genomic sequences, such as entire genomes or chromosomes that are often millions of bases in length.

Hirschberg [1975] describes an algorithm that finds the longest common subsequences between two strings using dynamic programming in linear instead of quadratic space. His approach is a divide and conquer strategy, where the dynamic programming matrix is iteratively divided into smaller subsections that can be processed with less memory. The scheme was later adapted by Myers and Miller [1988], who applied Hirschberg's approach to Gotoh's alignment algorithm [1982], resulting in an algorithm for aligning sequences using affine gap costs in linear space. For a more detailed description of linear-space alignment methods see

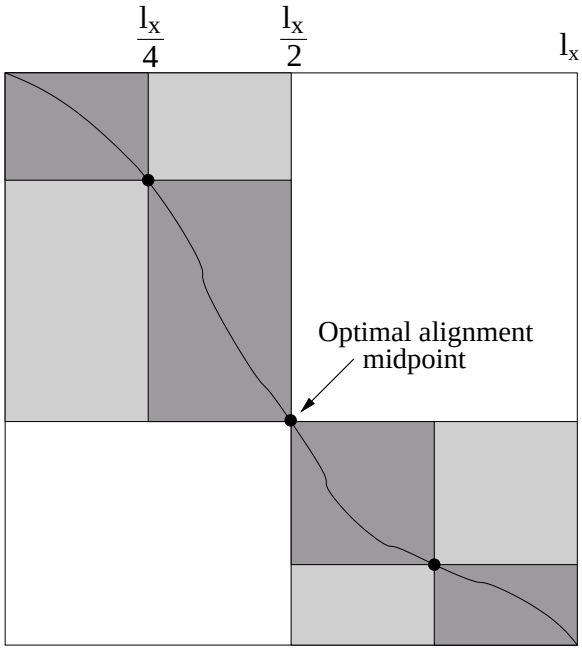

Figure 2.10: Hirschberg's divide and conquer approach. The optimal alignment is represented by a black line running through the alignment matrix from the top-left to the bottom-right. An initial pass calculates the alignment midpoint, where the optimal alignment passes through the column $i = \frac{l_x}{2}$. The alignment matrix is then divided into two sections that can be solved with less memory. The approach is repeated to further divide the matrix.

the survey by Chao et al. [1994].

The divide and conquer strategy by Hirschberg is illustrated in Figure 2.10 and works as follows. First, the score-only alignment algorithm is used to find the alignment midpoint, that is, the point where the optimal alignment passes through the column $i = \frac{l_x}{2}$. The alignment matrix is then divided into four sections, where only the top-left and bottom-right sections shown in grey are processed. These smaller subsections can be solved with less memory. The two greyed sections are solved independently for a portion of the optimal alignment and the results are combined to produce the optimal alignment for the entire two sequences. Further, the division of the alignment matrix can be repeated any number of times to further reduce memory requirements. In practice, the matrix is divided until the regions to be solved are small enough to be processed with the available main-memory.

The scheme requires just $O(l_x)$ space, but takes about twice as long as the naïve scheme to find the optimal pairwise alignment [Myers and Miller, 1988]. Indeed, we believe that

linear space traceback methods are becoming less useful in practice: as sequences grow in number but not in length[9], it is execution speed rather than main-memory requirements that is the limiting factor for many alignment tasks such as searching. Further, schemes that explore a limited region of the alignment matrix, such as those described subsequently in Section 2.3, are commonly employed and already significantly reduce the amount of main-memory required to align sequences.

2.2.7 Locally optimal alignments

In this section we describe methods for identifying locally optimal alignments. Locally optimal alignments are alignments between two sequences with a score less than the optimal alignment score that can provide valuable further insights into the relationship between sequences [Nicholas et al., 2000]. For example, a motif that represents a functional site in one sequence may occur more than once in the other sequence, resulting in multiple high-scoring alignments between the sequences that are of equal interest. Of greatest interest are locally optimal alignments that are high-scoring and do not intersect with the optimal alignment or one another [Waterman and Eggert, 1987; Barton, 1993]. Figure 2.11 shows an example of a pair of sequences that produce an optimal alignment as well as a high-scoring, non-intersecting locally optimal alignment. The path on the left-hand side of the alignment matrix represents the optimal alignment that would be found with a local alignment algorithm such as the Smith-Waterman algorithm described in Section 2.2.4. A second alignment on the right-hand side of the matrix has a lower score but may still provide information about the relationship between the two sequences. The optimal and locally optimal alignments are shown to the right of the alignment matrix.

Several researchers have investigated methods for finding and displaying locally optimal alignments [Waterman and Eggert, 1987; Zuker, 1991; Barton, 1993; Naor and Brutlag, 1993; Chao, 1998]. The first such approaches were described by Waterman and Eggert [1987] and Sellers [1984], which identify the highest-scoring alignment between two sequences as well as all additional, non-intersecting alignments that score above a predefined threshold. However, these original methods were slow because they required multiple passes through the alignment matrix. Altschul and Erickson [1986b] later drew the distintion between two different definitions of local optimality and in Altschul and Erickson [1986a] they provided

[9]The collection statistics presented in Table 7.5 on page 215 indicate that average sequence length in GenBank has not changed greatly over time

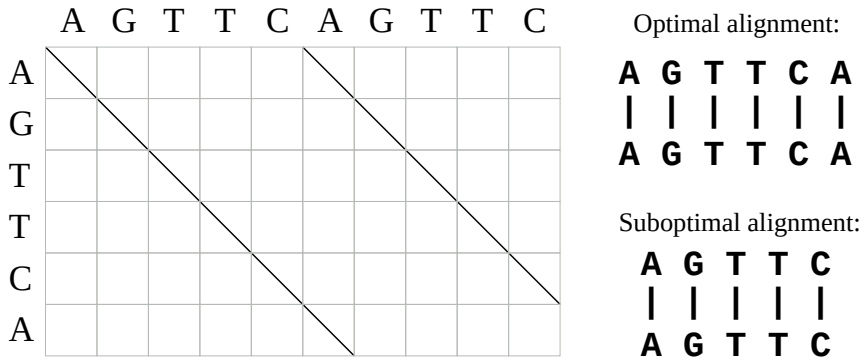

Figure 2.11: Two local alignments between the sequences `AGTTCA` *and* `AGTTCAGTTC` *using scoring scheme with a match score of 1. The path through the left-hand side of the matrix corresponds to the optimal alignment (shown top-right) with a score of 6. The path through the right-hand side of the matrix corresponds to a locally optimal alignment (shown bottom-right) with a score of 5.*

an O(mn) algorithm for finding locally optimal alignments in a single pass of the matrix for one of these two definitions.

2.2.8 Summary

The most frequently employed approach to measure the degree of similarity between biological sequences is to perform a sequence alignment. In this section, we described methods for constructing and scoring sequence alignments. Sequence alignments model the three main evolutionary changes in proteins and DNA; substitutions, insertions, and deletions. Alignments can be performed between pairs of protein sequences, between pairs of DNA sequences, or between a DNA sequence and a protein sequence, although protein comparisons are more sensitive and are generally preferred by biologists.

Sequence alignments can be either global, spanning the entire length of both sequences, or local, spanning only a region of each sequence. We have reviewed methods for performing global and local alignments using dynamic programming. We have also described methods for aligning sequences using both linear and affine gap costs. If the optimal alignment itself is required, in addition to the alignment score, then additional information must be recorded during the alignment process. We have described methods for recording the optimal alignment using traceback, as well as methods for recording traceback that require linear instead of quadratic space. Finally, we have surveyed methods for finding additional, locally

optimal alignments between two sequences.

In the next section, we describe methods that limit the explored region of the alignment matrix to reduce the time taken to align sequences with minimal impact on sensitivity.

2.3 Limited exploration of the alignment matrix

The alignment algorithms described in Section 2.2 consider every possible path through the alignment matrix and are guaranteed to find the optimal alignment between two sequences (with the exception of Durbin's two-state variation [1998] which may produce suboptimal alignments). In this section we describe two schemes that limit the region of the alignment matrix that is explored. The first scheme considers only cells that lie within a fixed diagonal band [Chao et al., 1992] and is used by several alignment tools including the popular FASTA search algorithm [Pearson and Lipman, 1988]. The second scheme does not consider low-scoring regions of the alignment matrix [Zhang et al., 1998a] and is used by the popular BLAST search algorithm [Altschul et al., 1997]. Both schemes provide an heuristic approach that is not guaranteed to find the optimal alignment. In practice, however, these approaches reduce the time taken to perform an alignment with minimal loss in accuracy [Sankoff and Kruskal, 1983; Pearson and Lipman, 1988; Altschul et al., 1997].

2.3.1 Banded alignment

The concept of aligning biological sequences by considering only a fixed diagonal band within the alignment matrix was first proposed by Sankoff and Kruskal [1983] and Spouge [1991]. Chao et al. [1992] later presented an algorithm that incorporated Hirschberg's insight [1975], resulting in an approach to banded alignment in linear rather than quadratic space. The general approach for aligning sequences within a fixed diagonal band works as follows. First, an offset or *diagonal* $d = i - j$ within the alignment matrix of interest is selected, usually due to the presence of a high-scoring ungapped alignment between the two sequences with relative offset d. One of the sequence alignment algorithms described in Section 2.2 is employed to either locally or globally align the sequences, however only cells that lie within a band centered around the relative offset d are considered. The band extends by $\frac{w}{2}$ cells in either direction of diagonal d, where w is the fixed band width. Specifically, only cells in the alignment matrix where $d - \frac{w}{2} \leq i - j \leq d + \frac{w}{2}$ are processed.

The banded alignment approach is illustrated in Figure 2.12. The short black line represents the high-scoring ungapped alignment of interest that is used to select diagonal d that

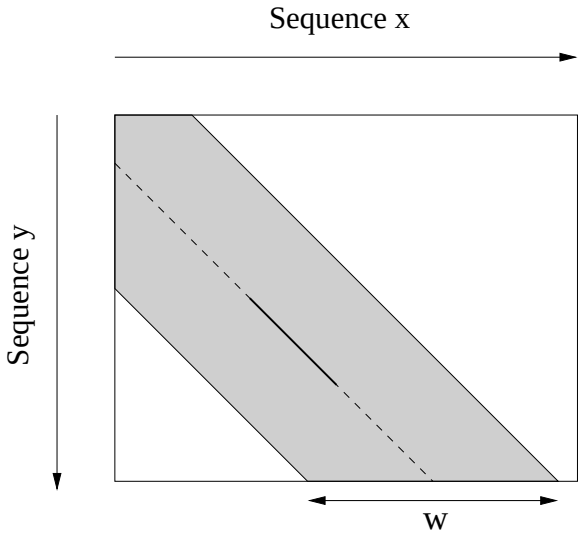

Figure 2.12: Alignment within a diagonal band. The short black line represents a high-scoring ungapped alignment that is used to select the diagonal of interest, shown as a dotted line. The grey region represents the band of width w in the alignment matrix that is processed.

is denoted by a dashed line. The grey area represents the region of the alignment matrix that is processed. The banded approach fails to identify alignments that are not entirely contained within the banded region; this may be due to a large number of insertions that shift the alignment to a diagonal outside of the processed region. For example, an alignment starting on diagonal d that contains more than $\frac{w}{2}$ insertions but no deletions in one of the sequences will not be identified by the banded approach. When used with a carefully chosen value for w, however, this approach is reasonably effective at reducing the time taken to align sequences with minimal impact on the level of sensitivity to homologous relations because optimal alignments between related sequences rarely contain a large number of gaps [Sankoff and Kruskal, 1983].

2.3.2 Dropoff alignment

The *dropoff* technique [Altschul et al., 1997; Zhang et al., 1998a] is another successful approach for reducing the number of cells processed during a sequence alignment. The dropoff approach limits the region of the alignment matrix that is considered and is used in conjunction with the global alignment recurrence relations described in Section 2.2.3. These recurrence relations are not only applicable to global alignment; they can also be used to

perform seeded alignment, where the alignment origin is a fixed point [i,j] that is not necessarily the start of both sequences [Altschul et al., 1997]. We discuss seeded alignment in more detail in Section 3.1.3.

The dropoff approach works as follows. During alignment, the value of the highest scoring cell in the dynamic programming matrix processed so far, h, is recorded. Each row or column in the alignment matrix is processed up until the point where the cell value is less than the highest score observed minus the value of the dropoff parameter X, that is, where $B(i,j) < h - X$. As a result, only regions of the alignment matrix with a score that is greater than the best score observed so far minus X are considered. Cells at the boundry of the explored region are initialised to $-\infty$. Figure 2.13 illustrates how this technique limits the area of the matrix to be processed. The alignment between two pairs of sequences is illustrated; a pair of related lysozyme sequences are aligned on the left, while a lysozyme and unrelated influenza virus sequences are aligned on the right. The grey shaded regions represent cells in the alignment matrix that are process during alignment.

Like banded alignment, the dropoff approach is an heuristic that is not guaranteed to find the optimal alignment between two sequences. Alignments that contain low-scoring regions, that is, regions with a collective score below $-X$, are overlooked. In practice, however, the dropoff scheme is effective in reducing the computational cost of alignment, while being sensitive in finding homologous sequences because optimal alignments between related sequences do not often contain massively low-scoring regions [Altschul et al., 1997]. The approach is highly effective at reducing computational costs when aligning highly dissimilar sequences, because lower alignment scores further limit the region of exploration. This is illustrated in Figure 2.13; a large portion of the alignment matrix on the left is explored when aligning the closely related lysozyme sequences, however the alignment of two unrelated sequences on the right results in far more limited area of exploration. We believe that the dropoff technique is more effective than banded alignment because it is adaptive, varying both the direction and size of the region to be computed based on the alignment scores determined so far. The dropoff approach is therefore a highly effective filtering component of the BLAST genomic search algorithm, because the vast majority of pairwise alignments are between dissimilar sequences as shown in Section 3.1.3. We discuss the application of the dropoff approach to search, and to the BLAST algorithm in particular, further in Section 3.1.3.

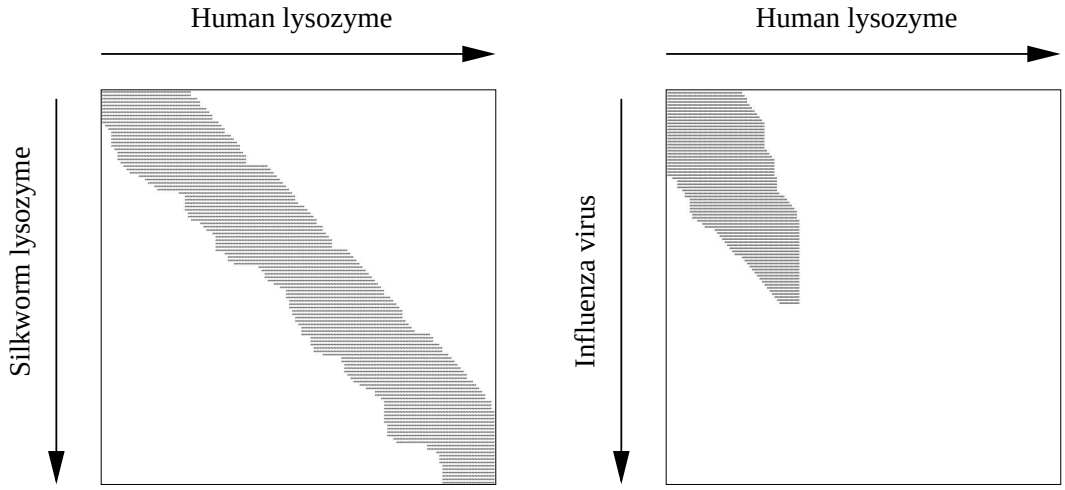

Figure 2.13: Pairwise alignment using the dropoff approach, where the grey shaded region represents the portion of the alignment matrix that is explored. The matrix on the left is used to globally align the human lysozyme sequence (PDB accession 1LZ1) to the closely related silkworm lysozyme sequence (PDB accession 1GD6, chain A). The matrix on the right is used to align the human lysozyme sequence to an unrelated influenza virus protein sequence (PDB accession 1AA7, Chain A). Our own implementation of the dropoff heuristic, the BLOSUM62 substitution matrix, open gap penalty of 11, extension penalty of 1, and dropoff parameter $X = 38$ *were used.*

2.3.3 Summary

In this section we have described two methods for limiting the explored region of the alignment matrix. The first method only processes cells that lie within a banded region centered around a diagonal of interest. The second method uses a dynamic approach where the explored region is bounded by low-scoring cells. By limiting the number of cells processed, these approaches reduce the time taken to perform an alignment. Neither approach is guaranteed to find the optimal alignment between two sequences, however both schemes are reasonably sensitive to homologous relationships as we show in Section 3.1. The dropoff approach is arguably the more effective of the two schemes because the explored region is adaptive and varies in direction and size based on the degree of similarity between the sequences.

In the next section, we discuss substitution matrices that are commonly used to score the alignment of amino acids in protein comparisons.

2.4 Substitution matrices

Substitution matrices (also known as *scoring matrices* or *mutation data matrices*) are commonly used to score the conservation and substitution of amino acids in protein sequence alignments. For an alphabet of size a, a matrix of size $a \times a$ provides an alignment score for every possible pair of amino acids [Dayhoff et al., 1978; Henikoff and Henikoff, 1992].

Substitution matrices quantify the likelihood of a given residue mutating into another during the evolutionary process. Mutations between amino acids that conserve physical or chemical properties are more likely during evolution because they are less likely to change the structural or functional characteristics of the protein as a whole [Nicholas et al., 2000]. Scoring matrices assign a high score for similar residue pairs that are more likely to be interchanged and a low score for dissimilar residue pairs that are less likely to be interchanged [Dayhoff et al., 1978; Henikoff and Henikoff, 1992]. Residue pairs where the two residues are the same, and the residue is conserved, generally score highly.

In this section we describe methods for the construction and usage of the two most popular series of mutation data matrices, the PAM series [Dayhoff et al., 1978] and the BLOSUM series [Henikoff and Henikoff, 1992]. Both series of matrices are based on the same fundamental theory, where alignments between related proteins are used to calculate the probability of seeing residue pairs aligned, given that the residues are homologous. Let p_i be the frequency of occurrence of residue i in protein sequences. Next, we define q_{ij} to be the frequency with which amino acids i and j are aligned within alignments of related sequences between homologous proteins. For the construction of most matrices, including the BLOSUM and PAM series, we assume that $q_{ij} = q_{ji}$. The probability P_{ij} of seeing residues i and j aligned, given that the residues are homologous, is calculated as $P_{ij} = q_{ij}/p_i p_j$. The value $s(i, j)$ that is recorded in the substitution matrix and used to score alignments is the base two logarithm of the probability P_{ij}, that is $s(i, j) = log_2(P_{ij})$. Logarithm of odds are recorded in the matrix instead of the odds themselves to reduce the computation required to score alignments; log values can be added to calculate statistical significance whereas the odds themselves must be multiplied. In practice, scores are multiplied by a constant factor (such as two for the BLOSUM series of matricies) and rounded to the nearest integer before being recorded in the scoring matrix, so that integer values can be used for faster computation yet still provide a sufficiently fine scale. The key difference between the PAM and BLOSUM matrices is the approach used to estimate the homologous amino acid alignment frequencies, q_{ij}, during construction. We discuss those differences next.

2.4.1 PAM matrices

The PAM series of matrices were developed by Dayhoff and co-workers [1978]. To create the matrices, a set of global alignments of closely related proteins were carefully constructed. The alignments included gaps and spanned both highly conserved and more variable regions. The alignments were then used to estimate transition frequencies for highly similar sequence pairs containing only 1 in 100 amino acid substitutions. These frequencies formed the basis of the PAM1 matrix, which was used to extrapolate other matrices in the PAM series by successively multiplying a matrix containing the residue mutation probabilities by itself. Using this method, a series of matrices suitable for aligning sequences across a range of evolutionary distances were constructed. The PAM250 matrix, for example, was constructed by multiplying the mutation probability matrix for PAM1 by itself 250 times. The matrix is suitable for aligning sequences that diverge by 250 PAM units, that is, sequences with approximately 250 point mutations per 100 residues. PAM values greater than 100 are possible because a single amino acid may mutate several times.

2.4.2 BLOSUM matrices

Several years after the popular PAM matrices were constructed, the BLOSUM series of matrices was proposed [Henikoff and Henikoff, 1992]. Rather than align closely related sequences and use extrapolation to model more distant relationships, transition frequencies for the BLOSUM series were observed directly from clusters of more distantly related proteins. This approach was feasible thanks to an increase in the quantity of available protein sequence data since the PAM matrices were devised in the 1970's.

To construct the BLOSUM matrices, mutation frequencies were observed from alignments in the BLOCKS database [Henikoff and Henikoff, 1991]. The database contains protein regions that have been clustered into highly conserved groups and aligned without gaps. An example of an entry in the BLOCKS database is shown in Figure 2.14. The first twelve sequence regions in the block are shown, and the regions form an ungapped multiple alignment. When constructing the BLOSUM series, mutation frequencies were observed from pairs of residues in the same column of the alignment.

The BLOSUM series of matrices were constructed for a range of evolutionary distances using the BLOCKS collection. This was achieved by clustering sequence pairs with at least X% identity, and then counting only substitutions between clusters, weighted by the size of each cluster. As a result, the value X represents the level of divergence the matrix is designed

```
P97773              (202)  SFTNGETSPTVSMAELEHLAQN
ROR4_HUMAN|P45445   (203)  SFTNGETSPTVSMAELEHLAQN
ROR3_HUMAN|P35399   (283)  SFTNGETSPTVSMAELEHLAQN
ROR2_HUMAN|P35398   (291)  SFTNGETSPTVSMAELEHLAQN
ROR1_MOUSE|P51448   (258)  SFTNGETSPTVSMAELEHLAQN
ROR1_HUMAN|P35397   (258)  SFTNGETSPTVSMAELEHLAQN
ROR4_MOUSE|P70283   (194)  SFTNGETSPTVSMAELEHLAQN
RORB_RAT|P45446     (197)  SFNNGQLAPGITMSEIDRIAQN
Q98934              (197)  SFNNGQLAPGISMTEIDRIAQN
RORB_HUMAN|Q92753   (197)  SFNNGQLAPGITMTEIDRIAQN
Q61027              (253)  SFCSAPEVPYASLTDIEYLVQN
RORG_MOUSE|P51450   (253)  SFCSAPEVPYASLTDIEYLVQN
```

Figure 2.14: A portion of entry IPB003079A in the BLOCKS database containing nuclear receptor proteins. Each row contains the sequence name, a start offset in brackets, and a subsequence beginning at that offset.

to detect. The popular BLOSUM62 matrix, for example, was constructed by clustering sequences with at least 62% identity. Larger values of X are suitable for detecting less distant relationships and smaller values are suitable for detecting more distant relationships.

Studies have shown that the BLOSUM series provides better accuracy than PAM matrices [Henikoff and Henikoff, 1993; Pearson, 1995]. The BLOSUM62 matrix, shown in Figure 2.15, provides the best overall accuracy [Henikoff and Henikoff, 1993] and is the default choice for many homology search tools. Positive values in the matrix represent amino acid pairs that are likely to be exchanged through mutation, and negative values represent unlikely mutations. For example, lysine (`L`) and isoleucine (`I`) produce an alignment score of 2 which indicates the pair of amino acids are frequently substituted for one another. The likely substitution of these two residues is supported by their shared aliphatic and hydrophobic properties [Taylor, 1986] and their co-occurrence in the fourth column from the right-hand side of the alignment of nuclear receptor proteins in Figure 2.14.

2.4.3 Other approaches

In addition to the popular BLOSUM and PAM series, other less popular mutation data matrices have been proposed. They include matrices that are based on the physico-chemical properties of proteins and matrices that encapsulate the degree of similarity between the codons that encode for each amino acid. However, none of these approaches have proved as successful as methods based on observed frequencies of mutation in related proteins [Johnson

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

Figure 2.15: BLOSUM62 substitution matrix.

and Overington, 1993]. Altschul [1993] describes a method for aligning sequences using a set of substitution matrices that have been optimised for varying evolutionary distances instead of a single matrix, however this approach is not widely employed. Although substitution matrices are most commonly used for protein alignment, matrices have also been proposed for nucleotide sequence comparisons; States et al. [1991] and Chiaromonte et al. [2002] describe substitution matrices that scores transitions (interchanges between the pyrimidines C and T, or the purines A and G) higher than transversions (interchanges between a pyrimidine and a purine) based on the observation that transitions occur more frequently in related genomes.

2.4.4 Summary

Substitution matrices are commonly used to score protein sequence alignments. The matrices provide an alignment score for each pair of amino acids that reflects the likelihood of mutation between the residues. In this section, we have described the basic theory behind substitution matrices and approaches used to construct them. The two popular series of matrices are the PAM series and the BLOSUM series. The PAM matrices were constructed using mutation

frequencies from a small collection of closely related and carefully aligned proteins. The frequencies were extrapolated to derive matrices for a range of evolutionary distances. The BLOSUM matrices were constructed from alignments in the BLOCKS database and mutation frequencies were observed directly from alignments across a range of evolutionary distances. In addition to the BLOSUM and PAM series, other methods for constructing substitution matrices have been proposed, however they have proved less successful.

2.5 Conclusion

In this chapter, we have provided a background to genomic data and biological sequence comparison. Publicly available databases such as GenBank contain a mass of genomic sequence data and these collections have exhibited exponential growth over the past three decades. Molecular biologists, geneticists, and other life scientists use sequence comparison methods as their first step for discovery of information about unknown or poorly annotated genomic sequences. Therefore, sequence comparison is an important tool that aids in the determination of the structure, function and evolutionary origin of proteins and DNA. We have described algorithms by Sellers [1974] and Smith and Waterman [1981] for globally and locally aligning biological sequences using dynamic programming. We have also presented algorithms that support the two most commonly used gap penalty systems: linear gap costs and affine gap costs.

The basic alignment algorithms record only the highest alignment score for two sequences. If the optimal alignment itself is required, additional traceback information must be recorded. We have described the basic traceback approach as well as a divide-and-conquer approach that requires linear instead of quadratic space. We have also briefly described methods for recording locally optimal alignments that may provide further insight into relationships between sequences.

Pairwise sequence comparison is computationally expensive and requires quadratic time in the length of the two sequences. We have presented two methods that allow faster comparisons by not calculating scores for some cells in the alignment matrix; we refer to these methods as banded alignment and dropoff alignment. The banded and dropoff approaches are able to reduce the time taken to align sequences with only minimal impact on the degree of sensitivity to homologous relationships. We have also described the theory and application of substitution matrices for scoring alignments. We have presented an overview of the methods used to construct the popular PAM [Dayhoff et al., 1978] and BLOSUM [Henikoff

and Henikoff, 1992] series of matrices, which model mutation rates between all possible pairs of amino acids.

In the next chapter, we focus on methods for searching sequence databanks. We begin with descriptions of the popular heuristic search algorithms such as FASTA [Pearson and Lipman, 1988] and BLAST [Altschul et al., 1990; 1997] as well as index-based approaches such as CAFE [Williams and Zobel, 1996; 2002], PATTERNHUNTER [Ma et al., 2002; Li et al., 2004] and BLAT [Kent, 2002]. We then describe distributed search schemes and iterative algorithms such as PSI-BLAST [Altschul et al., 1997] and SAM [Karplus et al., 1998]. We also discuss important issues related to database search such as methods for assessing the statistical significant of alignments, the effect of low-complexity regions in collection sequences, and methods for assessing the accuracy of search tools. Finally, we present the problem of redundancy in genomic collections and discuss methods for identifying and managing redundancy.

Chapter 3

Searching Genomic Databases

A powerful tool for the discovery of information about sequences is *homology search*, where genomic collections are searched for high-scoring alignments. In some cases, it can rapidly accelerate gene discovery, allowing understanding of biochemical role, chemical function, and physical structure. For example, querying a database with an unknown sequence can in some cases reveal related, well-annotated sequences, providing an understanding of the poorly known sequence. In turn, this can facilitate focused wet lab experimentation, reducing the cost and time taken in tasks as diverse as drug discovery, cancer research, and crop production.

In this chapter, we survey successful existing approaches to homology search. We begin with an overview of exhaustive algorithms such as SSEARCH, FASTA [Pearson and Lipman, 1988] and BLAST [Altschul et al., 1990; 1997]. We provide a detailed description and analysis of each stage of the popular BLAST algorithm, and present usage data for the tool. In Section 3.2 we discuss other approaches to search such as index-based approaches, discontinuous seeds, distributed search, profiles, and iterative search algorithms. We conclude that BLAST is the most versatile, fast and accurate search scheme, motivating our improvements to the algorithm in Chapters 4, 5, 6 and 7.

In Section 3.3 we discuss several issues relating to homology search, beginning with an overview of methods for assessing the statistical significance of pairwise alignments. Next, we describe filtering schemes that mask low-complexity regions in sequences, and methods for assessing the retrieval effectiveness of homology search tools. Finally, we discuss redundancy in genomic data banks and methods for pruning near-identical sequences.

3.1 Popular search algorithms

In this section, we describe the three popular genomic search algorithms: SSEARCH, FASTA and BLAST. All three algorithms use an exhaustive approach to search, where the query is compared directly to each sequence in the collection. The SSEARCH algorithm, which stands for *Smith-Waterman search*, uses the Smith-Waterman local alignment algorithm [1981] described in the previous chapter to compare the query to each collection sequence. FASTA and BLAST are heuristic approaches that approximate the Smith-Waterman algorithm by employing a series of filtering stages to eliminate collection sequences unlikely to produce a high-scoring alignment. At the final stage, both schemes perform local alignment in a similar manner to SSEARCH. Although slightly less accurate than SSEARCH, the heuristic schemes provide much faster search.

3.1.1 SSEARCH: Smith-Waterman search

The SSEARCH algorithm compares a query sequence to each sequence in a collection and displays alignments that are statistically significant to the user, sorted in order of score[1]. Efficient implementations of SSEARCH, such as the version distributed with the FASTA [Pearson and Lipman, 1988] package of search tools, use the following approach to search a collection. First, the query sequence is aligned with each collection sequence using the efficient score-only version of the Smith-Waterman algorithm described in Section 2.2.4. Next, the statistical significance of each alignment, represented by an E-value, is calculated using the methods described in Section 3.3.1. Sequences with an alignment E-value below the user specified cutoff are re-aligned using the space-efficient traceback method described in Section 2.2.6. The final list of alignments are sorted by score and displayed to the user.

The Smith-Waterman search approach is guaranteed to find optimal alignments between the query and collection and is the most sensitive of the pairwise alignment algorithms [Shpaer et al., 1996; Brenner et al., 1998; Chen, 2003]. However, while it is accurate it is not fast, and without specialised hardware a database search using the Smith-Waterman local alignment is impractical for large genomic collections. On a general-purpose workstation, a search of the GenBank NR protein database with a typical query takes just over 1 hour, while a search of the GenBank NR nucleotide database with a typical query takes around 4.5 days, as shown in Table 3.1. This has necessitated heuristics for efficient search on desktop workstations, and to enable institutes to provide search services to large numbers of users.

[1] Based on our analysis of FASTA version 3.4, released September 2, 2005

Tool	Protein	Nucleotide	ROC_{50}
SSEARCH	64.45	6,473.51	0.467
FASTA	4.45	45.58	0.436
BLAST	1.18	7.14	0.415

Table 3.1: Typical protein and nucleotide search times in minutes for SSEARCH*,* FASTA*, and* BLAST*. Runtimes were recorded by searching the GenBank non-redundant protein database with a human α_1-antitrypsin protein sequence (GI accession 28966, 418 amino acids in length) and the GenBank non-redundant nucleotide database with an mRNA nucleotide sequence that encodes lysozyme in human (GI accession 4557893, 1,487 bases in length). Experiments were conducted on a Intel Pentium 4 2.8GHz workstation using version 3.4 of* FASTA *and* SSEARCH *and version 2.2.11 of* NCBI-BLAST*. The ROC_{50} measure of search accuracy, which is described in Section 3.3.3, is also reported for each tool based on the analysis by Chen [2004].*

We describe two popular heuristic approaches to search, FASTA and BLAST, next.

3.1.2 FASTA

The FASTP algorithm [Pearson and Lipman, 1985] — which was later revised as FASTA [Pearson and Lipman, 1988] — was the first successful heuristic approach to local alignment. The FASTA algorithm is slightly less accurate than the exhaustive Smith-Waterman search [Shpaer et al., 1996; Brenner et al., 1998; Chen, 2003] but is roughly 14 times faster for a typical protein search and over 140 times faster for a typical nucleotide search, as shown in Table 3.1. The algorithm was popular amongst biologists in the late 1980s and early 1990s when genomic collections were considerably smaller and FASTA was the only popular tool to generate gapped alignments. Today, a FASTA search of large genomic collection is less practical, with a search of the GenBank non-redundant nucleotide database taking over 45 minutes on a modern workstation (Table 3.1). However, the FASTA algorithm is still used to search smaller collections when a greater level of accuracy than that offered by BLAST is required.

The FASTA algorithm consists of four stages that combine to provide efficient yet sensitive search of genomic collections. The first stage uses the approach of Wilbur and Lipman [1983] to identify exact matches, or *hits*, of length W between the query and sequences in the collection. For protein comparison, a word length of $W = 1$ or $W = 2$ amino acids is

typically used; the shorter word length of $W = 1$ provides better sensitivity but roughly 3 to 4 times slower search than the longer word length of $W = 2$ [Pearson, 1996]. For nucleotide comparisons, the word length typically ranges between 4 and 6 bases.

To identify hits, fixed-length overlapping subsequences of length W are extracted from the query sequence q and the each collection sequence s. For example, suppose $W = 3$ and the following short sequence is processed: `ABCDEFGHIJKLMNOPQ`. The *words* extracted from the sequence are as follows: `ABC`, `BCD`, `CDE`, `DEF`, `EFG`, `GHI`, `HIJ`, and so on. In the first stage, all sequences are exhaustively, sequentially processed from the collection being searched, that is, each sequence is read from the database, parsed into words of length W, and matched against the query.

To match query and collection sequence words, FASTA uses a lookup table — as illustrated in Figure 3.1 — constructed from the query sequence. The table contains an entry for every possible word, of which there are a^W for an alphabet of size a; the example in Figure 3.1 illustrates the table for an alphabet of size $a = 3$, with three symbols `A`, `B`, and `C`, and a word length $W = 2$. Associated with each word in the table is a list of query positions that denote the zero or more locations where that word appears in the query sequence. In the example table, the word `BC` occurs at query positions 2 and 5. The query sequence is shown at the base of the figure, where the word `BC` occurs once starting at position 2 and once starting at position 5.

The search process using the lookup table is straightforward. First, the current collection sequence is parsed into words. Second, each collection sequence word is located in the query lookup table. Third, for each matching position in the query, a hit is recorded. Last, when a hit is recorded, a pair of positions, i, j, that identify the match between the query and collection sequence are passed to the second stage.

In the second stage, regions from the query sequence and current collection sequence that share a high density of identical words are identified and realigned using a simple ungapped alignment routine and a substitution matrix. To do this, the relative offset or *diagonal* of each of the hits identified in the first stage is calculate as $d = i - j$. Hits that occur on the same diagonal represent matches that may form part of a single alignment without gaps. Next, the ten regions along a diagonal that contain the highest density of hits are identified and rescored using a more fine-grain ungapped alignment. The alignment considers runs of identities less than W in length as well as conservative substitutions that are overlooked in the first stage, but does not consider insertions or deletions.

The first two stages of the FASTA algorithm are illustrated in the top half of Figure 3.2.

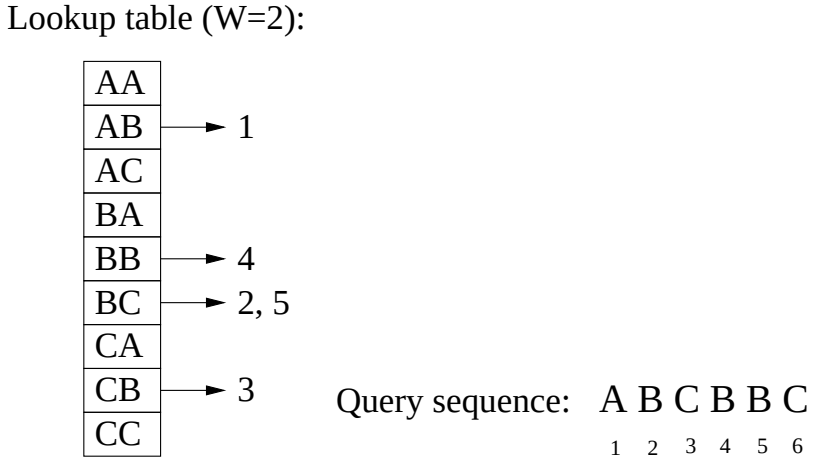

Figure 3.1: The lookup table used by FASTA *to identify hits. In this example, the alphabet contains 3 characters:* `A`*,* `B`*, and* `C`*. The lookup table on the left is constructed from the query sequence on the right using a word size* $W = 2$*. The table contains* $a^W = 3^2 = 9$ *entries, one for each possible word, and each entry has an associated list of query positions. The query positions identify the starting positions of that word in the query.*

The figure shows four matrices that are used to compute the similarity between two sequences. The width of each matrix is the length l_1 of the query sequence q, that is, there is one column for each residue in the query sequence. Similarly, the height of each matrix is the length l_2 of the collection sequence s that is being considered. Therefore, each cell in the matrix represents an intersection between a residue from each sequence, and is used to tabulate a score on the optimal evolutionary pathway that considers those two residues. This process is discussed in more detail in Section 2.2. At the top-left, short diagonal lines represent exact matches between the query and collection sequence of length W. The ten diagonal regions with the highest density of matches are then realigned as shown by the black lines at the top-right of the figure.

In the third stage, high-scoring regions from the second stage are combined to construct a longer alignment that contains gaps. An approximation of the optimal alignment score is derived by linking the regions, then summing the alignment score for each region and subtracting a special joining penalty that is analogous to a gap penalty. If the resulting score is greater than a predefined threshold then the collection sequence is passed on to the fourth and final stage. The third stage is illustrated at the bottom-left of Figure 3.2, where the black lines represent high-scoring regions from the second stage and the grey lines represent

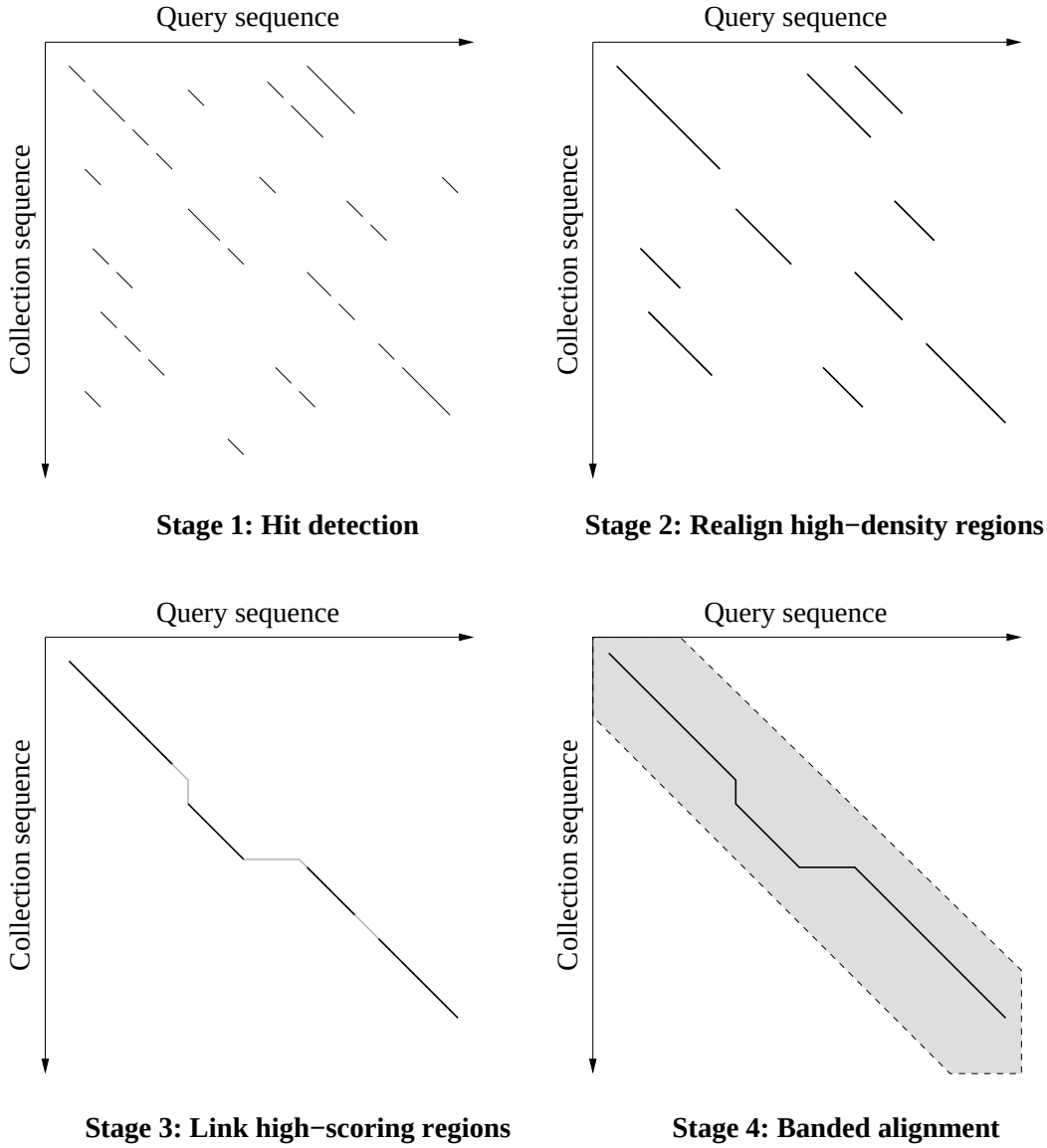

Figure 3.2: The four stages of the FASTA *algorithm. In the first stage, short matches of a fixed length W between the query and collection sequence are identified. In the second stage, the ten diagonal regions with the highest density are realigned. In the third stage, high-scoring regions are linked in an attempt to increase the alignment score. In the final stage, a banded Smith-Waterman alignment is performed.*

the links between them.

In the final stage, the Smith-Waterman algorithm [1981] is used to find the optimal local alignment between the query sequence and the collection sequence. The banded approach described in Section 2.3.1 is used to explore only a limited region of the alignment matrix, where the band is centered around the highest scoring region identified in the second stage. Statistically significant alignments are ranked in order of score and displayed to the user. We discuss the methods used by FASTA to calculate the statistical significance of alignments in Section 3.3.1.

3.1.3 BLAST: Basic Local Alignment Search Tool

Since 1990, BLAST [Altschul et al., 1990] has been the most popular heuristic local alignment tool and in widespread use, first as the BLAST1 suite of tools, and since 1997 as the BLAST2 tools [Altschul et al., 1997]. It is used to evaluate over 120,000 homology search queries each day [McGinnis and Madden, 2004] at the popular NCBI website[2], and is installed and maintained in almost all medium- to large-scale molecular biology research facilities. We estimate that the widespread use of the tool within universities, research centres, and commercial enterprises worldwide results in millions of queries being conducted daily. Indeed, the 1997 BLAST paper [Altschul et al., 1997] has been cited over 10,000 times[3] in electronically-available manuscripts.

BLAST has also been widely adapted for different platforms, architectures, and tasks. Several hardware-dependent implementations are available and many variations of its algorithms have been published including PSI-BLAST [Altschul et al., 1997], MEGABLAST [Zhang et al., 2000], PHI-BLAST [Zhang et al., 1998b] and IMPALA [Schaffer et al., 1999]. We discuss these variation of the BLAST algorithm in more detail later in this chapter.

The popularity of BLAST stems from its speed and accuracy. Several studies have shown BLAST to be slightly less accurate than FASTA [Shpaer et al., 1996; Park et al., 1998; Brenner et al., 1998; Chen, 2003; 2004], however a typical search of the GenBank non-redundant protein or nucleotide database takes 1.18 or 7.14 minutes respectively, as shown in Table 3.1 — this is between 3.7 and 6.4 times faster than FASTA and roughly 50 to 600 times faster than an exhaustive Smith-Waterman search.

The BLAST algorithm is a four-stage process that is both efficient and effective for searching genomic databases. The steps progressively reduce the search space, but each is more

[2]See `http://www.ncbi.nlm.nih.gov/`

[3]See: `http://scholar.google.com/`

fine-grain and takes longer to process each sequence than the previous. The first stage involves identifying matches of a fixed length W between the query and collection sequences called *hits*. In the second stage, these hits form the basis of ungapped alignments that do not consider the more computationally expensive insertion and deletion events. In the third stage, gapped alignments are performed between sequences using a similar approach to the alignment algorithms described in Section 2.2. In the final stage the alignments themselves are recorded using traceback and displayed to the user. Table 3.2 shows the average time spent performing each stage of the algorithm.

Before searching a collection, the `formatdb` tool that is distributed with NCBI-BLAST is used to convert the database from the human-readable FASTA format to a binary representation that is read by BLAST[4]. The collection sequences are converted to a new format that is more easily processed during search, where each of the twenty amino acids and each of the four nucleotide bases are represented by a distinct numeric code. Additional codes are used to represent the ambiguous characters described in Section 2.1. Further, nucleotide sequences are converted to a compressed representation that is discussed in more detail in Section 3.1.3.

The two most commonly used variants of the BLAST algorithm are BLASTP, which is used to search a protein database with a protein query, and BLASTN, which is used to search a nucleotide database with a nucleotide query (see Section 3.1.3 for a more thorough analysis of BLAST usage). We begin this section with a description of the BLASTP algorithm based on our analysis of version 2.2.11 of NCBI-BLAST that was released in June 2005 and closely resembles the approach presented in the 1997 BLAST paper [Altschul et al., 1997]. We then describe how the BLASTN approach for nucleotide search differs from the approach used for protein search. Next, we describe some popular variations of the BLAST algorithm including methods for performing translated searches between a protein query and nucleotide collection and vice versa. Finally, we present usage statistics for the online BLAST service offered by the NCBI.

Stage 1: Hit detection

In the first stage, BLAST identifies matches of a fixed length W between the query and sequences in the collection using an approach similar to the first stage of FASTA that was described in Section 3.1.2. Each sequence s from the collection being searched is retrieved

[4]See: `ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools/docs/README`

Stage	Task	Percentage of overall time	
		Protein	Nucleotide
1	Find high-scoring short hits	37%	85%
2	Identify pairs of hits on the same diagonal	18%	—
2	Perform ungapped extensions	13%	5%
3	Perform gapped extension	30%	9%
4	Perform traceback and display alignments	2%	1%

Table 3.2: Average runtime for each stage of the BLAST *algorithm for 100 randomly-selected sequences from the GenBank NR protein or nucleotide database searched against the entire database. Experiments were conducted using the NCBI implementation of* BLAST *with default parameters and no query filtering.*

and compared to the query sequence q using the algorithm of Wilbur and Lipman [1983]. This identifies exact matches (or *hits*) between fixed-length overlapping subsequences (or *words*) of length W extracted from the query and collection sequences.

There are two important differences between the hit detection process used by BLAST and the process used by FASTA. First, a longer word length is typically used; whereas FASTA identifies hits using a word length of $W = 1$ or $W = 2$ for protein comparison, a word length of $W = 3$ is typically used by BLAST [Altschul et al., 1997]. Second, the first stage of BLAST considers inexact, high-scoring word matches in addition to exact matches as hits. A match between a pair of words is considered high scoring if the match scores above a given threshold T when scored using a mutation data matrix. Two words that score above the threshold are referred to as *neighbours*. BLAST uses default parameter values of $W = 3$ and $T = 11$ [Altschul et al., 1997].

To identify high-scoring word matches between the query and collection sequences, BLAST uses a different approach to that employed by FASTA and described in Section 3.1.2 to construct the lookup table — this is shown in Figure 3.3 — that is dependent on the query sequence and a substitution matrix. We use the same query sequence, alphabet, and word length $W = 2$ as those used to illustrate the FASTA lookup table in Figure 3.1 to illustrate the BLAST lookup table. Instead of recording the position of exact matches only, each query position in a list denotes the offset of a word of length W in the query sequence that either matches or scores highly when aligned to the word represented by that list. For example, the word `AB` has hits at query positions 1 and 3, because the exact word `AB` occurs at position 1

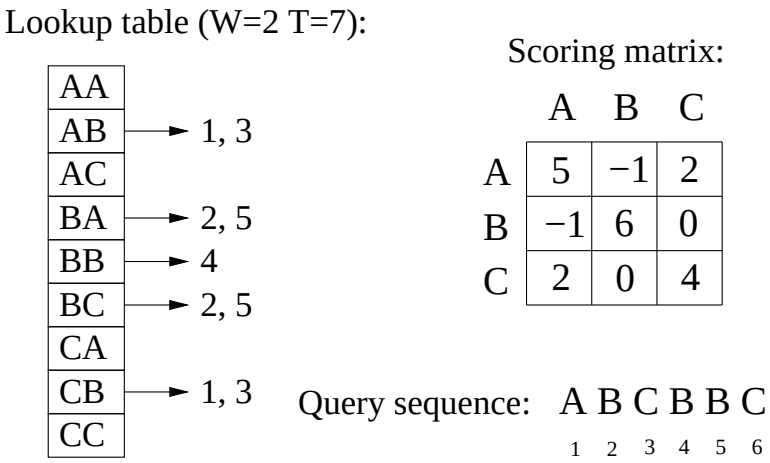

Figure 3.3: The lookup table used by BLAST *to identify hits. In this example, the alphabet contains 3 characters:* `A`*,* `B`*, and* `C`*. The lookup table on the left is constructed using the example scoring matrix and query sequence on the right with a word size* $W = 2$ *and threshold* $T = 7$*. The table contains* $a^W = 3^2 = 9$ *entries, one for each possible word, and each entry has an associated list of query positions. The query positions identify the starting position of words in the query that score above* T *when aligned to the collection sequence word.*

and a close-matching hit (`CB`) occurs at position 3. We describe the design of the lookup table in more detail in Section 5.1.

The search process used by BLAST to identify hits between the query and collection is the same process used by FASTA. The collection sequence is parsed into overlapping words of a fixed length W, and each word is used to access an entry in the lookup table. Each query position in the table represents a single hit and provides the offset into the query sequence i for that hit. The offset i from the query sequence q and offset j from the collection sequence s of each hit is passed to the second stage of the BLAST algorithm.

The use of neighbours to identify matching regions between the query and the collection drastically increases the number of hits, and as a result the overall sensitivity and runtime of BLAST. To illustrate, we conducted a simple experiment with 100 queries randomly extracted from the GenBank non-redundanct (NR) protein database. When default parameters of $W = 3$ and $T = 11$ were employed, on average 9.14 neighbours were generated for each word in those queries.

To examine the performance of this first stage for protein comparisons, and produce the results shown in the third column of Table 3.2, we carried out a simple experiment.

We randomly extracted 100 sequences from the GenBank NR protein database and then searched the entire database using each as query. We found that the first stage consumes on average around 37% of the total time for all four stages, and that an average of 229 hits were found per collection sequence; almost all sequences had at least one hit. The version of the non-redundant protein database we used for this experiment was downloaded on 30 June 2004 and contains 1,873,745 sequences in around 622 megabytes of sequence data. The experiment was carried out on an Intel Pentium 4 2.8GHz workstation with one gigabyte of main-memory while the machine was under light-load using NCBI-BLAST version 2.2.8. The best of three runs for each query was recorded.

Stage 2: Ungapped alignment

The second stage determines whether two or more hits h of length W could form the basis of a local alignment that does not include insertions or deletions of residues. To determine this, the *diagonal* d of each of the hits is determined by computing the difference in the query and collection sequence offsets, $d = j - i$. If two hits, $h[i_1, j_1]$ and $h[i_2, j_2]$, are found to occur on the same diagonal (since $j_1 - i_1 = j_2 - i_2$), are not overlapping, and $i_1 - i_2$ is less than a constant A, then an *ungapped extension* is attempted. The parameter A, which defaults to 40, influences the accuracy of BLAST and is one of many parameters that can be tuned in the algorithm.

An ungapped extension links the two hits by computing scores for matches or substitutions between the hits; it ignores insertion and deletion events, which are more computationally expensive to evaluate and calculated only in the third and fourth stages. After linking the hits, the ungapped extension is processed outward until the score decreases by a pre-defined threshold, using a dropoff approach that is similar to the scheme described in Section 2.3.2. High-scoring ungapped alignments resulting from this process are then passed on to the third stage.

The left side of Figure 3.4 illustrates the first two stages of the BLAST algorithm. The figure shows the matrices that are used to compute the alignment between the query sequence q and a collection sequence s similar to those illustrated in Section 3.1.2. The short black lines in Figure 3.4 represent high-scoring hits of length W that match between both sequences. That is, each such black line represents the beginning of the shortest possible evolutionary pathway (of minimum length W) that is considered by the BLAST algorithm. In the figure, there are two cases where two hits are located on the same diagonal less than the maximum

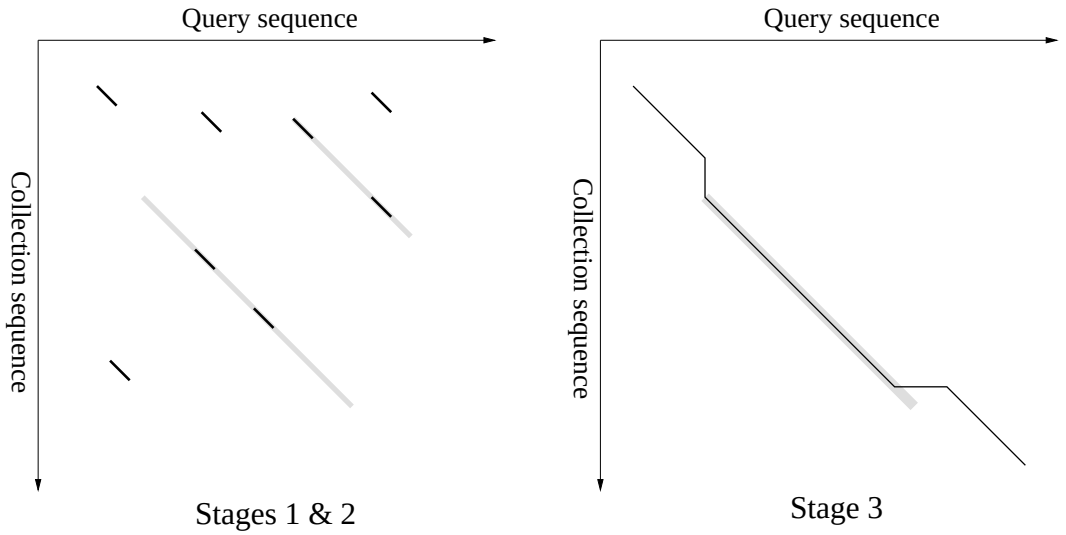

Figure 3.4: Illustration of the first three stages of the BLAST *algorithm. In stages 1 and 2, short high-scoring alignments or* hits *are identified, shown as short black lines. When two hits occur near each other and on the same diagonal an ungapped extension is performed, with the result shown as a longer grey line. In this example, the longer of the two ungapped extensions scores above* $S1$ *and is passed on to stage 3, where it is used as a starting point for constructing a higher-scoring gapped alignment.*

distance A apart. For each of these pairs, an ungapped extension is performed to determine if the hits are likely to form part of a high-scoring alignment, and the region covered by the extension is illustrated by a grey line. If an ungapped extension scores above the value of $S1$ — another constant determined by an external parameter — it is considered successful and is passed on to the third stage of the algorithm. In this example, the longer extension scores above $S1$, while the shorter does not.

In addition to the default mode where two nearby hits on the same diagonal are required to trigger an ungapped extension, BLAST can also be run in one hit mode, where a single hit alone triggers an extension. This leads to an increase in the number of ungapped extensions performed, increasing runtimes and improving search accuracy. To reduce the number of hits, a larger value of the neighbour threshold T is typically used when BLAST is run in one hit mode. The original BLAST algorithm [Altschul et al., 1990] used the one hit mode of operation, and the approach of using two hits to trigger an ungapped extension was one of the main changes introduced in the 1997 BLAST paper [Altschul et al., 1997]. The two hit mode has been shown to provide faster search with comparable accuracy [Altschul et al.,

1997], and is the default approach currently used by BLAST.

To identify pairs of nearby hits on the same diagonal, BLAST uses a array with $|q| + |s| - 2W + 2$ entries, one for each possible diagonal where a hit can occur. Each entry is used to record the location of the most recent hit on that diagonal; $|q| + |s| - 2W + 2$ entries are required because a hit of length W can occur on any diagonal $d = j - i$ ranging from $d = -|q| + W - 1$ to $d = |s| - W + 1$. Each entry in the array records the position in the collection where the last hit occurred on that diagonal. At the start of a database search, each entry is initialised to point to the start of the collection, however the array does not need to be reinitialized between collection sequences because the collection is processed sequentially from start to end. As a result, BLAST occasionally identifies a nearby pair of hits that lie on the same diagonal but are in fact related to different collection sequences. Our analysis of BLAST revealed that in practice this is sufficiently rare and does not significantly impact the performance or accuracy of the algorithm.

Once a pair of hits has been identified, an ungapped extension is performed along the diagonal where the hits occurred. The algorithm used to perform an ungapped extension using the dropoff heuristic is shown in Figure 3.5. Starting at the location of one of the hits that triggered the extension [i_{hit}, j_{hit}], pairs of residues from the query and collection sequence are aligned by traversing along the diagonal. Extension is abandoned when the total score decreases by more than a threshold, that is when `bestscore - score > dropoff`, and the highest scoring point of the extension is recorded. The extension process is then repeated in the other direction by traversing backwards along the same diagonal.

The ungapped extension algorithm presented in Figure 3.5 does not explicitly terminate when the end of the query sequence or current collection sequence is reached. BLAST uses an unpublished method that automatically terminates the extension process through the use of special character codes called sentinel codes [Sedgewick, 1990]. The sentinel codes are placed at either end of the query sequence and each collection sequence. The alignment score of the sentinel code to any residue is $-\infty$ and the extension process will automatically terminate when the start or end of either sequence is encountered. As a result, the computation required to perform an ungapped extension is reduced by eliminating the need to continually check when the sequence boundary has been reached. Sentinel codes are not employed for nucleotide search because collection sequences are recorded in a compressed form, which we describe later in this section.

We measured the performance of the second stage for protein searches, using the GenBank NR protein database and the same 100 randomly-selected queries described previously. On

/* Input: q, s, i, j, dropoff */

UngappedExtension
score $\longleftarrow$ 0
bestscore $\longleftarrow$ 0
$i_{best} \longleftarrow 0$, $j_{best} \longleftarrow 0$

while (bestscore − score $\leq$ dropoff)
    score $\longleftarrow$ score + $s(q_i, s_j)$
    if score > bestscore **then**
        bestscore $\longleftarrow$ score
        $i_{best} \longleftarrow i$
        $j_{best} \longleftarrow j$
    increment i
    increment j

/* Output: i_{best}, j_{best}, bestscore */

Figure 3.5: Ungapped extension algorithm used by BLAST *for aligning protein sequences.*

average, 9.8 ungapped extensions are performed per collection sequence, but less than 0.01% of these produce a score above the cutoff, $S1$. The effect is that around 11% of the database sequences are passed on to the third stage. The second stage consumes on average 31% of the total search time, as shown in Table 3.2.

Stage 3: Gapped alignment

In the third stage of the BLAST algorithm, a gapped alignment is performed to determine if the high scoring ungapped region forms part of a larger, higher scoring alignment. The right-hand side of Figure 3.4 illustrates an example where this is the case: the single, high-scoring ungapped extension identified in Stage 2 is considered as the basis of a gapped alignment, and the black line shows the alignment identified through this process.

Although BLAST identifies local alignments, the gapped alignment algorithm that is used differs from Smith-Waterman approach described in Section 2.2.4. Rather than exhaustively

computing all possible paths between the sequences, the gapped scheme explores only insertions and deletions that augment the high-scoring ungapped alignment. This step begins by identifying a *seed* point that lies within a high-scoring portion of the ungapped region. The algorithm then performs a *gapped extension* beginning at the seed, and proceeding in each direction, towards both the start and end of the sequences. The end-point and optimal alignment score for an extension is recorded, and the final alignment score is the sum of the scores resulting from extension in each direction.

A variation of global alignment algorithm is used to perform a gapped extension, with two important differences to Gotoh's algorithm [1982] that was described in Section 2.2.3. First, the seed point provides an origin for the extension process, and second, the alignment may end at any point in the alignment matrix. The alignment dropoff approach described in Section 2.3.2 is also employed to limit the region of the alignment matrix that is processed; the gapped extension does not consider low-scoring regions where the score falls by more than the value of the dropoff parameter, X. This parameter controls the sensitivity and speed trade-off of the gapped extension: the higher the value of X, the greater the alignment sensitivity but the slower the search process.

The seeded gapped alignment process with dropoff is illustrated in Figure 3.6. The black line in the center of the alignment matrix represents a high-scoring ungapped alignment that has been identified in the second stage. A seed point is chosen partway along the ungapped alignment and a gapped extension is performed in either direction from the seed. The grey region represents the portion of the alignment matrix that is processed by the dropoff approach.

The dropoff approach is highly effective in this context, since the majority of gapped extensions are triggered by ungapped alignments that are not part of a longer gapped alignment. Because the dropoff approach only considers high-scoring regions in the alignment matrix, the number of cells that need to be computed is greatly reduced. In our experiment with 100 queries and the GenBank collection described previously, we found that on average less than 2% of all cells in the matrix are processed when the dropoff technique is applied using default parameters.

A single collection sequence may produce multiple high-scoring alignments. In some cases these alignments represent separate, distinct regions of similarity between the two sequences that should be displayed separately to the user. In other cases, the alignments can be joined together to produce a longer alignment that was initially overlooked because it contains a low-scoring region not considered by the dropoff approach. Figure 3.7 illustrates these two

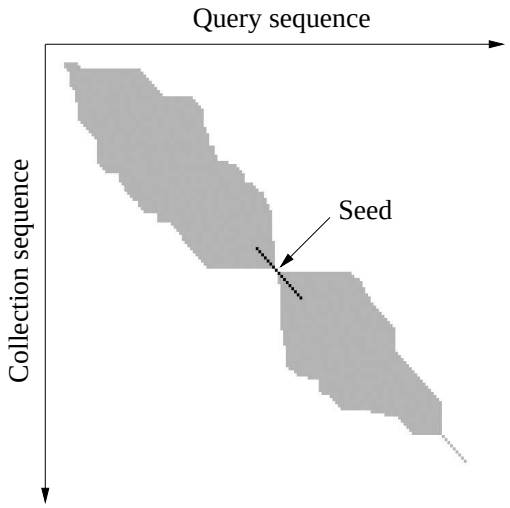

Figure 3.6: Schematic of a seeded gapped alignment with dropoff. The black line represents a high-scoring ungapped alignment between the query sequence and current collection sequence. A point on the alignment is selected as the seed and a gapped extension is performed in each direction, beginning at that point. The grey regions represent cells in the alignment matrix that exceed the best score seen so far, less a constant dropoff value.

scenarios. On the left-hand side is an example where the collection sequence produces two distinct high-scoring alignments. On the right-hand side is an example where a single high-scoring alignment has been separated into two parts, separated by a low-scoring region. Each part contains a high-scoring ungapped alignment and therefore has been identified separately. BLAST employs an unpublished strategy called *joining* that is used to link such alignments. Given a pair of alignments that are sufficiently close to each other to be joined, the collection sequence is realigned using a large dropoff parameter in an attempt to link them. Our analysis reveals that this joining approach increases the sensitivity of BLAST to alignments that contain large, low-scoring regions.

A second important, again unpublished, optimisation is employed to reduce the time required to align sequences, which we refer to as *containment*. The containment method reduces the number of gapped alignments performed for a single collection sequence when multiple high-scoring ungapped alignments are identified for that sequence. Any ungapped alignment that lies within a region of the alignment matrix already covered by a high-scoring gapped alignment is dismissed and not considered as the basis of a subsequent gapped alignment. Specifically, given a ungapped alignment starting at coordinates $[s_i, s_j]$ and ending

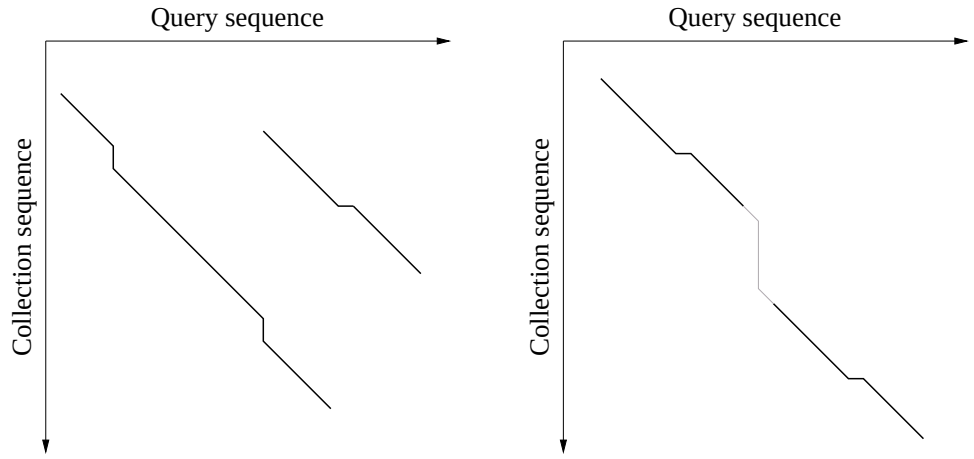

Figure 3.7: Multiple high-scoring alignments for a single collection sequence. The example on the left contains two distinct alignments that are displayed separately to the user. The example of the right contains a single alignment with an undetected low-scoring region, shown in grey, in the middle. The joining procedure links the two alignments identified by the dropoff approach that are shown as black lines to form a single alignment that is displayed to the user.

at $[e_i, e_j]$, and a gapped alignment starting at $[a_i, a_j]$ and ending at $[b_i, b_j]$, the ungapped alignment is not processed if $s_i \geq a_i$, $s_j \geq a_j$, $e_i \leq b_i$ and $e_j \leq b_j$. The containment method is illustrated in Figure 3.8. The ungapped extension labelled `A` is processed first and forms the basis of the gapped alignment that almost entirely spans both sequences. The ungapped extensions `B`, `C`, and `D` are all contained with the rectangular region bounded by the start and end of the gapped alignment, shown in grey, and are subsequently not processed. Although the ungapped alignment `D` does not form part of the gapped alignment, and may produce a different high-scoring alignment, it is dismissed by the containment approach. In practice, however, most alignments that are removed by the containment method form part of the already identified gapped alignment, such as extensions `B` and `C` in Figure 3.8, and can be removed with no effect on search accuracy. In our experiments with the GenBank NR protein database, we have found that the *containment* method reduces the average number of gapped alignments performed by approximately 9%.

The gapped alignment algorithm used to perform the third stage records only the score of the optimal alignment between the query and collection sequences. If the alignment scores at least $S2$ — the minimum score for the alignment to be deemed statistically significant — it is passed on to the fourth and final stage of BLAST where the alignment itself is recorded.

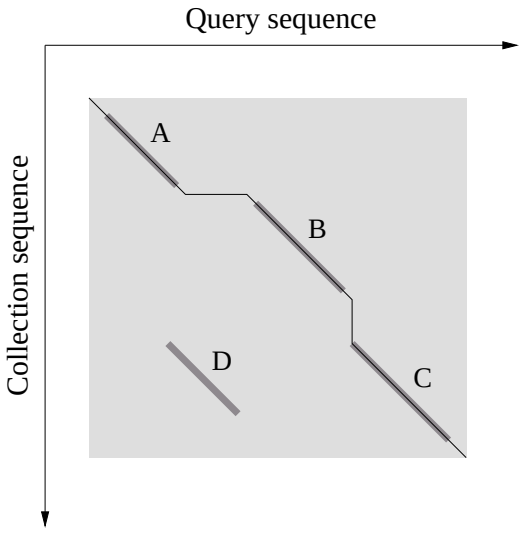

Figure 3.8: Illustration of the BLAST *containment method. Four high-scoring ungapped alignments labelled* `A`*,* `B`*,* `C` *and* `D` *are identified in the second stage. The alignment* `A` *is processed first and the gapped alignment shown as a thin black line is identified. The remaining ungapped alignments are contained with the rectangular region (shaded in grey) bounded by the start and end of the gapped alignment and are subsequently not processed.*

We discuss the methods used by BLAST to calculate an alignment E-value, which provides a measure of the statistical significance of the alignment, in Section 3.3.1. On average, in our experiment with the GenBank NR protein database described previously, we found that less than 0.01% of the gapped alignments performed during the third stage produced an E-value below the default cutoff of 10, and that this stage consumes on average 30% of the total search time, as shown in Table 3.2 on page 57.

Stage 4: Traceback and display

In the final stage of the BLAST algorithm, the alignments to be displayed to the user are rescored. During the rescoring, the alignment traceback pathway itself is recorded so that it can be displayed in the format shown in Figure 2.9; the third stage records only scores, and not the evolutionary pathway that leads to that score. We describe methods for recording the optimal alignment through traceback in Section 2.2.6. The other important difference during rescoring is that the value of the dropoff parameter, X, is increased in an attempt to find a higher scoring alignment [Altschul et al., 1997].

The number of alignments processed during this final stage of the algorithm is limited by two important factors. First, it is determined by the number of sequences found to score above the E-value cutoff in the third stage of the algorithm. Second, the value of the parameter V — which defaults to 500 — limits the total number of alignments to be displayed to the user. Around half of the 100 queries we evaluated in our experiment with the GenBank protein database described previously produced more than 500 high-scoring alignments. In this case, only the 500 highest scoring alignments from the third stage are rescored in this stage and subsequently displayed to the user. On average, the final stage consumes only 2% of the total search time.

Nucleotide search

The description of BLAST that we have presented so far pertains to BLASTP, the algorithm used to search a protein database with a protein query. The approach to searching nucleotide collections is similar, however some aspects diverge due to fundamental differences between the two types of sequence data such as alphabet size and alignment scoring scheme. We now describe the approach used to search nucleotide collections, which includes three important variations to protein search. First, the two-hit mode of operation is not employed, that is, only one hit is required to trigger an ungapped extension for nucleotide search. A longer word length is employed, by default $W = 11$, so that a single match of length eleven is considered as the basis of an alignment. Second, high-scoring matches between neighbouring words are not considered during the hit detection process; exact matches only of length W between the query and collection sequence are identified between nucleotide sequences. Third, nucleotide collection sequences are stored using a special *bytepacked* representation that reduces collection size by roughly 75% [Williams and Zobel, 1997].

The bytepacked representation is employed as follows. The `formatdb` tool converts a collection of nucleotide sequences from uncompressed FASTA format to a specially formatted file that encodes the sequences using bytepacked compression; sequences are stored using two bits per base, that is, each byte represents four nucleotide bases. This approach was shown by Williams and Zobel [1997] to yield storage space savings and, importantly, reduce retrieval times from disk when processing sequences.

We believe that bytewise compression is an important innovation in BLASTN. A diagram illustrating the approach further is shown in Figure 3.9. The figure shows three bytes that store a sequence of ten bases in length: the first byte stores four two-bit codes for `ATGA`, the

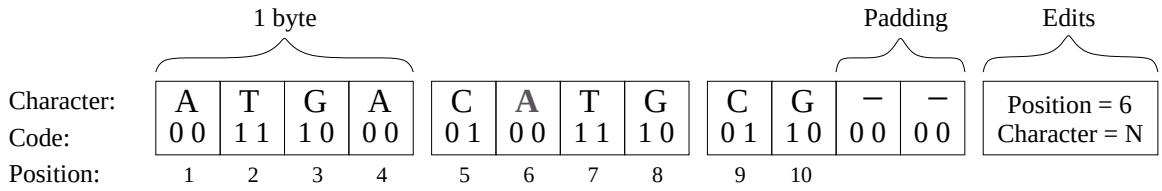

Figure 3.9: Bytepacked representation of the sequence ATGACNTGCG. *The 6th base is an unambiguous, therefore a randomly selected base (in this case A) is recored in its place, and an ambiguity code edit is stored at the end of the sequence.*

second four codes for `CATG`, and the third two codes for `CG`. The sequence length is stored separately, so decoding of the final byte is unambiguous. Note that the original sequence includes the ambiguity character `N` at position 6, which is replaced in the compressed representation by a random choice of the nucleotides represented by `N`. The additional structure shown to the right of the figure can optionally be decoded to restore the original sequence. In experiments investigating the relative sequential retrieval costs of compressed versus uncompressed representations, it has been shown that retrieval of the compressed representation is typically more than three times faster than retrieving the uncompressed representation regardless of whether ambiguity codes are decoded [Williams and Zobel, 1997].

BLAST searches a nucleotide collection by reading compressed sequences from disk and decompressing individual basepairs as required. The only exception to this is the first stage, which we discuss next, where collection sequences are processed partially in their compressed form. For the remaining three stages, each nucleotide base is extracted from the compressed sequence immediately before it is aligned. The ambiguity codes at the end of the compressed sequence are not decoded during the second and third stages of search, so that the ungapped and gapped alignments do not take the ambiguous characters into account. In practice, however, the ambiguous characters are sufficiently rare that this has little impact on accuracy [Williams and Zobel, 1997]. Only in the fourth stage, where a traceback is performed and the alignment itself is recorded, is the complete original sequence decoded and aligned to the query.

A modified approach is used to perform hit detection where sequences are compressed using the bytepacked representation. Instead of extracting each overlapping subsequence of length W from sequences in the collection, which would involve decompressing the sequences, BLAST extracts every fourth subsequence of length n where n is the largest value such that $n \equiv 0, \text{modulo}\, 4$ and $n \leq W - 3$. For the default value of $W = 11$, $n = 8$ and BLAST extracts

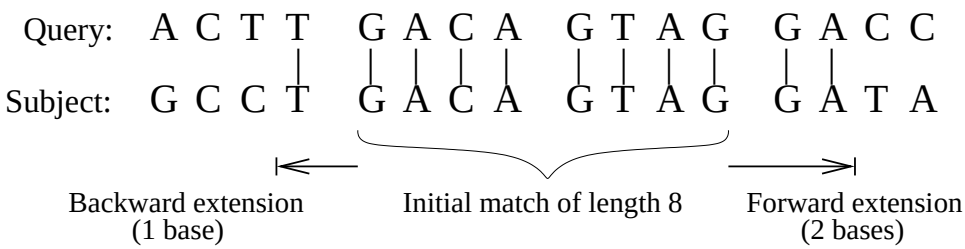

Figure 3.10: The NCBI-BLAST *approach to detecting hits of length $W = 11$. A byte aligned initial match of length $n = 8$ is identified first, then the query and collection sequences are inspected each side of the match to check if it forms part of a longer hit of length $W = 11$.*

two adjacent bytes from the collection at a time. The query lookup structure is therefore for subsequences of length $n = 8$, and not length $W = 11$. When an initial match of length $n = 8$ is found between a collection sequence and query, the original sequences are inspected $W - n$ bases in both directions surrounding the match to identify if a contiguous match of length W exists and, if this is the case, a hit is detected. This approach, which is illustrated in Figure 3.10, has two speed advantages: it allows two bytes to be compared in compressed form (that is, two collection sequence bytes do not need to be decompressed), and it permits whole bytes to be retrieved from the collection rather than fractions of bytes. In practice, $W = 7, 11, 15$ are the optimal choices to take advantage of this bytewise approach, since each must span at least 1, 2, or 3 bytes respectively.

BLAST also considers alignments between DNA sequences with differing orientations, that is, alignments between the query sequence and the reverse complement of a collection sequence. We discuss orientation and the reverse complement of sequences in Section 2.1.2 on page 17. Alignments with both orientations are considered in a single pass through the collection by searching for matches with the original query, and the reverse complement of the query simultaneously. A new query is created that is twice the length of the original, by appending the reverse complement of query sequence to the original sequence. For example, given the query sequence `GTCAAC`, the extended query `GTCAACGTTGAC` is constructed. The extended query is then used to perform all four stages of search as usual. When presenting results to the user, any alignment between a collection sequence and second half of the new query is reversed and complemented to produce an alignment between the original query and the reverse complement of the collection sequence.

To measure the percentage of time spent by BLAST performing each stage of a nucleotide search we conducted 50 searches using a test collection constructed from half of the sequences

in the GenBank non-redundant nucleotide database. Our NR/2 collection was created by randomly extracting half of the sequences from the NR collection and contains 6,862,797,036 basepairs in 1,511,546 sequences, ranging in length from 6 to 36,192,742 basepairs. A set of 50 test queries were randomly extracted from the NR database. Queries longer than 10,000 basepairs (typically entire genomes or chromosomes) were excluded from the selection process; BLAST searches with longer queries are too slow to be practical and less sensitive genome search tools such those discussed in Section 3.2.2 are better suited to such searches. The time taken to perform each stage was measured using NCBI-BLAST version 2.2.11 and the results of this experiment are shown in Table 3.2 on page 57. The best of three runs for each query was recorded.

The results highlight some interesting differences between the performance characteristics of protein and nucleotide searches. The first three stages of protein search each consume roughly one-third of the total search time, suggesting that improvements to the hit detection, ungapped alignment and gapped alignment algorithms could all reduce overall execution times and are worth investigated. In contrast, the dominant time cost for nucleotide searches is clearly the first stage, which consumes around 85% of the total time. It is therefore not suprising that new approaches for detecting hits between nucleotide sequences have received considerable attention at late, many of which are discussed in Section 3.2.

The number of hits, ungapped extensions and gapped extensions also varies considerably between searches with protein and nucleotide data. For nucleotide searches, an average of 3.5 hits are found per collection sequence; this contrasts with protein search where 229 hits on average are found per collection sequence. In the second stage, on average 3.1 ungapped extensions are performed per nucleotide collection sequence (not all hits trigger an ungapped extension because in some cases multiple hits are contained within a single ungapped alignment). This contrasts with protein search where on average 9.8 ungapped extensions are performed per collection sequence. In the third stage, gapped alignments are performed for 7% of nucleotide collection sequences, compared to 11% of collection sequences for protein searches.

BLAST variants

The BLAST algorithm forms the basis of a wide range of sequence analysis tools offered by the NCBI [McGinnis and Madden, 2004]. Several variations of tool have been developed or published, and we describe some of the more important advances here.

The BLASTX, TBLASTN and TBLASTX tools are provided by the NCBI[5] for translated searches, where a protein query is compared to a nucleotide collection or vice versa using the same fundamental algorithm as BLASTP that is described earlier in this section. The BLASTX tool compares a nucleotide query to a protein sequence database by first translating the sequence in all six reading frames as described in Section 2.2.2. The TBLASTN tool compares a protein query to a nucleotide sequence database by translating each collection sequence in all six reading frames during the search. Finally, the computationally expensive TBLASTX tool compares a nucleotide query to a nucleotide collection by translating both query and collection in all six reading frames, resulting in a total of 36 possible combinations of query and collection reading frames. In Section 3.1.3 we observe that translated searches are slower than regular BLAST searches; we speculate that this is due to the inherent ambiguity in the translation process that increases the search space and search times accordingly.

The popular MEGABLAST tool is publicly-accessible for online nucleotide searches through the NCBI website[6]. It uses a less sensitive but efficient greedy alignment algorithm [Zhang et al., 2000], a longer word length than typical tools, and compares multiple query sequences to each collection sequence in a single scan of the collection[7] (thereby reducing the costs when multiple queries are simultaneously posed by one or more users). It is not designed as a replacement for BLASTN, but is instead designed for quickly finding very similar sequences [McGinnis and Madden, 2004]. In particular, its default word length is $W = 28$, meaning that the minimum number of contiguous identical bases is 28 before any subsequent processing stage is attempted. The tool is generally faster than BLAST for searches with a large word length, long queries or multiple queries, however it is often slower than BLAST for sensitive searches with a word length of 16 or less [Gotea et al., 2003]. MEGABLAST also supports discontinuous matches using spaced seeds, which we discuss in Section 3.2.3.

The Pattern-Hit Initiated BLAST (PHI-BLAST) tool [Zhang et al., 1998b] is also available for searches through the NCBI website. The user provides a protein or DNA sequence and a pattern contained within that sequence as input. The pattern is a short sequence that may contain wildcards or gaps, and typically represents a motif such as a functional site that is conserved in the family of interest. Databases such as PROSITE [Hulo et al., 2004] provide patterns in this format for a range of families. PHI-BLAST then searches for sequences in the collection that are similar to the query and contain the specified pattern. The PHI-BLAST

[5]See: `http://www.ncbi.nlm.nih.gov/blast/blastcgihelp.shtml#translations`
[6]See: `http://www.ncbi.nml.nih.gov/BLAST/`
[7]See: `http://www.ncbi.nlm.nih.gov/blast/megablast.shtml`

algorithm begins by searching for occurrences of the pattern in the collection, and performs a gapped alignment using the dropoff approach between the query and collection sequence for each occurrence. High-scoring alignments are then displayed to the user.

Several tools have also been developed that provide post-processing and graphical visualisation of BLAST results [Plewniak et al., 2000]. For example, the AntiHunter [Lavorgna et al., 2005] tool post-processes BLAST output to aid in the identification of express sequence tag antisense transcripts. PowerBLAST adds functionality to BLAST so that it is suitable for annotating whole genomes [Zhang and Madden, 1997].

An alternative version of BLAST has been developed by Gish called Washington University BLAST (WU-BLAST)[8]. Although the author claims better sensitivity and faster search times than NCBI-BLAST, the precise details of the algorithm and source code have never been published and are not publicly available. An outdated evaluation of the tool by Rognes and Seeberg [1998] found WU-BLAST to be roughly 3 times slower than NCBI-BLAST but more accurate. A more detailed investigation that considers speed and accuracy results for varying parameter values, using a recent version of the tool on modern hardware, has not been conducted.

Usage statistics

The NCBI operate a complex network of machines that provide BLAST search services to users through the organisation's webpage. McGinnis and Madden [2004] recently reported that a farm of 200 linux machines is used to the perform searches using a distributed approach. Because BLAST performs best when the collection is small enough to reside in main memory, to avoid reading the database from disk for each search, large collections are divided amongst the available processors [Madden, 2005]. A collection such as the NR nucleotide database is divided amongst 10 to 20 machines in the farm, where each machine is responsible for searching a portion of the database and then the results from each machine are merged and displayed to the user.

We recently obtained detailed BLAST usage statistics from the NCBI for the online search service offered by the organisation [Madden, 2005]. The usage data included the query length, target database and search tool used for all BLASTN, BLASTP, BLASTX, TBLASTN and TBLASTX searches conducted between 6:00AM on 22 September and 6:00AM on 23 September, 2005 (UTC-05). A total of 142,822 searches were performed and an overview of the usage data is

[8]See: `http://blast.wustl.edu/`

	BLASTN	BLASTP	BLASTX	TBLASTN	TBLASTX	All
Searches against NR	53,747	15,326	16,190	4,129	458	89,850
Searches against other	27,038	21,304	3,252	746	632	52,972
Total searches	80,785	36,630	19,442	4,875	1,090	142,822
Average query length	1,225	288	1,164	64	1,829	1,004
Time per query (secs)	58.54	42.71	80.62	439.52	2404.55	89.29
Total time NR (days)	36.42	7.58	15.11	21.00	12.75	92.86

Table 3.3: Summary of usage statistics for BLAST *searches conducted during a 24 hour period through the NCBI web service. Estimates of the average query length, average query time and total processing time for all searches against the NR database are also reported.*

presented in Table 3.3. The data shows that BLASTN is the most frequently used member of the BLAST suite of tools, accounting for around 80,785 or 57% of the total number of searches conducted. BLASTP and the three translated search tools, which all used the same underlying protein comparison algorithm, account for a further 62,037 searches. Around 63% of queries are for searches against the protein or nucleotide non-redundant database.

We used this data to estimate the amount of CPU processing required to perform searches against the GenBank NR database each day. For each tool, we calculated the average query length from the available data and randomly selected ten queries with an average length from the relevant NR database. We used the ten queries to measure the average search time against the GenBank NR protein and nucleotide databases for each tool. The versions of GenBank used for the experiment were those described in Section 2.1.3 that are slightly older than the usage data, but should still provide a good approximation of search times. For nucleotide searches, we used the NR/2 collection described in Section 3.1.3 that contains half of the sequences from the NR database, and doubled the resulting search times as an estimate of the time required to search the complete NR database. We used the reduced NR/2 collection because it is small enough to fit into the main-memory of our test machine, which provides a better model of the total processing time in the NCBI search environment where the database is divided into sections that are small enough to be cached in main-memory. The experiment was conducted using NCBI-BLAST version 2.2.11 on a Intel Pentium 4 2.8GHz workstation with two gigabytes of main-memory — we do not have detailed information about the machines used at the NCBI so our test machine represents a typical high-end workstation in 2005.

The results of our timing experiments are shown in Table 3.3. The results show that

BLASTN searches against the NR database consume roughly 36.42 days of processing power on a single machine and that searches with the remaining four tools consume a further 56.44 days of processing. This represents a total of around 93 workstations performing searches for 24 hours to handle the load of searches against the NR database — searches against other collections represent a further 37% of queries that are not included in our runtime calculations. Further, queries are not evenly distributed through the day; for example, only 2,447 queries were posed between 8:00 PM and 9:00 PM yet 5,195 queries were posed between 4:00 PM and 5:00 PM. Our analysis also does not take into account overheads associated with distributing searches amongst the machines in the cluster and collating results for display to the user. These additional factors explain why a cluster of around 200 linux machines is required to handle the volume of searches conducted at the NCBI.

The results presented in Table 3.3 also prompt speculation about the relationship between the average search time and the volume of queries for each tool. Searches with BLASTN, BLASTP and BLASTX are considerably faster than searches with TBLASTN and TBLASTX – the latter takes around 40 minutes on average to search the non-redundant nucleotide database. Far fewer searches were conducted with these slower tools, with an apparent inverse relationship between the number of queries posed and average search time across the different versions of BLAST. This suggests that users will conduct fewer searches using tools that are slower. Therefore, improvements to the speed of search algorithms not only increase user satisfaction, but also lead to an increase in the number of searches conducted, potentially resulting in new insights and discoveries.

Weaknesses of the BLAST approach

Despite the success of BLAST as a general purpose, accurate and fast homology search tool, there are several weaknesses with the approach. First, BLAST searches are becoming slower each year with increases in collection sizes outpacing improvements to modern hardware [Chen, 2004; Attwood and Higgs, 2004]. In a study of the performance of BLAST, Chen [2004] reports that a query on the entire 2003 GenBank nucleotide database using a 2003 Intel-based server takes an average of around 260 seconds. In 2001, the same task took only 83 seconds on a 2001 GenBank collection and 2001 hardware, and in 1999 only 36 seconds. Indeed, the trend is that BLAST is becoming around 64% slower each year, and scales poorly when compared to index-based approaches such as those described in Section 3.2.1. This poor scalability is partly because BLAST is an exhaustive approach that requires enough

resources to store the entire collection in main-memory for reasonable search performance. As GenBank outgrows the available main-memory on most modern workstations, search times increase drastically, and perhaps because of this searches against the entire GenBank nucleotide database are no longer supported through the NCBI website[9]. Instead, a pruned version of the collection that is roughly one quarter the size and does not include "high-throughput, patent, genomic or sequence tagged cite (STS) sequences", which we refer to as the GenBank NR nucleotide database, is available for search [McGinnis and Madden, 2004].

The BLAST approach is also highly obscure. The algorithm relies on multiple heuristic methods to efficiently filter collection sequences while maintaining a high degree of sensitivity to distant homologies. Each heuristic adds another layer of complexity to the overall system. Further, an abundance of parameters accompany the heuristic approach, such as the word length and neighbourhood threshold used for hit detection, the score threshold used to trigger gapped alignments, and the dropoff parameter used to align sequences — each of these parameters provides a tradeoff between search speed and search accuracy. To further add to the complexity, some parameter settings are input as normalised scores, which are comparable across varying scoring schemes, while others are input as nominal scores, which are not (we discuss nominal and normalised scores in more detail in Section 3.3.1). Indeed, the NCBI-BLAST tool supports a total of 19 different parameters that control search behaviour and the alignment scoring scheme[10]. Each parameter must be carefully tuned to balance accuracy and search runtimes.

3.1.4 Summary

Exhaustive search methods such as Smith-Waterman search, FASTA and BLAST remain the most popular means of searching genomic databases. In this section, we described these three approaches. The SSEARCH algorithm compares the query to each sequence in the collection using the rigorous Smith-Waterman dynamic programming algorithm and is guaranteed to identify optimal alignments for a given scoring scheme. However, the highly sensitive approach is impractical for any collection of significant size — a search of the GenBank non-redundant nucleotide database takes 4.5 days.

Heuristic approaches such as FASTA and BLAST are considerably faster than SSEARCH, but are less sensitive to distant homologies where the percentage sequence identity is in the "twilight zone" of 25% to 40% [Chen, 2003]. The FASTA algorithm uses an heuristic approach

[9] See: `http://www.ncbi.nml.nih.gov/BLAST/`

[10] NCBI-BLAST version 2.2.10

to achieve significantly faster search with only a small loss in accuracy, however still requires around 45 minutes for the same search. The BLAST algorithm uses a four stage approach to search, where each stage filters out collection sequences that are unlikely to produce a high-scoring alignment. The four stages are hit detection, ungapped alignment, gapped alignment, and traceback and display. These stages combine to provide significantly faster search times than SSEARCH and FASTA with a small loss in accuracy. We have described each stage in detail, the different variations of the algorithm, and the underlying approaches to nucleotide and protein search. We have also presented the results of our runtime analysis of the algorithm, and a summary of usage statistics for the BLAST web service offered by the NCBI.

In the next section, we discuss other approaches to search, such as index-based approaches, algorithms for whole-genome alignment, methods that use discontinuous seeds, parallel and distributed search methods, and iterative search algorithms such as PSI-BLAST [Altschul et al., 1997].

3.2 Alternative search methods

In the previous section, we described the popular SSEARCH, FASTA and BLAST algorithms that perform an exhaustive search where the query is compared to each sequence in a collection. In this section, we describe several alternative approaches to search that have also proven successful. We begin with a description of index-based approaches that employ an on-disk inverted index to quickly identify collection sequence with a broad similarity to the query. We then describe algorithms for comparing whole genomes in Section 3.2.2, that typically employ an index structure that resides in main-memory to search smaller collections. Spaced seeds have been shown to provide better sensitivity and runtimes when searching entire genomes [Li et al., 2004] and we describe their application to search in Section 3.2.3. We describe methods for distributing the search task across multiple processors in Section 3.2.4. In Section 3.2.5 we describe iterative search algorithms, where the results from each iteration are used to update a profile describing the query for the next iteration. Finally, we describe some genomic search algorithms of interest that were not covered in the previous sections in Section 3.2.6 .

3.2.1 Index-based approaches

Several researchers have proposed genomic search schemes that use inverted indexing techniques commonly employed in text retrieval [Witten et al., 1999], which are most well-known now for their use in web search engines such as Google[11]. These schemes avoid exhaustive search through the use of a disk-based inverted index to support fast identification of sequences with broad similarity to a query. Unlike the exhaustive schemes, index-based approaches do not rely on the entire collection fitting into main-memory for reasonable search performance [Williams, 2003]. Early approaches to indexed-base search included SCAN [Orcutt and Barker, 1984], FLASH [Califano and Rigoutsos, 1993], RAMDB [Fondrat and Dessen, 1995], RAPID [Miller et al., 1999] and the work by Myers [1994], however the most successful approach is the CAFE indexed-based homology search tool [Williams and Zobel, 1996; 2002].

The significant difference between CAFE and algorithms such as FASTA and BLAST is in the first stage: CAFE uses a large, disk-based structure to identify hits (matching substrings of length W) between the query and collection sequences without exhaustively processing each sequence in response to the query. The approach used by CAFE to identify hits is conceptually the opposite of the approach used by FASTA and BLAST, which construct a lookup table from the query sequence, read overlapping words of length W from the collection, and then use the table to identify the offset of each occurrence of that word in the query. The CAFE approach constructs a large index from the entire collection, reads overlapping words of length W from the query, and then uses the index to identify the sequence and offset of each occurrence of that word in the collection.

An index is constructed once for each collection and includes a *postings lists* for each word of length W that occurs in the collection. The postings list contains the ordinal number of each sequence in the collection containing that word. In addition, the number of occurrences of the word, and the offset in the collection sequence of each occurrence is also recorded. This structure of postings lists is commonly used in information retrieval [Witten et al., 1999]. For example, consider the following postings list associated with the word `ACTT` of length four:

```
ACTT (5,2,[76,206]),(19,4,[35,79,184,203])
```

where the 5th and 19th sequences contain the word `ACTT`. The word occurs twice in the 5th

[11] See: `http://www.google.com/`

sequence, at offsets 76 and 206, and four times in the 19th sequence, at offsets 35, 79, 184, and 203.

To search the collection for high-scoring alignments to a query sequence, overlapping words of length W are extracted from the query. For each word, a postings list is retrieved from disk and the location of each hit is recorded using a set of *accumulators*. An accumulator is a data structure that stores the number and location of hits for a single collection sequence. Collection sequences are then ranked using the FRAMES metric, which is the keystone of the CAFE system. The FRAMES score for a collection sequence is based on the number and location of hits associated with the collection sequence and provides a coarse measure of similarity to the query. Collection sequences that score highly are subsequently retrieved from disk and aligned to the query, and the resulting alignments are presented to the user.

Results showed that CAFE was around eight times faster than BLASTN and over eighty times faster than FASTA [Williams and Zobel, 2002]. However, despite its promise, CAFE has not found widespread acceptance due to several drawbacks with the CAFE approach and index-based schemes in general [Williams, 2003]:

- The index structure consumes a large amount of disk space with Williams and Zobel [2002] reporting an index size for nucleotide data that is around 2.2 times larger than the original, uncompressed collection. This equates to almost 9 times larger than the size of the collection after it has been compressed using the bytepacked format employed by BLAST.

- The set of accumulators used by CAFE to perform ranking with the FRAMES metric is large and must reside in main-memory [Williams, 2003]. The data structure increases in size with longer queries and larger collections. Williams and Zobel [2002] report runtimes for queries with an average length of less than 500 characters that is not representative of typical queries today. Further, the current GenBank collection is considerably larger than the 1998 version tested in the paper.

- The CAFE system is a research prototype that does not support much of a functionality offered by tools such as BLAST. In particular, no method is currently provided for efficiently updating the index [Williams, 2003], which is an important aspect given that genomic databases are updated daily.

- Chen [2004] reports experimental results that show CAFE to be less accurate than popular exhaustive methods such as BLAST and FASTA.

Despite these shortcomings of the CAFE system, the idea of using an inverted index for faster search is a promising one. Unlike the exhaustive methods, CAFE scales well with the continued exponential growth of genomic data [Williams and Zobel, 2002], and index-based approaches are likely to be the only practical method for searching genomic collections in the future. Exhaustive methods such as BLAST rely on collections fitting into main-memory for reasonable search times [Williams, 2003] which is becoming increasingly infeasible. We expect that many of the drawbacks with index-based approaches will be addressed by future research, including proposed methods for ranking sequences that require less main-memory, support for index updates, and efficient alignment using schemes such as the dropoff method employed by BLAST.

3.2.2 Whole-genome alignment

The recent sequencing of entire genomes has created new, interesting problems and applications in genomic search and alignment. With the sequencing of highly-similar complete genomes such as human and mouse the comparison of two complete genomes is especially insightful [Couronne et al., 2003], and the alignment of entire genomes is a valuable aid to finding protein coding regions and assembling genomes from the output of shotgun sequencing methods [Miller, 2001].

Algorithms for comparing genomes must be able to deal with complications such as sequence rearrangements, motif duplications, incomplete drafts of genomes and provide appropriate visualisation for the mass of information generated by the comparison [Bray et al., 2003]. They must also be capable of aligning very long sequences with a large effective search space. As a result, popular search algorithms such as BLAST and FASTA that are designed for searching a large collection with a short query [Couronne et al., 2003] are not suitable for aligning large sequences such as entire genomes [Delcher et al., 1999; Miller, 2001]. Further, genomes are a fraction of the size of large data banks such as GenBank, which permits the use of large in-memory indexing structures.

Early approaches for aligning very long sequences such as entire chromosomes include the SIM algorithm [Huang et al., 1990; Chao et al., 1995], which uses a fast, space-efficient dynamic programming algorithm to align long sequences and is also capable of detecting high-scoring, suboptimal alignments that commonly result from large sequence comparisons. More recently, several tools that use suffix structures to compare and search whole genomes have been proposed. The popular MUMMER tool [Delcher et al., 1999; Kurtz et al., 2004] constructs

a suffix tree [Gusfield, 1997] from genomic data and uses the tree to identify matching regions. The more recent AVID tool [Bray et al., 2003] also employs suffix trees for whole-genome alignment. Suffix trees are capable of efficiently comparing millions of nucleotides but are space-inefficient main-memory structures that are not practical for searching large collections, as we discuss further in Section 3.3.4.

Index-based approaches similar to those described in Section 3.2.1 have also been applied to the alignment of very long genomic DNA sequences. The BLAT [Kent, 2002], PATTERNHUNTER [Ma et al., 2002; Li et al., 2004] and SSAHA [Ning et al., 2001] algorithms construct a main-memory index, which is only practical for searching smaller collections that contain at most a few hundred megabytes of data. Main-memory index-based approaches are particularly fast for less sensitive searches where the number of matching regions identified by the index is small because search time is dependent on the number of matches, rather than the size of the collection and number of matches when an exhaustive search is performed [Ning et al., 2001].

BLAT and SSAHA are staggeringly fast for the task of finding near-identical matches between longer queries and whole genomes [Ning et al., 2001; Kent, 2002]. Both tools create an index of non-overlapping words from the collection, which leads to significantly faster search but poor sensitivity. BLAT, for example, records the location of non-overlapping words in the collection with a default word length of $W = 12$. Overlapping words of length W from the query are then extracted and used to identify hits. This approach produces an index that is significantly smaller — roughly $\frac{1}{W}$ the size of an index that records every overlapping word — however a matching region of $2W - 1$ bases between two sequences is required before BLAT is guaranteed to detect it. Further, while the BLAT scheme has been tested on whole genomes — which are typically one to two orders of magnitude smaller than GenBank — it could not be applied on current desktop hardware to larger scale search problems because it relies on a main-memory index of the collection [Kent, 2002].

The PATTERNHUNTER approach also relies on a main-memory index to identify matching regions between the query and collection sequence efficiently [Ma et al., 2002; Li et al., 2004], however uses a novel approach to this first stage. Rather than searching for exact, contiguous matches between sequences, the PATTERNHUNTER algorithm searches for matching bases that follow a template or *seed*, which we discuss in more detail in Section 3.2.3. Using this approach, Li et al. [2004] report that PATTERNHUNTER is 5 to 100 times faster than BLAST and more accurate. However, similarly to BLAT, PATTERNHUNTER relies on a main-memory index of the collection and is impractical for very large scale database search; the reported

results in Ma et al. [2002] and Li et al. [2004] are for searches of collections in the order of a few tens of megabytes in size, using a query of similar size. The PATTERNHUNTER approach is patented and no openly-configurable versions are available.

Schwartz et al. [2002] have recently developed the BLASTZ tool which employs a variation of the BLAST algorithm for aligning entire genomes. This tool also employs spaced seeds and less sensitive alignment methods than the original BLAST algorithm to permit efficient comparison of very long sequences.

3.2.3 Discontinuous seeds

Ma et al. [2002] introduced the concept of *discontinuous* or *spaced seeds* with their PATTERNHUNTER approach. Spaced seeds provide a novel approach to matching in the first stage. Rather than requiring exact, contiguous matches of length n, matches between n bases within a window of length m are detected, where $m > n$. In addition, the n matching bases must follow a template or seed, that is, the offset of each of the n matches in the window of size m is important. The seed can be represented as a binary mask string, such as `111010010100110111`, where matches in all of the `1` positions are required for a hit and `0` denotes positions where either a match or a mismatch is allowed. For example, given the short seed `10101`, the short sequences `AAAAA` and `AGAGA` would produce a hit because the first, third and fifth bases match and the seed specifies that the second and fourth positions do not need to match. The sequences `AAAAA` and `AAAAC` would not produce a hit because the fifth base differs.

Search algorithms that use discontinuous seeds have received considerable attention lately; see Brown et al. [2004] for a recent survey covering the area. Indeed, the concept of allowing mismatches at fixed positions has some appealing properties, especially for DNA sequence alignment where changes to some bases do not affect the encoded protein sequence. Spaced seeds have been shown to provide better sensitivity because they do not rely on a matching region of W consecutive bases for an alignment to be considered [Brown et al., 2004]. Further, spaced seeds better accommodate for nucleotide substitutions that do not affect the encoded protein sequence. Indeed, Brown et al. [2004] report that spaced seeds produce fewer chance matches between unrelated sequences than continuous seeds (and as a result, faster search times) at the same level of sensitivity.

Spaced seeds have been employed in several search tools, and variations of the basic approach have been proposed. The most popular tools to employ spaced seeds are PATTERN-

HUNTER and MEGABLAST [Zhang et al., 2000]. Buhler [2001] investigate the use of multiple spaced seeds to improve the performance of their LSH-ALL-PAIRS tool for multimegabase genomic DNA sequence alignment. Their approach uses a set of randomly generated discontinuous seeds, where a match with any of the seeds is required to trigger a hit. Brejova et al. [2005] describe another extension of the spaced seed model called *vector seeds*, where positions in the seed are weighted differently and mismatches are permitted at variable positions. Methods for selecting seeds that maximise sensitivity while minimising search times have also received considerable attention of late [Gotea et al., 2003; Buhler et al., 2003; Sun and Buhler, 2004; Choi et al., 2004; Brejova et al., 2004]. Spaced seeds have been more commonly applied to nucleotide alignments, although Brown [2004; 2005] presents a framework for using spaced seeds in protein search and shows that they can achieve comparable search accuracy to the continuous seed used in BLAST and produce roughly 25% as many hits.

However, despite their attractive properties, spaced seeds appear to have limited applications. Sun and Buhler [2004] observe that discontinuous seeds are most effective for homology search when words are extracted from the collection only once to construct an index, using the schemes described in Sections 3.2.1 and 3.2.2. They appear to be less practical for exhaustive search algorithms such as FASTA and BLAST, perhaps because these tools extract words from the collection each time it is searched, and discontinuous words are more costly to extract than continuous words. The most successful applications of spaced seeds have been algorithms that record an index of the entire collection in main-memory such as PATTERNHUNTER and LSH-ALL-PAIRS. Main-memory indices have limited applications, as discussed in Section 3.2.2, and it is unclear how much of the performance gain reported for tools such as PATTERNHUNTER is due to spaced seeds as opposed to the use of an main-memory index. The only popular exhaustive search tool to employ discontinuous seeds is a variation of the MEGABLAST search tool, however it is unclear if MEGABLAST is faster than regular BLAST, and it is considerably less sensitive [Gotea et al., 2003]. One interesting area of research that has not been considered, however, is the application of spaced seeds to disk-based indexing schemes such as CAFE [Williams and Zobel, 2002].

3.2.4 Parallel and distributed search

The task of searching a genomic collection can be parallelised and distributed across a collection of processors. Hughey [1996] provides an overview of different approaches to parallelising homology search, and defines two alternative methods. The first method is the coarse-grain

approach where collection sequences are divided amongst the available processors, and each processor is responsible for searching its allocated portion of the collection. High-scoring alignments are then collated and displayed to the user. This approach is attractive because the processors can be loosely coupled, for example, a cluster of linux workstations connected via Ethernet such as the arrangement described in Section 3.1.3 can be used. However, the coarse-grain approach is not suitable for aligning a smaller number of very long sequences [Hughey, 1996]. The second method is the fine-grain approach, where the alignment of a single collection sequence is divided amongst the processors. Each processor is responsible for a portion of the alignment matrix, such as cells on a specific diagonal, and calculates values for those cells only. The fine-grain approach requires a tightly-coupled system due to a high inter-dependence between cells in the dynamic programming matrix [Hughey, 1996]. We describe some applications of the coarse-grain and fine-grain approaches in this section.

Yap et al. [1998] describe a system that divides collection sequences amongst processors in a parallel computer and reports a reduction in search times by a factor of almost N, where N is the number of available processors. Lin et al. [2005] describe a similar system for distributing BLAST searches across a cluster of loosely-coupled workstations with analogous results. Similarly, Deshpande et al. [1991] describe a distributed implementation of the FASTA algorithm. Several researches have also described parallel homology search tools for specific hardware. Brutlag et al. [1993] describe the BLAZE system that performs a Smith-Waterman search using the fine-grain distributed approach on the massively parallel MasPar computer with 4096 processors and report a substantial speed gain. Alpern et al. [1995] use microparallelism by conceptually dividing the 64-bit registers on the Intel Paragon i680 into four 16-bit parts that permits four cells in the alignment matrix to be processed at a time. The authors report a five-fold speed increase when aligning protein sequences.

Several commercial, special-purpose hardware solutions have also been developed including Paracel's GeneMatcher, Compugen's Biocellerator and TimeLogic's DeCypher [Rognes and Seeberg, 2000]. White et al. [1991] describe another special-purpose hardware solution that provides a 1000-fold speed-up over regular dynamic programming software solutions. Unfortunately, these approaches rely on expensive purpose built hardware and as a result they have not been widely adopted.

Rognes and Seeberg [2000] investigated the use of the MultiMedia eXtensions (MMX) and Streaming SIMD Extensions (SSE) instructions available on modern Intel processors and managed to achieve a six-fold speed increase when performing Smith-Waterman alignments. However, their approach has two main limitations; not all processors support the special

instructions and alignment scores greater than 255 are not considered. Further, the authors concede that their approach would be less effective if used in conjunction with the BLAST drop-off heuristic that is described in Section 2.3.2.

3.2.5 Profiles and iterative search

The homology search algorithms described so far in this chapter identify high-scoring pairwise alignments between a query sequence and sequences in a collection. Unfortunately, distantly related sequences often do not generate statistically significant pairwise alignments that can be distinguished from chance similarities [Pearson, 1996]. In this section we describe sequence profiles, that can be used to represent a family of related sequences. Sequence profiles provide greater sensitivity to evolutionary relationships because they encapsulate information about a family of sequences rather than a single member of that family alone [Aravind and Koonin, 1999]. For example, a profile can specify which regions of a sequence are more conserved than others, based on the degree of conservation between members of the same family. The two most commonly employed type of profiles are Position Specific Scoring Matrices (PSSMs) [Altschul et al., 1997; Schaffer et al., 1999] and profile Hidden Markov Models (HMMs) [Karplus et al., 1997; 1998; 2003]. We describe the SAM [Karplus et al., 1998] and PSI-BLAST [Altschul et al., 1997] iterative search methods that use profiles and are highly sensitive to subtle protein relationships. Profiles are commonly employed for protein sequence analysis and are less frequently used for the analysis of nucleotide data.

Sequence profiles

A sequence profile is constructed from a multiple alignment of related sequences. A profile encapsulates information about the group of related sequences, and can be used to efficiently search large collections [Altschul et al., 1997; Karplus et al., 1997]. The two most successfully employed types of profiles are PSSMs and HMMs. A PSSM is an array of numeric values with a rows and $|q|$ columns, where a is the alphabet size and $|q|$ is the length of the profile. Each column K_i of the PSSM represents a single residue position, and each value in the matrix K_{ij} represents the score resulting from the alignment of residue j from a collection sequence to residue position i in the profile. The alignment scores for each column K_i of the PSSM are derived from the residue frequencies at position i in the multiple alignment that was used to construct the profile, based on a similar approach to the construction of scoring matrices that is described in Section 2.4.

	K	V	F	E	R	C	E	L	A	R	T	L	K	R	L	G	M	D	G	Y	R	G	I	S	L	A	N	W	M
	K	I	F	R	K	C	E	F	A	E	L	L	E	R	R	Y	R	L	S	R	E	D	I	K	-	-	N	W	V
	K	V	F	R	K	C	E	F	A	Q	L	L	E	T	K	Y	Y	L	S	R	N	D	I	K	-	-	N	W	V
	K	V	F	R	E	C	E	F	A	E	L	L	E	T	R	Y	C	L	S	R	N	D	I	K	-	-	N	W	V
	K	H	F	S	R	C	E	L	V	H	E	L	R	R	-	-	-	Q	G	F	P	E	N	L	M	R	D	W	V
	K	K	F	D	K	C	S	L	A	K	A	L	L	-	-	-	A	Q	G	F	S	K	A	D	L	R	N	W	V
A	-1	-1	-3	-1	-2	-1	-1	-2	0	-2	0	-2	-2	-1	-1	-1	0	-2	0	-2	-1	-1	0	-1	-1	0	-2	-3	-1
R	2	-1	-3	4	5	-4	-1	-3	5	3	1	-3	3	5	3	-1	1	-1	0	4	2	-1	1	0	-1	2	-1	-3	-3
N	-1	-2	-4	0	-1	-3	-1	-4	-2	-1	-2	-4	-1	-1	-1	-1	2	-1	-2	-2	3	0	-3	-1	-2	4	6	-4	-3
D	-1	-3	-4	3	-1	-4	1	-4	-2	-1	-2	-4	0	-1	-1	-1	-2	2	0	-3	0	4	2	3	-2	-1	3	-5	-4
C	-4	-2	-3	-3	-4	10	-4	-2	-3	-4	-2	-2	-4	-2	-2	-2	-2	-3	-1	-3	-3	-4	-2	-3	-1	-1	-4	-3	-1
Q	1	-1	-4	0	1	-4	1	-3	-1	3	-2	-3	1	0	0	-1	4	4	-2	-1	0	0	-2	0	-1	-2	-1	-3	-2
E	0	-2	-4	2	2	-4	5	-4	-2	3	-1	-4	4	-1	0	-1	-1	0	-2	-2	2	3	-2	0	-1	0	0	-4	-3
G	-2	-3	-4	-2	-3	-3	-2	-4	-2	-3	2	-4	-3	-2	-1	2	-1	-3	-2	-3	-2	2	-2	-2	-2	0	-1	-3	-4
H	-1	4	-2	-1	-1	-4	-1	-3	-1	5	-2	-3	-1	-1	-1	0	-2	-1	5	-1	-1	-2	-2	-2	-2	-1	0	-3	-3
I	-3	2	-1	-4	-4	-2	-4	1	-3	-4	-2	1	-2	-2	-1	-1	-1	-1	-2	-2	-4	-4	-2	-2	1	-1	-4	-3	2
L	-3	0	0	-3	-3	-2	-3	4	-1	-3	0	5	1	-2	1	-1	-1	2	-4	-1	-3	-4	4	1	3	-2	-4	-2	1
K	6	2	-4	1	4	-4	0	-3	-1	3	2	-3	2	1	2	-1	-1	-1	-4	0	0	2	0	3	-1	-1	-1	-4	-3
M	-2	0	0	-2	-2	-2	-3	1	-2	-2	-1	2	-1	-1	0	-1	-1	0	-1	-1	-3	-3	-2	-1	4	1	-3	-2	4
F	-4	-2	7	-4	-4	-3	-4	5	-1	-3	0	0	-3	-2	-1	1	3	-2	-3	5	-4	-4	0	-3	0	-1	-4	1	-1
P	-2	-3	-4	-2	-2	-4	-2	-4	-3	-2	-2	-4	-2	-2	-1	-1	-1	-3	-4	-4	4	-2	-1	-2	-1	-2	-3	-4	-3
S	-1	-2	-3	2	-1	-1	2	-3	-2	-1	-2	-3	-1	0	-1	-1	-2	-1	-2	-2	2	-1	-3	1	-1	-1	0	-3	-2
T	-1	-1	-3	-1	-1	-1	-1	-2	0	-2	-1	-2	-2	2	-1	-1	-1	-2	3	-2	-1	-2	-1	-1	-1	0	-1	-3	-1
W	-4	-3	1	-4	-4	-3	-4	-1	-1	-3	2	-2	-3	-2	-1	1	-1	-3	-1	0	-4	-4	-1	-3	-1	-1	-5	12	-3
Y	-2	-1	3	-3	-2	-3	-3	1	-3	-1	-3	-2	-2	-2	-1	5	-1	-2	-3	4	-3	-3	-3	-2	-1	-2	-3	2	-2
V	-3	3	-1	-3	-3	-1	-3	0	-2	-3	-2	0	-2	-2	-1	-1	2	-1	-3	-2	-3	-4	-2	-2	0	-1	-4	-4	5

Figure 3.11: A Position Specific Scoring Matrix constructed from a multiple alignment of related lysozyme protein sequences. The multiple alignment at the top contains subsequences from lysozyme proteins with GI accessions 27261765, 20159661, 7327646, 30088921, 28465341, and 4557894 (listed from top to bottom). The PSSM at the bottom was constructed from the multiple alignment by PSI-BLAST

Figure 3.11 shows a multiple alignment of related lysozyme sequences and a PSSM that was constructed by PSI-BLAST [Altschul et al., 1997] from that multiple alignment. Each column in the PSSM corresponds to a residue position in the multiple alignment. Generally, residues that occur frequently at each position in the multiple alignment also score highly at that position in the PSSM, while residues that do not appear at a position score poorly in the corresponding column in the PSSM. For example, the far right column of the multiple alignment contains the residues `M` and `V`. Accordingly, residues `M` and `V` both score highly in the last column of the PSSM with alignment scores of 4 and 5 respectively.

A PSSM records information about the characteristics of a protein family that cannot

be represented by a single member of the family on its own. As a result, PSSMs can be used to dramatically increase the sensitivity of sequence alignment methods to homologous relationships when compared to basic pairwise methods [Park et al., 1998; Chen, 2003]. Further, dynamic programming algorithms that compute the optimal alignment between two sequences such as those described in Section 2.2 are easily adapted to align a PSSM to a sequence with little or no change in computational cost, and the BLAST algorithm is easily adapted to search a database of collection sequences with a PSSM instead of a query sequence and scoring matrix [Altschul et al., 1997].

Profile-based search methods

One important application of profiles is iterative search, where several passes through the collection are made, and each pass identifies high-scoring alignments between the profile and the collection that are used to update the profile for the next iteration [Altschul et al., 1997; Karplus et al., 1998]. This iterative approach is similar to the *query expansion* or *relevance feedback* schemes that have received much attention in information retrieval [Billerbeck and Zobel, 2004]. The popular PSI-BLAST algorithm performs iterative BLASTP searches using a PSSM as follows.

1. Perform regular BLAST search with user-provided query sequence and collection.
2. Construct a Position Specific Score Matrix from high-scoring alignments with an E-value below the threshold t.
3. Use the PSSM constructed from high-scoring alignments to search the collection.
4. If the list of high-scoring alignments is unchanged between this iteration and the last, or the maximum number of iterations j has been reached, then stop, otherwise return to Step 2.

The PSSM is updated with similar (and likely related) sequences from the collection during each iteration, which increases sensitivity to more distantly related sequences. When default parameters are employed, only alignments with an E-value less than $t = 0.002$ are used to update the profile (where an E-value reflects the likelihood an alignment was due to chance, and a lower E-value indicates greater similarity as discussed in Section 3.3.1) and a maximum of $j = 20$ iterations are performed[12].

[12] NCBI-BLAST version 2.2.10

Park et al. [1998] compare the accuracy of several homlogy search tools using the SCOP database, which classifies sequences into related groups, and methods for assessing retrieval accuracy similar to those described in Section 3.3.3. They found that iterative methods such as PSI-BLAST detect twice as many homologous matches and three times as many remote homologues than standard pairwise approaches such as FASTA and BLAST. However, this increased sensitivity comes at a cost — each iteration of PSI-BLAST takes roughly as long as a single BLAST search [Altschul et al., 1997], resulting in an overall search time up to around j times longer. Further, the accuracy of iterative methods is degraded by two common problems. First, PSI-BLAST searches can go astray with false positive alignments contaminating the profile [Schaffer et al., 2001]. This problem occurs when an alignment to a unrelated collection sequence produces an E-value below j, and the sequence is used to update the PSSM for the next iteration. As a result, the new profile no longer represents only sequences related to the query and may produce high-scoring alignments to other unrelated sequences in subsequent iterations. The likelihood of contamination can be reduced by improved estimates of the statistical significance of alignments, such as those discussed in Section 3.3.1. The second problem with iterative methods is profile saturation [Li et al., 2002], where a large number of near-identical sequences dominate the profile so that it no longer represents the original query. Such saturation can be caused by the over-representation of proteins with numerous entries in the collection. Altschul et al. [1997] describe measures to counter this problem by applying varying weights to collection sequences during profile construction, however research by Li et al. [2002] suggests that redundany in collections continues to pose a problem for iterative search techniques. We discuss this issue in more detail in Section 3.3.4.

Profiles can also be used to build a database of protein families, where each entry in the database is a profile that represents a single family. For example, the BLOCKS database provides a collection of profiles that are used to represent conserved regions in protein families [Pietrokovski et al., 1996]. The IMPALA algorithm [Schaffer et al., 1999] uses a complementary approach to PSI-BLAST and compares a single query sequence to a collection of PSSMs. However, unlike PSI-BLAST which uses the heuristic BLAST approach, IMPALA compares the query to each PSSM using the rigorous Smith-Waterman algorithm. Since collections of PSSMs are typically much smaller than collections of sequences, this approach is feasible in practice [Schaffer et al., 1999]. IMPALA is roughly as sensitive as PSI-BLAST, however, it is limited to the comparison of proteins that have been annotated and grouped into related families and is not as comprehensive as regular sequence database search tools [Schaffer et al., 1999]. Recently, a new yet unpublished heuristic variation of the IMPALA approach has been

devised called RPS-BLAST that uses the BLAST algorithm instead of Smith-Waterman and is provided as a component of the CD-SEARCH tool for annotating protein domains [Marchler-Bauer and Bryant, 2004]. Another approach by Panchenko [2003] compares two profiles for even greater sensitivity to weak similarities, however the approach relies on sufficient data to construct a pair of profiles.

Hidden Markov Models

The second common profile representation is through the use of Hidden Markov Models [Karplus et al., 1997]. Profile HMMs are attractive because they are able to capture additional information about a group of related sequences that is not supported by PSSMs, such as the likelihood of insertions and deletions at different residue positions. A solid statistical theory is also available for scoring alignments with HMMs, which form the basis of the SAM sequence analysis toolkit [Karplus et al., 1997; 1998; 2003]. HMMs have been shown to provide better sensitivity to homologous relationships than PSSMs [Park et al., 1998], however are more computationally expensive to align. The SAM-T98 method [Karplus et al., 1998] is an iterative search algorithm that is similar to PSI-BLAST but uses HMMs instead of PSSMs. The SAM-T98 algorithm works as follows. First, a BLAST search with a high E-value cutoff is used to identify collection sequences with a broad similarity to the query. Next, collection sequences with a low E-value are used to construct an initial profile HMM. The HMM is then used to rescore the longer list of collection sequences with a broad similarity to the query, and the resulting high-scoring alignments are used to update the HMM. This process is repeated for another three iterations, and the results of the final iteration are displayed to the user. The SAM-T98 is slower but more sensitive than PSI-BLAST [Park et al., 1998].

3.2.6 Other search algorithms

In this section we describe three homology search algorithms that were not covered in previous sections. We describe the SALSA [Rognes and Seeberg, 1998] and SENSEI [States and Agarwal, 1996] tools that are exhaustive search methods with similarities to BLAST. We also describe the Intermediate Sequence Search approach that uses an intermediate sequence to detect homologous relationships.

Rognes and Seeberg [1998] describe the SALSA algorithm that uses a three stage approach similar to the exhaustive algorithm used by BLAST. High-scoring word matches between the query and collection are identified in the first stage, and the second stage identifies

high-scoring ungapped alignments. In the third stage, SALSA attempts to link high-scoring ungapped regions to produce a longer alignment with gaps. Rognes and Seeberg [1998] report that SALSA is almost as sensitive as a Smith-Waterman search and is faster than FASTA.

States and Agarwal [1996] proposed some novel approaches to search in their BLAST-like alignment tool SENSEI. They propose the use of filtering techniques, such as those described in Section 3.3.2 on page 95, to remove low-complexity regions from the query sequence and this approach has since been integrated into BLAST. They also describe a one-bit per base representation for nucleotide sequences that indicates if the base is a purine or a pyrimidine (thus A and G, and C and T, are indistinguishable), which enables faster search but has not been widely adopted.

Park et al. [1997] and Gerstein [1998] describe a transitive method to sequence comparison that uses an intermediate sequence to detect homology. Their approach is based on the following observation; given a high-scoring alignment between sequences A and B, and a high-scoring alignment between sequences B and C, homology between sequences A and C can be inferred. The Intermediate Sequence Search (ISS) method uses the original query sequence to search the collection with an algorithm such as FASTA or BLAST, and then performs additional searches with each collection sequence that produced a high-scoring alignment as the query. This transitive approach is more accurate than simple pairwise algorithms such as FASTA and BLAST, however not as accurate as iterative centroid-based methods such as PSI-BLAST and SAM [Park et al., 1998]. Further, the method involves several passes through the collection and is prohibitively slow for large databases.

3.2.7 Summary

This section presented alternative methods for searching genomic collections. We began with a description of index-based approaches that use an inverted index to identify collection sequences with broad similarity to the query. CAFE is the most successful such approach, which is around eight times faster than BLAST. Unfortunately, index-based approaches suffer from several drawbacks including high main-memory and disk usage, and reduced search accuracy.

In Section 3.2.2 we discussed algorithms for comparing entire genomes. In this area, several staggeringly fast approaches have been developed that rely on greatly reduced sensitivity, or an index of the entire collection residing in main-memory. As a result, many of these approaches are applicable only to limited search tasks. One significant development in

this area, however, is the use of discontinuous seeds for the first, word-match stage of search. Discontinuous seeds provide faster and more accurate search for alignment algorithms that employ a main-memory index.

We surveyed parallel and distributed approaches in Section 3.2.4. These schemes divide collection sequences, or region of the alignment matrix, amongst a cluster of processors for very fast search. However, they rely on purpose-built or expensive hardware and are not in widespread use. In Section 3.2.5 we described sequence profiles, including Position Specific Scoring Matrices and profile Hidden Markov Models, that capture the characteristics of a group of related sequences. Profiles are used in iterative search algorithms such as PSI-BLAST and the SAM-T99 method. These tools perform multiple passes of the collection and use high-scoring alignments to construct a profile, which is used as a query in subsequent iterations. Iterative search algorithms are slower but more sensitive than basic pairwise approaches such as BLAST. Finally, we discussed other important search algorithms not covered in the previous sections in Section 3.2.6.

In the next section, we discuss issues related to genomic search including measures of alignment significance, low-complexity regions in sequences, methods for assessing the accuracy of homology search tools and the management of redundancy in sequence databases.

3.3 Issues in genomic search

In this section we describe some general issues and considerations that relate to the design of a homology search tool. We begin with an overview of methods used to assess the statistical significance of alignments, and the probability that an optimal alignment score is due to a chance similarity rather than a homologous relationship. We describe the methods used by popular homology search tools such as FASTA and BLAST to convert an alignment score to a measure of statistical significance. In Section 3.3.2 we describe low-complexity regions in biological sequences and explain how these regions skew alignment scores and the assessment of statistical significance. We describe algorithms for filtering regions with a low-complexity for more robust search. We also present methods for assessing the accuracy of genomic search tools such as measures based on classifications in the SCOP database [Murzin et al., 1995] in Section 3.3.3. Finally, we explain how redundancy in genomic collections, in the form of near-duplicate entries, can affect search performance and survey techniques for managing redundancy in Section 3.3.4.

3.3.1 Statistical significance of alignments

Homology search tools such as BLAST align the query to sequences in a collection and report high-scoring alignments to the user. In this section we discuss methods for calculating the statistical significance of an alignment, which forms an important component of homology search algorithms. BLAST uses alignment E-values to distinguish alignments between homologous sequences from alignments between unrelated sequences. The alignment E-value reflects the probability that a given alignment score results from a chance similarity, and we detail the approach used by BLAST to calculate E-values.

Popular methods for assessing the significance of alignments rely upon the random independence model that considers genomic sequences to be string of letters randomly drawn from the relevant alphabet [Karlin and Altschul, 1990; Altschul and Gish, 1996]. Although each amino acid and nucleotide base has a different frequency of occurrence [Robinson and Robinson, 1991], this model assumes that there is no relationship between adjacent amino acids and that genomic sequences can be represented by a zero-order Markov model. This assumption is supported by the poor compressibility of protein and DNA sequence data [Nevill-Manning and Witten, 1999; Weiss et al., 2000; Korodi and Tabus, 2005]. Under this assumption, Karlin and Altschul [1990] show analytically that optimal alignment scores between random or unrelated sequences follow an extreme value or Gumbel distribution when gaps are not permitted. That is, the probability of a pair of unrelated sequences having an optimal alignment score S that is greater than a specific alignment score s is defined as

$$P(S > s) = 1 - exp(-Kmne^{-\lambda s})$$

where m and n are the lengths of the sequences being compared and the constants K and λ are synonymous to the *location* and *scale* of the distribution, and are dependent on the alignment scoring scheme. For ungapped alignments, the values of K and λ can be derived for any scoring scheme analytically. Although the analytical results do not extend to alignments that contains gaps, Waterman and Vingron [1994] and Collins et al. [1988] have show empirically that gapped alignments such as those generated by the Smith-Waterman algorithm also follow an extreme value distribution, assuming the gap penalty is sufficiently high. For gapped alignments, however, empirical methods must be employed to estimate the values of K and λ for each scoring schemes.

Figure 3.12 shows the alignment scores generated by FASTA [Pearson and Lipman, 1988] for a database search and illustrates that they follow an extreme-value distribution. For this

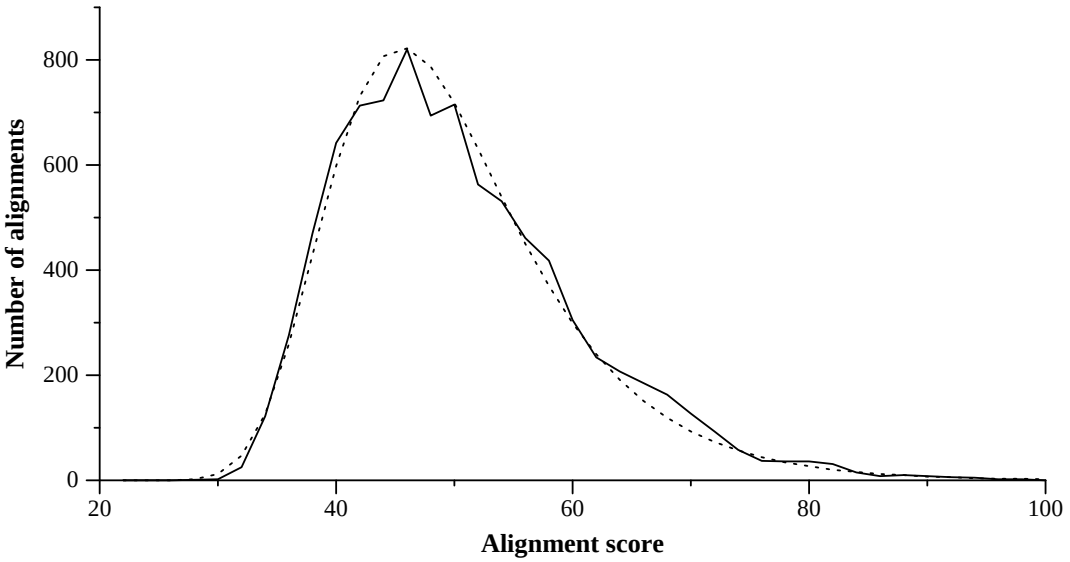

Figure 3.12: Distribution of actual and expected alignment scores for a FASTA *(version 3.4) search using a freshwater prawn lysozyme protein as the query (GI accession 30088921) to search the pruned version of the ASTRAL database described in Section 3.3.3. The actual distribution of alignment scores is represented by the solid line and the expected extreme-value distribution is shown as a dotted line.*

search, FASTA calculated values of $K = 0.024$ and $\lambda = 0.168$ and used these to estimate the distribution of alignment scores, which is shown as a dotted line. The solid line represents the observed alignment scores, which closely follows the expected distribution (the dotted line). In addition to the scores shown, 13 collection sequences produced optimal alignments with a score above 120. Such large scores have a low probability of occurence in the expected distribution of scores due to chance similarities and this suggests that these collection sequences are related to the query. Collection sequences with an optimal alignment score below 120 may also be related to the query, because not all homologous sequence pairs produce a statistically significant alignment during a database search [Pearson, 1996].

Values of K and λ that define the distribution of alignment scores can be calculated for any arbitrary scoring scheme that does not permit gaps. However, for alignments with gaps such as those generated by the Smith-Waterman algorithm and search tools such as FASTA and BLAST, values of K and λ must be calculated empirically. Earlier versions of the FASTA algorithm estimated these parameters by observing the alignment scores resulting from random permutations of the query and collection sequences [Pearson, 1996]. This

random shuffling approach was effective because it accounted for the composition of the sequences, however it was also time consuming. Later releases of FASTA use the actual alignment scores resulting from the database search to estimate the location (K) and scale (λ) of the extreme value distribution [Pearson, 1998]. This is achieved by first pruning outliers from the alignment scores, these are likely to be true homologues, and then *curve fitting* an extreme value distribution to the remaining scores. The result of this curve fitting exercise is shown in Figure 3.12.

The curve fitting approach is applicable to FASTA and SSEARCH because both algorithms generate an alignment score for every sequence in the collection (in the case of FASTA, the third stage produces an approximation of the alignment score for every sequence). However, BLAST does not generate alignment scores for every collection sequence, rendering this approach infeasible [Altschul et al., 1997]. Instead, values of K and λ are pre-computed by way of random simulation for each scoring matrix and gap penalty combination [Altschul and Gish, 1996]. The Smith-Waterman algorithm is used to align randomly generated sequences with a typical composition (that is, they are constructed by randomly selecting residues using the Robinson and Robinson background amino-acid frequencies [Robinson and Robinson, 1991]) and the resulting parameters are hard-coded into the BLAST program [Waterman and Vingron, 1994; Altschul et al., 2001]. This approach has two main drawbacks. First, it relies upon the query and collection sequence having a typical amino acid composition for an accurate assessment of significance, which is often not the case as we discuss next. Second, only a limited selection of scoring matrices and gap penalties have been simulated and are available to BLAST users [Altschul and Gish, 1996].

Two additional values are hard-coded into the BLAST program, α and β, that are calculated for every supported combination of scoring matrix and gap penalties. These parameters are used to perform *edge effect correction* when calculating the statistical significance of an alignment [Altschul and Gish, 1996; Altschul et al., 2001]. The correction takes into consideration that high-scoring alignments have a non-negligible length and generally cannot start a short distance from the end of either sequence, which affects the distribution of alignment scores between unrelated or random sequences.

During a database search, BLAST reports the statistical significance of each alignment to the user as an E-value, which represents the chance that an alignment with at least the same score would be found if a randomly constructed query with typical amino-acid composition was searched against a randomly generated database. A smaller E-value results from a higher-scoring alignment and indicates that the sequences are more likely to be related. The

length of the query sequence and the size of the collection are taken into consideration when calculating the E-value. There are three stages used by BLAST to determine the statistical significance of a alignment:

1. A alignment score of *nominal score* S is determined for each alignment using a mutation data scoring matrix and a function for penalising gaps. Nominal scores are generally written without units.

2. The nominal score is converted to a normalized score S' as:

$$S' = \frac{\lambda S - ln\ K}{ln\ 2}$$

   Scores in this normalized form are expressed in bits, and are comparable across different scoring schemes. A bit is the amount of information that is necessary to distinguish between two possibilities, such as whether or not two sequences are related. An increase in the normalized score of 1 bit indicates that an alignment is twice as likely to be statistically significant.

3. The normalized score is converted into an E-value as:

$$E = \frac{Q}{2^{S'}}$$

   where Q is the search space size, that is, $Q \approx mn$ where m and n are the total number of residues in the query and the collection respectively. BLAST uses the values of α and β to calculate the exact value of Q [Altschul and Gish, 1996; Altschul et al., 2001], taking into consideration that high-scoring alignments generally cannot start a short distance from the end of either sequence.

The resulting value of E is reported to the user as the alignment E-value. The equations can also be inverted to determine the minimum nominal score required to achieve a specific E-value. This approach is used by BLAST to determine the cutoff parameter $S2$ from the user-specified E-value. For a more detailed description of the approach used by BLAST to calculate alignment E-values see Altschul and Gish [1996].

More recently, new methods for assessing the significance of an alignment have been proposed that improve the accuracy of E-value calculations in the presence of *compositional bias*. Schaffer et al. [2001] found that query and collection sequences often contain a large

degree of compositional bias that skews the distribution of alignment scores. Since values of K and λ used by BLAST are based on simulations of randomly generated sequences with an average composition, query sequences with a non-standard composition render these parameters inaccurate. This can lead to inflated E-values, which has a particularly detrimental effect on the PSI-BLAST algorithm; the iterative approach relies on accurate assessments of significance to determine which sequences should be used to construct a profile for the next iteration of search [Schaffer et al., 2001]. Inaccurate E-values can lead to contamination of the profile, resulting in the further false positives, as discussed in Section 3.2.5.

Schaffer et al. [2001] describe an approach that rescales alignment scores to better reflect the amino acid composition of the sequences being aligned. Their approach calculates the value λ_u that is the scaling parameter for the extreme distribution of scores where gaps are not permitted. The value of λ_u can be calculated analytically and reflects the actual composition of the sequences being aligned. The scaling parameter λ'_u is then calculated for the ungapped alignment of sequences with a standard composition. Finally, the scoring matrix is rescaled so that $\lambda_u = \lambda'_u$. Assuming that the non-standard composition of the sequences has a similar magnitude effect on gapped alignment scores as ungapped alignment scores, this rescaling will improve the accuracy of the E-value measure. Indeed, Schaffer et al. [2001] find that their approach, which they refer to as *composition-based statistics*, greatly improves the accuracy of PSI-BLAST and it has been included as a default option for the tool. Yu et al. [2003] describe an approach that goes one step further by adjusting the values in the substitution matrix, rather than simply rescaling it, so that it is suitable for non-standard residue frequencies.

3.3.2 Filtering low-complexity regions

In the previous section, we described methods for measuring the significance of an alignment. These measures are based on the random independence model, where it is assumed that sequences have a standard composition and each amino acid or base in the sequence has an independent probability of occurence. Unfortunately, many sequences contain low-complexity or repeat regions, containing a bias towards certain amino acids, that strongly violate this model [Kreil and Ouzounis, 2003]. This leads to less meaningful alignments, spurious matches between unrelated sequences that share a similar compositional bias, and inflated measures of alignment significance [Altschul et al., 1994]. One solution to this problem is the use of composition-based statistics, which was described in the previous section. Another approach

MESIFHEKQEGSLCAQHCLNNLLQGEYFSPVELSSIAHQLDEEERMRMAEGGVTSEDYRTFLQQPSGNMDDSGFF
SIQVISNALKVWGLELILFNSPEYQRLRIDPINERSFICNYKEHWFTVRKLGKQWFNLNSLLTGPELISDTYLAL
FLAQLQQEGYSIFVVKGDLPDCEADQLLQMIRVQQMHRPKLIGEELAQLKEQRVHKTDLERVLEANDGSGMLDED
EEDLQRALALSRQEIDMEDEEADLRRAIQLSMQGSSRNISQDMTQTSGTNLTSEELRKRREAYFE**KQQQKQ**
QQQQQQQQQGDLSGQSSHPCERPATSSGALGSDLGDAMSEEDMLQAAVTMSLETVRNDLKTEGKK

Figure 3.13: Human ataxin-3 protein (GI accession 13518019) with a low-complexity region identified by SEG *that is shown in bold face.*

is to use a low-complexity filter, which we describe next.

Altschul et al. [1994] report that more than half of proteins contain a low-complexity region, and these regions are usually relatively short, typically ranging in length from 15 to 50 residues [Wootton and Federhen, 1993]. Common types of low complexity regions include short period repeats, aperiodic mosaics and homopolymers [Wootton and Federhen, 1993]. Many search tools process the query with a low-complexity filter before performing a database search. The filter removes these regions by replacing them with the ambiguous residue character `X` for protein, or `N` for nucleotide data. Kreil and Ouzounis [2003] provide a good overview of different methods for detecting biased regions. In this section, we describe the popular SEG [Wootton and Federhen, 1993; 1996; Altschul et al., 1994] algorithm for identifying low-complexity regions in protein sequences that is the default option for filtering BLAST protein query sequences.

Figure 3.13 shows an example of a protein sequence with a low-complexity region that was identified by SEG. The glutamine repeat in the human protein ataxin-3, which is highlighted in bold face, is associated with a neurodegenerative disease [Chow, 2004]. Glutamine repeats commonly occur in a wide range of proteins with differing evolutionary origin, structure and function. As a result, a high-scoring alignment may result from two sequences that contain a repeat region even if they are not biologically related. By removing low-complexity regions from the query sequence before a database search, such spurious matches are avoided.

The SEG algorithm identifies low-complexity regions as follows. First, a window of length L is passed across the sequence, where L typically ranges between 10 and 40 amino acids. For the region of the sequence in the window, a *complexity state vector* is constructed. The vector has a elements $\{n_1 ... n_a\}$ where a is the alphabet size, and represents the number of occurrences of each amino acid in the region. The vector is defined by

$$0 \geq n_i \geq L, \sum_{i=1}^{a} n_i = L$$

and elements in the vector are sorted in descending order. For example, given a nucleotide alphabet of size $a = 4$, the complexity vector of the sequence `AGCAAAC` is (4,2,1,0) because the most common letter, `A`, appears four times, the next most common letter, `C`, appears twice, the next most common letter, `G`, appears once and `T` does not appear at all. The sequence `AAAAAA` has a low complexity with the vector (6,0,0,0) and the sequence `AGCTAGCT` has a high complexity with the vector (2,2,2,2). Wootton and Federhen [1993] present several methods for converting the complexity state vector to a single value that represents the complexity of the subsequence. If the measure exceeds a threshold then residues in the window are masked using the `X` character.

A similar filter called DUST is used to mask repeat and low-complexity regions in DNA sequences [Hancock and Armstrong, 1994], and is the default tool for nucleotide BLAST searches. The DUST algorithm is similar to SEG, but slides a larger window with a default length of $L = 64$ across the sequence and identifies frequently occurring nucleotide triplets in the region.

3.3.3 Assessing retrieval accuracy

In this section, we discuss methods for assessing the accuracy of homology search tools. Assessments are usually based on the retrieval performance for searches of a collection of proteins that have been classified into groups of related sequences. The SCOP [Murzin et al., 1995; Andreeva et al., 2004] and PIR [Wu et al., 2003] databases both provide sequence classifications that can be used to determine whether a pair of sequences are homologous or not. Measures such as the *Receiver Operating Characteristic* (ROC) score [Gribskov and Robinson, 1996], which rewards matches between related proteins and penalises matches between unrelated proteins, can then be applied to the output of a genomic search to assess search accuracy.

The Structural Classification of Proteins, or SCOP database [Murzin et al., 1995; Andreeva et al., 2004], that was described in Section 2.1.3, has been used widely to measure the accuracy of homology search techniques [Park et al., 1997; Brenner et al., 1998; Park et al., 1998; Gerstein, 1998; Müller et al., 1999; Park et al., 2000a; Chen, 2003]. Measures based on the SCOP database are considered to be unbiased and highly rigorous. The collection contains proteins that have been hierarchically classified into four levels: class, fold, superfamily and family, based on information about the three-dimensional structure of each protein. To test the retrieval effectiveness of a search tool, sequences from the database are extracted

and used to search the entire database. The results from the search are annotated using the classifications in SCOP; matches to sequences from the same superfamily are correct and considered true-positives, while matches to sequences from a different superfamily are incorrect and considered false-positives.

The commonly-used Receiver Operating Characteristic (ROC) provides a measure of accuracy based on the ranked list of alignments returned by each search [Gribskov and Robinson, 1996]. The ROC score provides a measure between 0 and 1, where higher values reflect better sensitivity and selectivity. A list of true positives of length L is used to determine the score. For the SCOP test, this list is comprised of SCOP sequences from the same superfamily as the query. When measuring the ROC score, the list of alignments is truncated after the first n false positives, where n is typically 50 or 100. The ROC_n score is calculated as:

$$ROC_n = \frac{1}{nL} \sum_{1 \leq F \leq n} u^F$$

where F is the position of the F^{th} false positive in the list of reported alignments, and u^F is the number of true positives that ranked ahead of that false positive. For example, a search that ranks all true positives before any false positives receives a perfect ROC score of 1. A search that does not return any true positives receives a ROC score of 0. The ROC score measures both search sensitivity, that is the detection of true homologous relationships, and search selectivity, that is the ranking of homologous relationships above non-homologous relationships.

The ROC measure can be applied in one of two ways. First, a ROC score may be derived from the results for each query and then averaged across all queries to provide an overall measure of accuracy, or second, the search results for each query can be combined into a single results list sorted by E-value and the ROC measure applied [Chen, 2003; Kann et al., 2005]. We conjecture that the latter approach is better suited to assessing the accuracy of alignment E-values which should be comparable across queries. However, the former is arguably a better method for assessing the accuracy of underlying search algorithms rather than scoring schemes, because it considers each search independently, which reflects how search results are presented to the user. For our experiments throughout this thesis, the reported ROC score is determined by the former method of averaging scores across all queries where $L \geq 1$. We report ROC scores rounded to three decimal places, as is the practice used by Schaffer et al. [2001]. We contend that any effect on overall accuracy that affects ROC scores by less

than 0.001 is practically insignificant, given that variations in ROC greater than 0.01 are evident when existing search algorithms such as BLAST, FASTA and Smith-Waterman are compared [Chen, 2004] (as shown in Table 3.1 on page 51).

Generally, not all sequences in the SCOP database are used to assess retrieval effectiveness because the collection has a high degree of internal redundancy, as we show next. Because alignments between highly similar sequences are easy to detect, and homology search tools vary most significantly in their ability to detect more distant relationships, pruned versions of the SCOP database are commonly used for assessments [Park et al., 1997; 1998; Gerstein, 1998; Brenner et al., 1998; Park et al., 2000a]. The ASTRAL Compendium for Sequence and Structure Analysis [Chandonia et al., 2004] is derived from SCOP and provides subsets of the collection that have been filtered so that no two sequences share more than X% identity, where X typically ranges between 40 and 95. For our accuracy assessments, we use the ASTRAL compendium filtered at 90% identity and perform searches with every sequence in the database as the query. The most recent version 1.69 of the ASTRAL collection is 82% smaller when filtered at 90% identity[13]; this illustrates that the SCOP database has a high degree of internal redundancy and contains a large number of near-duplicate sequences.

In addition to ROC, several other measures of retrieval effectiveness have been proposed, mostly in the area of information retrieval. These include Mean Average Precision, Recall-Precision, Macro-averaged measurement schemes, Coverage versus Error, R-precision, and Mean Reciprocal Rank. Chen [2004] compares these different measures and concludes that they are all almost equally effective for assessing search accuracy.

Some researchers have used the classifications of the Protein Information Resource (PIR) instead of SCOP to assess search accuracy [Shpaer et al., 1996], however Brenner et al. [1998] and Chen [2004] believe that assessments based on PIR are untrustworthy due to three shortfalls with using the collection for this task. First, the superfamily classifications in the PIR database are based on sequence alignments, and have been defined with the aid of the same tools they are being used to evaluate, leading to a bias in favour of some tools. Second, classifications in PIR are based on sequence as well as structural similarities, which are a less reliable measure of relatedness and can lead to erroneous classifications. Third, some closely related sequences are categorised by the PIR database into different superfamilies, due to rigid classification boundaries. The PIR database defines smaller, more closely related groups of proteins and does not use a hierarchical classification system. Brenner et al. [1998]

[13]Based on pre-filtered releases of version 1.69 of ASTRAL available at `http://astral.berkeley.edu/`

found that as a result of this arrangement, each superfamily in the database is homologous to on average another 1.6 superfamilies. For these reasons, we have chosen to use the SCOP database to assess retrieval effectiveness in our work.

Although annotated protein sequence collections such as the SCOP and PIR databases provide sequence classifications that are ideal for evaluating the retrieval effectiveness of protein homology search tools, we are not aware of any similar classifications that exist for nucleotide sequence data. As a result, the assessment of nucleotide search tools has been recognised as a difficult task that has not been adequately solved [Miller, 2001]. Anderson and Brass [1998] describe a method for assessing the accuracy of DNA search tools, however their approach only tests sensitivity to alignments between protein coding regions and is based on artificial rather than real sequence data. Li et al. [2004] describe a method for assessing the accuracy of nucleotide search tools which they used to assess their PatternHunter approach. Their measure of retrieval effectiveness compares the search results for each query to the complete set of alignments identified by the exhaustive Smith-Waterman algorithm. The proportion of Smith-Waterman alignments that are identified with at least half the optimal score is recorded. This provides a measure of sensitivity but not selectivity, because it does not distinguish homologous from non-homologous relationships. Nonetheless, we believe this is the best approach currently available to assess the accuracy of nucleotide search tools.

3.3.4 Managing redundancy

Comprehensive genomic databases such as the GenBank non-redundant database contain a large amount of internal redundancy [Holm and Sander, 1998; Park et al., 2000b; Li et al., 2001b; Rapp and Wheeler, 2005]. Although exact duplicates are removed from such collections, there remains large numbers of near-identical sequences. Such near duplicate sequences can appear in genomic databases for several reasons, including the presence of:

- closely-related homologues in the database with only minor sequence variations,
- partial sequences that do not span the entire length of the protein or genome,
- sequences with expression tags and fusion proteins, where one protein has been added by biologists to the start or end of another to aid in purification or production,
- post translational modifications that cause changes in the protein sequence allowing it to attain its functional state,

```
      PQNTVFDAKRLIGRKFDEPTVQADMKHWPFKV I QAEV                 (gi 156103)
KNQVAMNPQNTVFDAKRLIGRKFDEPTVQADMKHWPFKV I QAEV                (gi 156105)
       QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV V QAEVDV L RFRSNT K ER (gi 156121)
KNQVAMNPQNTVFDAKRLIGRKFDEPTVQADMKHWPFKV V QAEVDV L RFRSNT K  (gi 552059)
KNQVAMNPQNTVFDAKRLIGRKFDEPTVQADMKHWPFKV I QAEVDV Q RFRSNT R  (gi 552055)
```

Figure 3.14: Example of highly-similar heat shock protein sequences in the GenBank NR database. Residue positions that differ between the sequences are shown in bold face, and the GI accession numbers are shown in brackets.

- and sequencing errors that are caused by experimental errors when determining a protein or DNA sequence.

These minor sequence variations lead to the over-representation of protein domains, particularly those that are under intensive research. For example, we have found that the GenBank database contains several thousand protein sequences from the human immunodeficiency virus. Figure 3.14 shows a group of highly-similar sequences found in the GenBank non-redundant protein database. The sequences vary in length and have been aligned to highlight their similarities. Residue positions where the sequences differ are shown in bold face. The five closely related sequences are from heat shock proteins in differing strains of the pinewood nematode.

Database redundancy has several pernicious effects that can "confound efforts to analyze and understand the data and to apply further research purposes" [Rapp and Wheeler, 2005]. First, a collection that contains redundancy is larger and therefore takes longer to query. Second, redundancy can lead to highly repetitive search results for any query that produces high-scoring alignments to a cluster of over-represented sequences. This results in spurious near-identical alignments that can obscure matches of interest [Altschul et al., 1994]. Third, large-scale redundancy has the effect of skewing the statistics used to determine alignment significance, by inflating the search space size that is used to calculate E-values, ultimately leading to decreased search effectiveness [Nicholas et al., 2000]. Fourth, the PSI-BLAST algorithm [Altschul et al., 1997] can be misled by redundant matches during iteration, causing it to bias the profile towards over-represented domains; this can result in a less sensitive search or even profile saturation [Park et al., 2000b;a; Li et al., 2002].

Reducing redundancy in a sequence database is essentially a two-stage process: first, redundancy within the database must be identified by grouping similar sequences into clusters;

then, the clusters must be managed in some way. In this section we describe past approaches to these two stages.

The first stage of most clustering algorithms involves identifying similar sequence pairs, that is, identify arbitrary pairs of highly-similar sequences in the database. Besides managing redundancy, the identification of similar sequence pairs is useful in many other area of bioinformations including clustering of EST (expressed sequence tag) data [Burke et al., 1999; Malde et al., 2003] and genome comparison [Kurtz et al., 2004]. An obvious approach to this is to align each sequence with every other sequence in the collection using a pairwise alignment scheme such as Smith-Waterman local alignment [Smith and Waterman, 1981]. This is the approach taken by several existing clustering algorithms, including d2_cluster [Burke et al., 1999], OWL [Bleasby and Wootton, 1990], KIND [Kallberg and Persson, 1999], and Itoh et al. [2004]. However, this approach is impractical for any collection of significant size; each pairwise comparison is computationally intensive and the number of pairs is quadratic in the number of sequences. In our own tests with BLAST, we found that an all-against-all comparison of a 100 Mb collection takes several days on a modern workstation.

Several schemes, including CLEANUP [Grillo et al., 1996], NRDB90 [Holm and Sander, 1998], RSDB [Park et al., 2000b], CD-HI [Li et al., 2001a] and CD-HIT [Li et al., 2001b], use fast clustering approaches based on greedy incremental algorithms. In general, each proceeds as follows. To begin, the collection sequences are sorted by decreasing order of length. Then, each sequence is extracted in turn and used as a query to search an initially-empty representative database for high-scoring matches. If a similar sequence is found, the query sequence is discarded; otherwise, it is added to the database as the representative of a new cluster. When the algorithm terminates, the database consists of the representative (longest) sequence of each cluster. This greedy approach reduces the number of pairwise comparisons but has three drawbacks: first, a match is only identified when one sequence is a substring of another; second, cases where the prefix of one sequence matches the suffix of another are neglected; and, third, clusters form around longer sequences instead of natural centroids, potentially leading to a suboptimal set of clusters.

Existing greedy incremental algorithms also use a range of BLAST-like heuristics to quickly identify high-scoring pairwise matches. The CLEANUP [Grillo et al., 1996] algorithm builds a rich inverted index of short substrings or *words* in the collection and uses this structure to score similarity between sequence pairs. NRDB90 [Holm and Sander, 1998] and RSDB [Park et al., 2000b] use in-memory hashtables of decapeptides and pentapeptides for fast identification of possible high-scoring sequence pairs before proceeding with an alignment. CD-HI [Li

et al., 2001a] and CD-HIT [Li et al., 2001b] use lookup arrays of very short subsequences to more efficiently identify similar sequences. However, despite each scheme having fast methods for comparing sequence pairs, the algorithms still operate on a pairwise basis and remain $O(n^2)$ in time. Indeed, we show in Section 7.1.5 that CD-HIT — the fastest of the greedy incremental algorithms mentioned and most successful existing approach — scales poorly, with superlinear complexity in the size of the collection, and requires over 9 hours to process the current GenBank non-redundant protein database.

ICASS [Parsons, 1995] and Itoh et al. [2004] reduce the number of pairwise comparisons by partitioning the collection according to phylogenetic classifications and clustering only sequences within each partition. This reduces the number of pairwise comparisons; however, the approach assumes that the database has been pre-classified and ignores possible matches between taxonomically distant species. Further, the number of phylogenetic divisions is growing at a far slower rate than database size. Therefore, a quadratic growth rate in computation time remains a limitation.

Another way to avoid an all-against-all comparison is to pre-process the collection using an index or suffix structure that can be used to efficiently identify high-scoring candidate pairs. Malde et al. [2003] and Gracy and Argos [1998] investigated the use of suffix structures such as suffix trees [Gusfield, 1997] and suffix arrays [Manber and Myers, 1993] to identify groupings of similar sequences in linear time. However, suffix structures also require large main-memories and are not suitable for processing large sequence collections such as GenBank on desktop workstations as we discuss next. Malde et al. [2003] report results for only a few thousand EST sequences. In our experiments with the freely available XSACT software, this was confirmed: the software required more than 2 Gb of main memory to process a 10 Mb collection of uncompressed nucleotide data. Similarly, the algorithm described by Gracy and Argos [1998] requires several days to process a collection with around 60,000 sequences. Alternative approaches that are suitable for processing larger collections, such as external suffix structures and compressed suffix arrays, have also been proposed [Grossi and Vitter, 2000; Mäkinen and Navarro, 2004; Tata et al., 2004; Cheung et al., 2005]. However, these data structures also appear to be unsuitable in practice because they either use a large amount of disk space, are slow for searching, or have slow construction times. Nonetheless, we conjecture that investigating data structures for identifying all pairs of similar sequences in a fixed number of passes is the correct approach.

Once a set of clusters have been identified, most existing approaches retain a single representative sequence from each cluster and delete the rest [Holm and Sander, 1998; Park

et al., 2000b; Li et al., 2001a;b]. The result is a representative database with fewer sequences and less redundancy. However, purging near-duplicate sequences can significantly reduce the quality of results returned by search tools such as BLAST. There is no guarantee that the representative sequence from a cluster is the sequence that best aligns with a given query. Therefore, some queries will fail to return matches against a cluster that contains sequences of interest, which reduces sensitivity. Further, results of a search lack authority because they do not show the best alignment from each cluster. Also, the existence of highly-similar alignments, even if strongly mutually redundant, may be of interest to a researcher.

Itoh et al. [2004] describe a system that retains all members of each cluster. This approach calculates an upper bound on the difference in score between aligning a query to any sequence in a cluster and aligning the same query to a chosen representative. During search, the query is compared to the representative and the upper bound is added to the resulting alignment score; if the increased score exceeds the scoring cutoff, all sequences in that cluster are loaded from an auxiliary database and individually aligned to the query. While this approach ensures there is no loss in sensitivity, it comes at a substantial cost: unless a high scoring cutoff is used during search—Itoh et al. use a nominal score cutoff of 150 in their experiments—there will be numerous false positives resulting in longer query evaluation times. They report experiments using Smith-Waterman [1981] alignment and it is unclear if their approach would work well if applied to an heuristic search tool such as BLAST.

3.3.5 Summary

In this section, we discussed a range of issues relating to the performance of homology search tools. First, we described methods for assessing the statistical significance of high-scoring alignments resulting from a database search. The optimal alignment score of unrelated or random sequences follows an extreme-value distribution. Search tools such as FASTA and BLAST report the likelihood that an alignment was due to chance, called the E-value, from calculations based on this distribution. Accurate calculations of the alignment E-value rely on sequences conforming to the random independence model, which is violated by sequences with a compositional bias. We described more advance measures of significance that attempt to accommodate for this bias. Another approach to correct for composition bias and low-complexity regions is query filtering. In Section 3.3.2 we described filtering algorithms such as SEG that identify regions of the query sequence with low-complexity and mask them prior to search.

In Section 3.3.3 we discussed methods for assessing the accuracy of homology search tools. The SCOP database provides classifications of protein sequences that can be used to distinguish homologous matches from non-homologous matches. We described ROC scores, which provide a measure of search accuracy at identifying related proteins. We also discussed methods for assessing nucleotide search tools and their limitations.

Finally, we discussed redundancy in genomic collections and existing approaches to managing redundancy in Section 3.3.4. Although exact duplicates are removed from collections such as GenBank, a large number of near-duplicate sequences remain. Such redundancy has several negative effects on homology search, and we discussed methods for identifying and managing redundancy. Many of these schemes rely on an all-against-all comparison which is prohibitively slow for large collections, while others rely on index and suffix structures that require too much memory. Further, most schemes reduce redundancy by creating a representative database which is not as comprehensive. We present new approaches for managing redundancy in Chapter 7.

3.4 Conclusion

Homology search is a key tool for understanding the role, structure, and biochemical function of genomic sequences. In this chapter, we surveyed existing approaches to homology search focusing on their efficiency, accuracy, and applicability to large-scale database search. We began with an overview of popular exhaustive methods, and described the BLAST algorithm in detail. We also presented our own detailed analysis of each stage of the algorithm, and discussed usage data provided by the NCBI. Next, we discussed some alternative approaches to search. We considered index-based approaches such as CAFE, and the advantages and disadvantages of such schemes. We also considered main-memory index systems that are highly successful for whole-genome alignment and the recent flurry of research into discontinuous seeds. We briefly described distributed approaches to search that offer substantial speed gains but require expensive hardware. We also investigated highly sensitive iterative search algorithms such as PSI-BLAST.

In Section 3.3, we discussed issues relating to genomic search. We began this section with a discussion of alignment significance and methods for calculating the likelihood of an optimal alignment score resulting from chance. We explained that alignment scores between unrelated sequences follow an extreme value distribution and how this trend can be used to derive an alignment E-value that reflects the statistical significance of an alignment.

We discussed methods for handling low-complexity regions in sequences including filtering algorithms such as SEG. Next, we showed how the SCOP database and ROC scores are commonly employed to assess the retrieval accuracy of homology search tools. We presented approaches that measure both sensitivity (the detection of homologous sequences) as well as selectivity (distinguishing between homologous and non-homologous sequences). Finally, we discussed redundancy in the form of near-duplicate entries in genomic data banks and the affects this has on search. We surveyed existing approaches to managing redundancy that typically prune sequences to create a representative database.

Despite a plethora of new approaches to homology search, including index-based schemes, discontinuous seeds and distributed approaches, the exhaustive BLAST algorithm remains the most successful, practical and versatile tool for searching large collections such as GenBank. Indeed, the survey we have presented in this chapter only covers a small selection of approaches that have been proposed in the past decade. Many new tools have been developed where the authors claim improvements over algorithms such as BLAST but the approach fails to find widespread appeal. Miller [2001] recently commented that several areas within bioinformations, such as homology search, are cluttered with mediocre tools that have not been thoroughly evaluated or compared. Miller believes this is because many researchers "find it much easier and more fun to develop a new program than to adequately verify that it actually improved upon earlier work" and that this can hinder, rather than assist, the research efforts of the biomedical community.

In the remainder of this thesis we present new approaches to homology search, and apply our new schemes to the hugely popular and successful BLAST algorithm with a thorough assessment of each concept. We aim to improve the efficiency of BLAST with no effect on search accuracy — we believe that faster search times are welcomed by biologists, but not at the expense of search accuracy. Indeed, many new schemes have been proposed that provide less sensitivity search but faster search times, such as CAFE, MEGABLAST and BLAT, none of which have found the same level of appeal as BLAST.

In Chapter 4 we describe three improvements to the gapped alignment stage of BLAST that roughly halve the time taken to align protein sequences. In Chapter 5 we describe a new data structure for protein hit detection that provides an 41% speed gain for the first stage of BLAST. In Chapter 6 we present methods for comparing nucleotide sequences in their compressed form that more than doubles the speed of BLASTN searches. Finally, we present a new scheme for managing redundancy in Chapter 7 that increases search speed by a further 22%, reduces collection size by 27% and provides more meaningful search results.

Importantly, we show that none of our schemes affect search accuracy.

When combined, our improvements roughly double the speed of nucleotide and protein BLAST searches with no significant change in accuracy. Our methods have been integrated into our own release of the BLAST software that is freely available for download at `http://www.fsa-blast.org/`. Further, the ideas presented in this thesis are not only applicable to BLAST; most of the concepts are general and can be easily adapted for use with other homology search tools.

Chapter 4

Improved Gapped Alignment

Despite the success of BLAST, there have been no fundamental changes to the algorithm since 1997. Several new approaches to genomic search have been proposed, however none offer the same degree of sensitivity and speed for searching large collection such as GenBank. Further, most of these new schemes are distinct in their approach to the first stage of homology search where short matching regions between the query and collection are identified. For example, index-based techniques such as CAFE are unique in their use of an inverted index to identify matching words. Discontinuous seeds are another recent, novel approach to identifying short, similar regions between the query and the collection. However, there has been little focus on improving the final stages of search methods such as BLAST that compute alignments.

We showed in Section 3.1.3 that computing gapped alignments consumes an average of 32% of the total processing time for protein sequence BLAST searches. Therefore, optimisation and innovation in the final stages of BLAST warrants investigation. This chapter proposes two such innovations, *semi-gapped alignment* and *restricted insertion alignment*. Both reduce the computational cost of alignment with no detectable effect on accuracy.

Semi-gapped alignment is a fundamental, new step in the BLAST algorithm. This step compromises between ungapped and gapped alignment: insertions and deletions are permitted only every N residues, that is, gaps are allowed but not always at the optimal position. In the overall BLAST process, semi-gapped alignment follows the ungapped alignment stage, aiming to efficiently and accurately reduce the number of gapped alignments required in the stage that follows. When carefully parameterised, this new technique reduces the average time taken to compute gapped alignments in the third and fourth stages of BLAST by 40% without affecting accuracy.

Restricted insertion alignment is an heuristic that can be applied to semi-gapped or gapped alignment. We have observed that in optimal alignments, insertions in one sequence are very rarely adjacent to insertions in the other. By preventing this event, it is possible to make a saving in computation following each alignment between two residues that occurs through insertion or deletion. This reduces the time taken for the gapped alignment stages in BLAST by around 8%. When semi-gapped alignment is used together with restricted insertion alignment and the optimisation we describe next, the speed of the gapped alignment stage in BLAST is more than doubled.

Our improvements to BLAST include an overlooked optimisation described by Zhang et al. [1997]. This optimisation was proposed to reduce the number of accesses to previously computed values. However, we observe that this optimisation permits a rearrangement of the recurrence relation used to compute gapped alignments, reducing the number of arithmetic and comparison operations per alignment with no effect on the result. We describe and explain this optimisation, and show that it reduces the cost of the alignment stages by around 20%. The NCBI implementation of BLAST does not use this optimisation.

This chapter is organised as follows. Section 4.1 describes our two new gapped alignment techniques, and illustrate how they can be employed by BLAST. We also present the optimised recurrence relations and explain how they reduce the computation involved per alignment. In Section 4.2, we present and discuss the results of comparing our new techniques to the existing methods used by BLAST. Our analysis is based on our own implementation of the gapped alignment stages that we have integrated into NCBI-BLAST. Finally, Section 4.3 presents a summary of this work. A preliminary version of the results and discussions presented in this chapter appeared in Cameron et al. [2004].

4.1 New Approaches to Gapped Alignment

In this section, we propose three improvements to the gapped alignment stages of BLAST. The first is an optimisation to the dynamic programming recurrence relations used to align sequences that reduces the computation for each cell in the alignment matrix with no effect on the result. The second is a new stage in the BLAST algorithm that filters alignments between the second and third stages. The third aims to reduce the time taken to generate gapped alignments in the third stage by disallowing unlikely events. All of the novel approaches differ from traditional gapped alignment by reducing the computation required for each cell in an alignment matrix. This approach is orthogonal to the dropoff technique already employed

by BLAST, which instead limits the number of cells that need to be processed. Therefore, a significant advantage of our novel techniques is that they can be used in combination with the existing dropoff heuristic. We report results of integrating our alignment scheme into the final stages of BLAST in Section 4.2.

4.1.1 Optimised gapped alignments

In Section 2.2.5 on page 32 we describe Gotoh's algorithm [1982] for globally aligning sequences using affine gap costs. BLAST uses Gotoh's algorithm to perform a gapped extension in each direction from a seed point, as discussed in Section 3.1.3 on page 62. In this section, we describe an optimisation to Gotoh's algorithm that was previously employed by Zhang et al. [1997] for aligning a protein sequence with a DNA sequence. We show that this optimisation can also be applied to BLAST, and that it affords a reduction in computation for some cells in the alignment matrix. The optimisation has not been previously applied to BLAST, or any of the other popular search tools described in Chapter 3 to our knowledge, most of which use Gotoh's algorithm to align sequences.

The following recurrence relationships (with base cases omitted for clarity of presentation) are used by BLAST to perform gapped alignments between the query x and a collection sequence y in the third stage:

$$
\begin{aligned}
M(i,j) = &\quad B(i-1,j-1) + s(x_i, y_j) \\
B(i,j) = &\quad \max \begin{cases} I_x(i,j) \\ I_y(i,j) \\ M(i,j) \end{cases} \\
I_x(i,j) = &\quad \max \begin{cases} B(i-1,j) - d \\ I_x(i-1,j) - e \end{cases} \\
I_y(i,j) = &\quad \max \begin{cases} B(i,j-1) - d \\ I_y(i,j-1) - e \end{cases}
\end{aligned}
$$

where $M(i,j)$, $I_x(i,j)$, $I_y(i,j)$, and $B(i,j)$ represent the best score for an alignment ending at $[i,j]$ with a match, an insertion in x, an insertion in y, or any of these events, respectively. Once again, $s(x_i, y_j)$ is the score resulting from aligning the i^{th} residue of x and the j^{th} residue of y, d is the penalty incurred for the first insertion in a gap and e is the penalty incurred for each subsequent insertion.

Consider now the computational cost of computing a gapped alignment using this recurrence. The processing of each cell requires arithmetic operations — for the addition of

match scores or subtraction of gap penalties — as well as comparisons for determining the maximum of either two or three values. Therefore, a careful implementation of the rules results in an algorithm that uses five arithmetic operations and four comparisons for each cell in the matrix. (Two comparisons are required to find the maximum of three values.)

Zhang et al. [1997] propose an optimisation where $I_x(i,j)$ and $I_y(i,j)$ store scores for $[i+1,j]$ and $[i,j+1]$ respectively. The motivation of this is to reduce accesses to previously computed values. However, there is an important additional advantage that — to our knowledge — has not been previously observed: it permits rule reorganisation that can reduce the computation required to produce gapped alignments, leading to the savings in computation that we discuss next. In addition, the reorganisation forms the first step in our new semi-gapped alignment scheme.

Let the values of I_x and I_y include the gap penalty associated with the next insertion in the alignment. That is, $I_x(i,j)$ represents the best score for an alignment ending at $[i+1,j]$ with an insertion with respect to y, and $I_y(i,j)$ represents the best score for an alignment ending at $[i,j+1]$ with an insertion with respect to x. With this modification, the recurrence relations described previously in this section are modified as follows:

$$M(i,j) = B(i-1,j-1) + s(x_i, y_j) \tag{1}$$

$$B(i,j) = \max \begin{cases} I_x(i-1,j) \\ I_y(i,j-1) \\ M(i,j) \end{cases} \tag{2}$$

$$I_x(i,j) = \max \begin{cases} M(i,j) - d \\ I_x(i-1,j) - e \end{cases} \tag{3}$$

$$I_y(i,j) = \max \begin{cases} M(i,j) - d \\ I_y(i,j-1) - e \end{cases} \tag{4}$$

These rules afford a performance advantage over the original recurrence relations in two ways. First, the processing of each cell requires four instead of five arithmetic operations, since the value of $M(i,j) - d$ in equation (3) can be reused in equation (4). Second, the number of comparisons that need to be performed can be reduced from four to three for some cells. To illustrate this, consider the case where the computation of $B(i,j)$ reveals that $I_x(i-1,j) \geq M(i,j)$, that is, when $I_x(i-1,j)$ is found to be the largest of the three values in equation (2). In this event, it can be deduced that $I_x(i-1,j) - e > M(i,j) - d$ since $e < d$, avoiding the need to compare these two values when calculating $I_x(i,j)$ in

equation (3). Similarly, the computation of $B(i,j)$ may reveal instead that $I_y(i,j-1)$ is the largest of the three values considered in equation (2). In this case, it can be deduced that $I_y(i,j-1) \geq M(i,j)$ and, therefore, that $I_y(i,j-1) - e > M(i,j) - d$. This avoids the need to compare these two values when calculating the value of $I_y(i,j)$ in equation (4). For this work, we have implemented this more efficient recurrence relation and used this as a baseline in our experiments in Section 4.2.

4.1.2 Semi-gapped alignment

In this section, we propose a new *semi-gapped* algorithm that compromises between the speed of ungapped alignment and the accuracy of gapped alignment. Our motivation is to add a new stage in the BLAST algorithm that efficiently and accurately reduces the subset of sequences identified by ungapped alignment to a very small set that are subsequently aligned using the computationally expensive gapped alignment stage. Specifically, we aim to reduce the computation at each cell in the alignment matrix while still producing alignment scores similar to those found using the Gotoh algorithm.

Our idea is to restrict where insertions and deletions can occur in an alignment, leading to an approach that combines the features of fast, heuristic ungapped alignment and the slower, more rigorous gapped alignment. The basic approach is as follows: we allow insertions in sequence y only at every N^{th} character (characters y_j, where $j \equiv 0, \text{modulo}\, N$) and insertions in sequence x at every N^{th} character (characters x_i, where $i \equiv 0, \text{modulo}\, N$). Figure 4.1 illustrates the effect these two constraints have on how values are derived for each cell in the matrix. We explain the rationale behind this approach next, and discuss the reduction in computational cost and the effect of varying N later.

We propose that semi-gapped alignments be attempted only after an ungapped alignment exceeds a cutoff, and that a successful semi-gapped alignment be used to trigger a further gapped alignment as the final stage of BLAST. We believe that this semi-gapped alignment stage should be an effective and efficient additional step in BLAST for three reasons:

1. We have observed that most gapped alignments calculated using affine gap costs contain long ungapped regions separated by gaps of more than one indel that move the alignment from one matrix diagonal to another; semi-gapped alignment still permits these gaps, but forces the gap start and end points to be moved. In the worst case, the optimal position for opening the gap is midway between rows or columns where the insertion is allowed, and the gap position must be moved by $\lfloor \frac{N}{2} \rfloor$ residues from

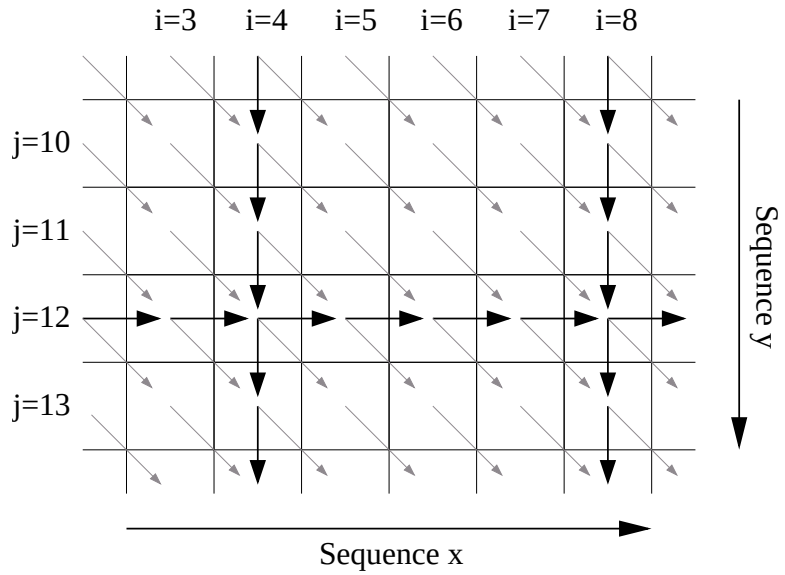

Figure 4.1: Portion of the dynamic programming matrix used to perform a semi-gapped alignment where $N = 4$. Arrows indicate how values are derived for each cell. For most cells only the match operation is considered. For columns where $i = 4$ or $i = 8$, insertions with respect to x are also considered. Similarly, insertions with respect to y are also considered for the row where $j = 12$.

this optimal position. On average, a move of $\frac{\lceil \frac{N}{2} \rceil \times \lfloor \frac{N}{2} \rfloor}{N}$ residues is required. Unless the optimal gap is surrounded by high-scoring matches or the neighbouring ungapped regions are flanked by low-scoring matches, this shift has little influence on the overall score.

2. The scoring penalty associated with opening and extending a gap often outweighs any reduction in score that may be caused by opening the gap at a sub-optimal location.

3. It has been observed by Altschul [1998] that when aligning distantly related protein sequences, "conserved residues frequently fall into ungapped blocks separated by relatively non-conserved regions". This suggests that gaps often occur in less conserved regions and that the exact location of the gap is of less importance, given that the surrounding matches are likely to be low scoring anyway.

We now explain how the constraints imposed by our semi-gapped scoring technique affect the recurrence relations and the processing required for each cell in the alignment matrix.

	I_x allowed	I_x not allowed
I_y allowed	$M(i,j) = B(i-1,j-1) + s(x_i, y_j)$ $B(i,j) = \max \begin{cases} I_x(i-1,j) \\ I_y(i,j-1) \\ M(i,j) \end{cases}$ $I_x(i,j) = \max \begin{cases} M(i,j) - d \\ I_x(i-1,j) - e \end{cases}$ $I_y(i,j) = \max \begin{cases} M(i,j) - d \\ I_y(i,j-1) - e \end{cases}$	$M(i,j) = B(i-1,j-1) + s(x_i, y_j)$ $B(i,j) = \max \begin{cases} I_y(i,j-1) \\ M(i,j) \end{cases}$ $I_y(i,j) = \max \begin{cases} M(i,j) - d \\ I_y(i,j-1) - e \end{cases}$
I_y not allowed	$M(i,j) = B(i-1,j-1) + s(x_i, y_j)$ $B(i,j) = \max \begin{cases} I_x(i-1,j) \\ M(i,j) \end{cases}$ $I_x(i,j) = \max \begin{cases} M(i,j) - d \\ I_x(i-1,j) - e \end{cases}$	$B(i,j) = B(i-1,j-1) + s(x_i, y_j)$

Table 4.1: Recurrence relations for each cell type in semi-gapped alignment. For all recurrence relations, cells where $i = 0$ or $j = 0$ are initialised to $-\infty$ except for the alignment starting point $[0, 0]$ that is initialised to zero

Observe that, depending on its location in the matrix, each cell in a semi-gapped alignment permits one of four cases:

1. Insertion in either the query or collection sequence is permissible, that is, the standard gapped alignment recursion applies; or,
2. Only insertion with respect to the query sequence is allowed; or,
3. Only insertion with respect to the collection sequence is allowed; or,
4. No insertion is permissible, that is, the standard ungapped alignment recursion applies.

Therefore, different recurrence relations are applicable depending on what operations are permitted. We consider these operations with respect to the four cases next.

The semi-gapped recurrence relations are shown in Table 4.1. The original recurrence described in Section 4.1.1 is used for the first case, that is, for cells where $j \equiv 0, \text{modulo}\, N$ and $i \equiv 0, \text{modulo}\, N$, and where insertion in both sequences is allowed; these relations are shown at the intersection of the row labelled "I_y allowed" and the column "I_x allowed".

	I_x allowed		I_x not allowed	
I_y allowed	Arithmetic operations =	4	Arithmetic operations =	3
	Comparisons =	3 or 4	Comparisons =	1 or 2
	Cell type frequency =	$\frac{1}{N^2}$	Cell type frequency =	$\frac{N-1}{N^2}$
I_y not allowed	Arithmetic operations =	3	Arithmetic operations =	1
	Comparisons =	1 or 2	Comparisons =	0
	Cell type frequency =	$\frac{N-1}{N^2}$	Cell type frequency =	$\frac{(N-1)^2}{N^2}$

Table 4.2: Frequency of each type of cell in a semi-gapped alignment and number of associated operations.

When insertion in sequence y is not permitted, that is, $j \not\equiv 0, \text{modulo}\, N$ and the second case applies, then there is no need to compute the value of I_x and the recursion is simplified to that shown at the intersection of the row labelled "I_y not allowed" and the column labelled "I_x allowed". Similarly, when $i \not\equiv 0, \text{modulo}\, N$, then the third case applies and there is no need to compute the value of I_y and the recursion is simplified to that at the intersection of the row "I_y allowed" and the column "I_x not allowed". Last, when only ungapped alignment is permitted, the fourth case applies and the simple recursion at the intersection of "I_y not allowed" and "I_x not allowed" is used.

Computational Costs

Table 4.2 shows the number of arithmetic operations and comparisons required for each of the four different recurrence relations. In addition, it shows how varying N affects the number of cells in an alignment matrix that are computed using each of the four relations. For example, the table shows that in the regular gapped alignment case — shown as the intersection of "I_x allowed" and "I_y allowed" — four arithmetic operations are required per cell, and either three or four comparisons (as described for the optimisation in Section 4.1.1) are required. The table also shows that only 1 in every N^2 cells require this fully-gapped alignment. For the cases where I_x is allowed but I_y is not, and where I_y is allowed but I_x is not, only three arithmetic operations and two comparisons are required. We can also apply optimisations similar to those used in the regular gapped alignment case that reduces the number of comparisons for some cells further, from two down to one. Importantly, ungapped alignment is inexpensive — requiring only one arithmetic operation — and for $N \geq 4$ the

N	1	2	3	4	5	6
Arithmetic operations	4.00	3.75	3.22	2.94	2.76	2.64
Comparisons	4.00	3.00	2.33	2.00	1.80	1.67

N	7	8	9	10	11	12
Arithmetic operations	2.55	2.48	2.43	2.39	2.36	2.33
Comparisons	1.57	1.50	1.44	1.40	1.36	1.33

Table 4.3: Average number of operations per cell for varying values of N.

majority of cells are in this class.

When $N \geq 2$, an additional overhead is required to determine the class of each cell. Since the value of j does not change while processing each row in the matrix, there is no need to determine for each cell whether insertion in sequence y is allowed; this can be done once for each row at negligible cost. In contrast, the value of i must be checked for each cell individually to determine if $i \equiv 0, \text{modulo}\, N$. This involves an additional arithmetic operation and comparison per cell to determine its class.

Table 4.3 illustrates how the average number of operations per cell varies as N is increased from 1 to 12, including the overhead of determining the class of each cell. When $N = 1$, only gapped alignment is used and, as N increases, the average computational cost per cell decreases. However, the reduction in computational cost as N is increased has an effect on accuracy. Therefore, similarly to other parameters in BLAST, the value of N must be carefully chosen. We report experiments with varying values of N in Section 4.2.

Triggering Gapped Alignment

Semi-gapped alignment is an additional stage in BLAST that follows ungapped alignment and precedes gapped alignment. Therefore, similarly to all other non-final stages, a criterion must be established to trigger processing of an alignment in a subsequent stage.

After experimentation, we found the following approach is effective for deciding whether semi-gapped alignments should be passed to the final, gapped alignment stage in BLAST. We score each candidate sequence using semi-gapped alignment and, if the score exceeds $R \times S2$, then we proceed to gapped alignment. The value of $S2$ is the existing BLAST nominal score required to achieve cutoff that was discussed in Section 3.3.1, and R is a new constant such

that $0 < R \leq 1$. Using the existing $S2$ constant has an important advantage: if the score from semi-gapped alignment exceeds $S2$, then there is no requirement for gapped alignment and the sequence can be passed to stage four of the BLAST algorithm where the refined score and traceback information is determined. We report experiments that vary R in Section 4.2.

Figure 4.2 illustrates how the semi-gapped alignment stage is incorporated into the BLAST algorithm. First, an ungapped extension is performed (Figure 4.2a) and the resulting alignment scores above an $S1$ cutoff score of 40. Second, a semi-gapped alignment (Figure 4.2b) is performed. The light grey lines highlight the columns and rows in the semi-gapped alignment matrix where insertions are allowed; we show that the two gaps in the alignment occur in these permitted regions. The resulting score of 87 is recorded, and because this lies between $R \times S2 = 63$ and $S2 = 90$, a gapped alignment is required to determine if the alignment scores more than $S2$. The resulting gapped alignment (Figure 4.2c) scores 95, and is therefore statistically significant and displayed to the user.

4.1.3 Restricted insertion alignment

In this section, we describe our third technique to improve gapped alignment in BLAST. This novel approach — which we refer to as *restricted insertion alignment* — is orthogonal to semi-gapped alignment, and can be applied either to it or to the gapped alignment stage in BLAST.

Restricted insertion alignment aims to reduce computation for unlikely evolutionary events. We have experimentally observed that, in optimal gapped alignments, gaps in one sequence are very infrequently adjacent to gaps in the other. Figure 2.9 on page 34 illustrates the rare case where two gaps occur adjacent to one another in the optimal alignment between two sequences. In our experiments with 100 random sequences from the GenBank NR protein database described in Section 3.1.3 we found that this event is extremely rare: less than 0.02% of gapped alignments generated by BLAST contain adjacent gaps. Importantly, in 99% of these rare cases, an alignment that does not contain adjacent gaps and scores no more than 5% less than the optimal also exists. As a result, only 17 of the 1,013 million alignments scoring below an E-value cutoff of 10 are not detected when restricted insertion is imposed.

We propose taking advantage of the infrequency of adjacent gaps, and propose that these are not permitted in the alignment process. This works as follows. If the best score for the current cell $[i, j]$ is derived from $I_y(i, j-1)$, that is, from an insertion with respect to

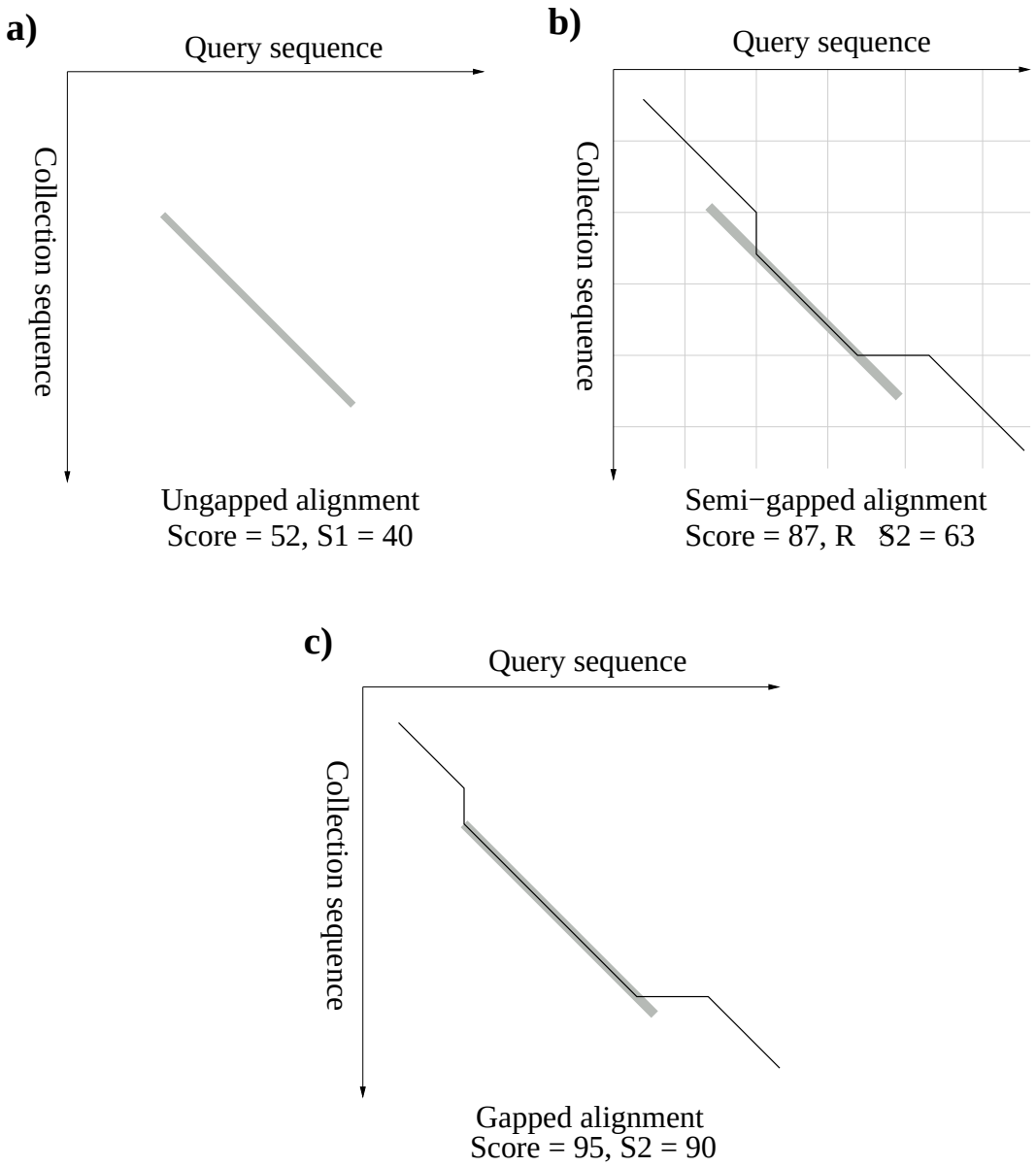

Figure 4.2: The new process for scoring sequences. a) First, an ungapped extension is performed. b) If the resulting alignment scores above the $S1$ cutoff, a semi-gapped alignment is then performed. The grid overlay shows the rows and columns where insertions are permitted. c) Finally, if the semi-gapped alignment scores between $R \times S2$ and $S2$ then a gapped alignment is performed. In this example, the alignment scores above the cutoff at each stage, and its final score is high enough for it to be displayed to the user.

sequence x, then the value of $I_x(i,j)$ is not calculated. Similarly, the value for $I_y(i,j)$ is not determined if the best score for the cell is derived from $I_x(i-1,j)$.

With these new constraints, the recurrence relations from Section 4.1.1 are rewritten as follows:

$$\begin{aligned} M(i,j) = &\quad B(i-1,j-1) + s(x_i, y_j) \\ B(i,j) = &\quad \max \begin{cases} I_x(i-1,j) \\ I_y(i,j-1) \\ M(i,j) \end{cases} \end{aligned}$$

$$\begin{aligned} \textit{if } B(i,j) = &\quad I_y(i,j-1) \\ \textit{then } I_x(i,j) = &\quad -\infty \\ \textit{else } I_x(i,j) = &\quad \max \begin{cases} M(i,j) - d \\ I_x(i-1,j) - e \end{cases} \end{aligned}$$

$$\begin{aligned} \textit{if } B(i,j) = &\quad I_x(i-1,j) \\ \textit{then } I_y(i,j) = &\quad -\infty \\ \textit{else } I_y(i,j) = &\quad \max \begin{cases} M(i,j) - d \\ I_y(i,j-1) - e \end{cases} \end{aligned}$$

The outcome of this new restriction is a saving in computation per cell. For cells where either $M(i,j) \leq I_y(i,j-1)$ or $M(i,j) \leq I_x(i-1,j)$, there is a reduction from four arithmetic operations and three comparisons to two arithmetic operations and two comparisons.

Our restricted insertion alignment algorithm is similar to the two state variation of the Gotoh algorithm [Durbin, 1998] described in Section 2.2.5 on page 33, which stores only the larger of I_x and I_y for each cell in the matrix. However, the aim and effect of this other approach is different: it aims to reduce main-memory requirements for alignments with traceback, and the effect is that adjacent gaps are treated as a single gap with only one opening penalty. There are no computational savings in the approach and it does not offer any savings for score-only alignment without traceback. However, importantly, Durbin observes — in support of both our and their approach — that heuristics for adjacent gaps rarely affect the resulting alignment.

We have applied the restricted insertion approach to the semi-gapped and gapped alignment filtering stages of BLAST. However, when recording optimal alignments in the final stage using the traceback approach we do not employ this new technique. As a result, adjacent gaps may occur in the final alignments that are reported to the user.

4.1.4 Summary

In this section, we presented three new approaches to gapped alignment in BLAST. First, we described an optimisation to the original recurrence relations that reduces the computation for cells in the alignment matrix with no effect on the result. Next, we introduced a new method for aligning sequences called semi-gapped alignment that compromises between the speed of ungapped alignment and sensitivity of gapped alignment. Our approach permits insertions at only some positions in the alignment matrix, resulting in less computation per cell. Our new method is employed as an addition filtering stage in BLAST between the ungapped and gapped alignment stages. Finally, we described an optimisation to gapped alignment called restricted insertion that reduces computation by dismissing unlikely events.

In the next section, we experimentally compare our new methods to the existing alignment algorithms used by BLAST. We show that our methods more than halve the time taken to align sequences in the third and fourth stages.

4.2 Results

In this section, we present the results of experiments with our semi-gapped and restricted insertion alignment schemes. We begin by describing the collections and measurement techniques we used to quantify performance, and then present an overall comparison of results. The result summary is followed by detailed presentations of the effect of parameter choices for our schemes.

4.2.1 Collections, Measurements, and Environment

For our evaluation, we measure search accuracy using the classifications in the SCOP database based on the approach described in Section 3.3.3 on page 97. We used version 1.65 of the ASTRAL Compendium for Sequence and Structure Analysis [Chandonia et al., 2004] that was released on 19 December 2003. The database has been filtered so that sequences with greater than 90% identity are removed, resulting in a collection that contains 8,759 structurally classified protein sequences that have each been assigned to one of 1,293 superfamilies.

Each sequence from the ASTRAL database was extracted and used to search the entire collection. An alignment with a sequence from the same superfamily as the query is considered correct and an alignment with a sequence from a different superfamily is considered

incorrect. For our experiments, the ranked list of alignments returned by BLAST were scored using the ROC_{50} measure, and the reported score represents the average across all queries.

Unfortunately, the SCOP database is too small to allow meaningful comparison of the speed of alignment algorithms. Therefore, we used the GenBank non-redundant protein database for timing experiments and, as described previously, randomly selected 100 sequences from the collection that were then used as queries to search that collection. We used the 30 June 2004 release of GenBank NR, which contains 1,873,745 sequences in around 622 megabytes of sequence data.

When searching the ASTRAL database, we use the size of the GenBank NR database for the calculation of E-values. We did this to ensure that the cutoff scores used when measuring the speed of searching NR and accuracy of searching ASTRAL are comparable. Without this adjustment, the values of $S2$ (and, therefore, $R \times S2$) are lower when measuring the accuracy of the semi-gapped alignment technique which, in turn, increases the number of alignments that are rescored and improves ROC scores. Using the same effective database size when measuring speed and accuracy avoids this bias.

For our timing experiments with GenBank NR, we ran each of the 100 queries and recorded all high-scoring ungapped extensions reported using the first two stages of NCBI-BLAST with default parameters. These ungapped alignments were then used as input to all algorithms we tested, and timings reported are for all stages following ungapped alignment only; these timings therefore include all non-ungapped alignment stages, collecting traceback information, and preparation of alignments for display in the BLAST standard format. The best elapsed time of three runs was reported for each query.

The results presented are based on experiments carried out on an Intel Pentium 4 2.8GHz workstation with one gigabyte of main-memory while the machine was under light-load, that is, no other significant processes were running. All schemes — with the exception of NCBI-BLAST — include our gapped alignment recursion optimisation proposed in Section 4.1.1. For baseline comparisons, we used NCBI-BLAST version 2.2.8. All code was compiled with the same compiler flags and optimisations as NCBI-BLAST.

Our implementations of the gapped alignment stages used the same Karlin-Altschul statistics to score the alignments as NCBI-BLAST, which were described in detail in Section 3.3.1 on page 91. Specifically, the pre-computed values of λ, K, α and β were taken from NCBI-BLAST version 2.2.8. The composition of collection sequences was not used for scoring, that is, we did not use the composition-based statistics described by Schaffer et al. [2001] that are by default disabled in NCBI-BLAST. No filtering was applied to the query sequences.

Scheme	GenBank NR		ASTRAL
	Time (secs)	Alignments Reported	ROC_{50}
Combined	4.90	31,160	0.339
Semi-gapped alignment only	5.02	31,160	0.339
Restricted insertion only	7.67	31,163	0.339
Baseline	8.34	31,163	0.339
NCBI-BLAST	10.34	31,161	0.339

Table 4.4: Average runtime and number of high-scoring alignments for 100 queries on the the GenBank non-redundant database, and SCOP ROC_{50} scores for the ASTRAL collection. All alignment techniques use default parameters.

4.2.2 Overall Results

Table 4.4 shows a comparison of our techniques to NCBI-BLAST. Our results show that the combination of semi-gapped and restricted insertion alignment — labelled as *combined* — better than halves the average time taken to carry out alignments compared to NCBI-BLAST. On average, over five seconds is saved per query when searching GenBank NR. Importantly, all schemes we tested have indistinguishable ROC scores, and this is supported by the total number of alignments reported from the queries on the GenBank NR collection. (We report the total number of alignments returned from the GenBank search as an indicator of overall accuracy performance, and have found that these alignments are almost identical for all schemes. However, we believe that ASTRAL ROC scores are a more definitive indicator of performance.)

We compared our schemes to our own baseline and NCBI-BLAST. Our baseline — which is optimised by the recursion reorganisation described in Section 4.1.1 — is around 20% faster than NCBI-BLAST. However, importantly, our heuristic approaches are still much faster: in particular, the combination of schemes is around 40% faster than our optimised baseline. We have also found that the semi-gapped alignment stage is efficient and effective: it discards an average of 87% of the high-scoring database sequences from Stage 1 and each semi-gapped alignment is performed in less than half the time of a gapped alignment.

In the experiments reported in Table 4.4, default BLAST parameters were used. These include a gapped trigger score of 22.0 bits (which affects $S1$), scoring dropoff of $X = 15.0$

	E=100		E=10		E=0.1	
	ROC_{50}	Time	ROC_{50}	Time	ROC_{50}	Time
Combined	0.360	5.67	0.339	4.90	0.300	4.39
Semi-gapped alignment only	0.360	5.85	0.339	5.02	0.300	4.46
Restricted insertion only	0.360	7.78	0.339	7.67	0.300	7.53
Baseline	0.360	8.46	0.339	8.34	0.300	8.19
NCBI-BLAST	0.360	10.36	0.339	10.34	0.300	10.37

Table 4.5: Average query evaluation time in seconds for searching the GenBank non-redundant database, and results of a SCOP accuracy test for ROC_{50}. Each alignment technique is reported for a range of E-value cutoffs.

bits, an E-value cutoff of $E = 10.0$, and a maximum number of alignments to be reported of 500. For semi-gapped alignment, we use $N = 10$ and $R = 0.68$, and open and extend gap penalties of $o_s = 7$ and $e_s = 1$ respectively. We discuss parameter choices further in the next sections.

Our schemes can be alternatively parameterised to improve accuracy while having runtimes similar to NCBI-BLAST. For example, by lowering the scoring required to trigger gapped alignment from the default value of 22.0 to 20.2 bits, the ROC_{50} score of the combination scheme increases from 0.339 to 0.342, and the average runtime increases to 10.24 seconds. However, since our primary aim is to reduce the computational cost of BLAST without affecting its accuracy, we do not discuss this in detail further here.

4.2.3 Varying the E-value

Table 4.5 shows the effect of varying the E-value cutoff on different schemes. As described in Section 3.3.1, an E-value represents the chance that an alignment with at least the same score would be found if a randomly constructed query with typical amino-acid composition was searched against a randomly generated database. BLAST uses a default E-value cutoff of 10, however a lower cutoff is often selected by users to reduce false positives that have chance similarities to the query. By default, the PSI-BLAST algorithm only considers alignments with an E-value below 0.002 for inclusion in the next iteration of search, because false positives have an even greater detrimental effect on search accuracy.

The results in Table 4.5 show that as E decreases, the reduction in processing costs

varies between different schemes; indeed, in unreported experiments with smaller values of E, this same trend continues. For semi-gapped alignment schemes, query evaluation times fall: when $E = 0.1$, the query evaluation is around 10%–25% faster than for $E = 100$, and almost two and a half times faster than NCBI-BLAST when $E = 0.1$. This is because a smaller cutoff increases $S2$, resulting in rescoring of fewer semi-gapped alignments using the slow gapped scheme. For the other schemes — NCBI-BLAST, our optimised baseline, and restricted insertion only — reducing E has almost no effect on speed, because none have the additional filtering step of semi-gapped alignment.

In terms of accuracy, the cutoff has the same effect on all schemes: the ROC_{50} scores achieved by all techniques are the same for all cutoffs we tested, including for cutoffs up to three magnitudes smaller than reported in Table 4.5. We also found that ROC scores vary depending on the E-value cutoff used. This is because a larger cutoff leads to an increase in the number of reported alignments which in turn improves ROC scores. Our results show a 17% reduction in ROC score for an E-value cutoff of 0.1 compared to the less selective cutoff of 100. This suggests that care must be taken when comparing homology search tools to ensure that measures of statistical significance and cutoff scores are comparable across different schemes. When performing a comparison, one tool may be unfairly penalised if a more stringent cutoff is applied.

We also measured the effect of varying the maximum number of reported alignments for each scheme. We found no distinguishable difference in accuracy between the schemes when a maximum of 5, 50, 500, or 5000 alignments were reported.

4.2.4 Varying N and R

Semi-gapped alignment is parameterised by two constants, N and R. The value of N controls the ratio of gapped to ungapped alignment: small values of N favour gapped alignment, and large values of N favour ungapped alignment. The value of R influences the number of semi-gapped alignments that are subsequently rescored using the gapped stage in BLAST. In our previous overall results, we report experiments with $N = 10$ and $R = 0.68$. This section shows how these values were derived experimentally, how the choices of N and R are dependent, and that they are robust when other parameters are varied.

Figure 4.3 shows how increasing the ratio of ungapped to gapped alignment affects alignment scores. As N increases, the score produced by semi-gapped alignment becomes increasingly lower, falling to around 15% less for values of $N \geq 11$. This is as expected: semi-gapped

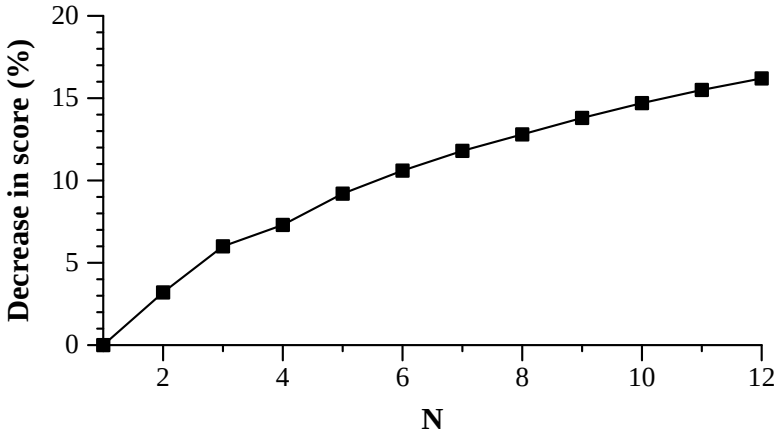

Figure 4.3: Average decrease in score between semi-gapped alignments and gapped alignments for varying values of N. The same gap penalties were used for both methods. Only gapped alignments with an E-value below 10 *were considered.*

alignment forces insertions to occur in suboptimal locations, and the average distance between the optimal gap location and the closest permitted gap location increases with N. However, as semi-gapped alignments are only performed to identify regions that must be rescored with gapped alignment, differing scores only affect which alignments are considered in the next stage and not what is returned to users. In practice, we have found that $N = 10$ affords an excellent tradeoff between speed and accuracy.

The score reduction effect is compounded by the dropoff technique, which processes only cells that score above a dynamic threshold. Decreasing scores from semi-gapped alignment leads to a reduction in the number of matrix cells processed and, in turn, this can lead to high scoring alignments not being considered during the semi-gapped alignment stage. As we show later, lowering the open gap penalty provides an effective solution to the problem.

Figure 4.4 shows the effect of varying R for different values of N. Each curve in the figure shows the accuracy and speed tradeoff for a fixed value of N but with varying R. For all curves, decreasing R improves the ROC score and increases average query evaluation time. Interestingly, for values of $N \leq 10$, the curve has a characteristic shape, where the ROC score improves significantly as R is decreased from one and then reaches a near-maximum for values of approximately $R \leq 0.5$. The point on each curve shows the setting $R = 0.68$, which we advocate as the default setting and use in all experiments reported throughout our results.

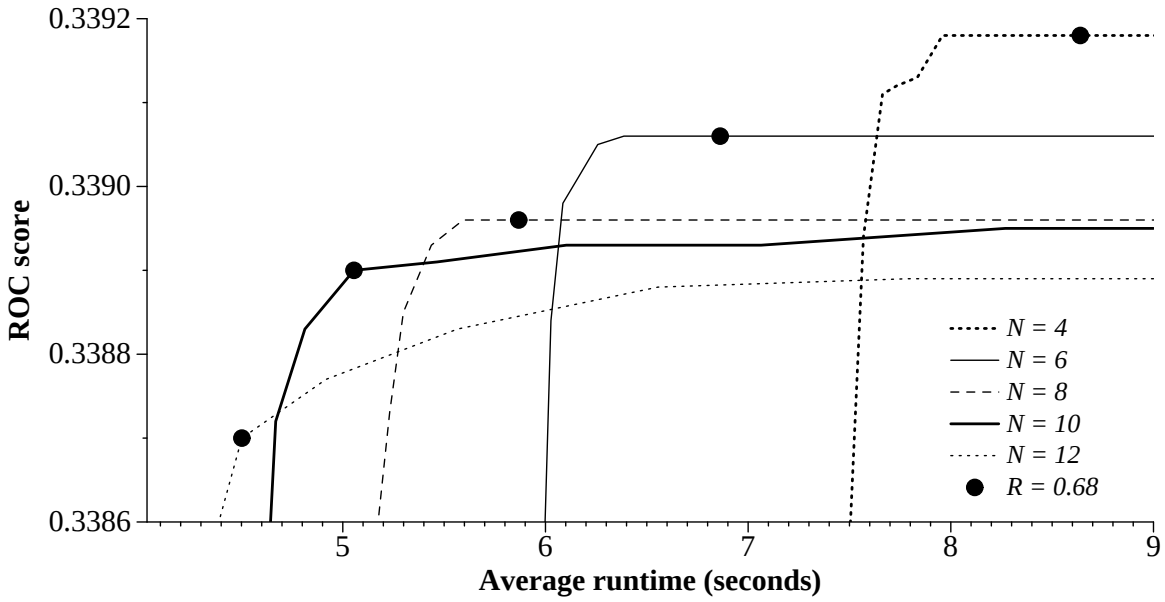

Figure 4.4: Accuracy versus query evaluation times for combined semi-gapped alignment and restricted insertion alignment using different values of N and R. The default value of $R = 0.68$ is highlighted on each curve.

4.2.5 Varying the Gap Open Penalty

As discussed in Section 4.1.2, our semi-gapped alignment algorithm additionally penalises gaps by forcing them to occur in suboptimal locations. Specifically, on average, gaps occur $\frac{\lceil \frac{N}{2} \rceil \times \lfloor \frac{N}{2} \rfloor}{N}$ residues from their optimal open position. To compensate for this, we have explored lowering the open gap penalty for semi-gapped alignment.

Let o_g and e_g denote the open gap penalty and extend gap penalty used for gapped alignment respectively. Similarly, let o_s and e_s denote the open and extend gap penalties used for semi-gapped alignment. Because gap length has no effect on the constraints imposed by semi-gapped alignment, we let $e_s = e_g$. However, for open gap penalties, we propose $o_s = o_g - C$, where C is a constant value that reduces the open cost for semi-gapped alignment. We use a normalized value for C to ensure that this compensation is comparable across different scoring schemes.

Table 4.6 shows the effect of varying the open gap penalty o_s and its related value of C. The combined scheme, and default values of $o_g = 11$, $N = 10$, and $R = 0.68$ are used for all results shown. In addition, because changing the open gap penalty affects the region explored by the dropoff technique, we have chosen a value of X for each value of C that

Semi-gapped open gap penalty (o_s)	6	7	8	9
C (bits)	6.53	6.15	5.76	5.38
Dropoff X parameter (bits)	14.4	15.0	16.5	19.2
ROC_{50} score	0.338	0.339	0.339	0.339
Time (secs)	5.35	4.93	5.23	6.18

Table 4.6: Average query evaluation time for varying semi-gapped alignment open gap penalties o_s and the related value C. For each penalty, a value of X that produces equivalent ROC_{50} scores was chosen.

provides an ROC score approximately equal to the baseline; this allows comparison of o_g values to identify the minimum query evaluation time. Our results show that $C \approx 6.0$ bits works well, that is, $o_s = 7$ is a suitable value for the BLAST default parameters.

In unreported experiments, we have also found that $C = 6.0$ bits, $N = 10$, and $R = 0.68$ provides a good compromise between accuracy and speed for most popular scoring schemes and choices of open gap penalty, o_g. We recommend and use $C = 6.0$ bits to calculate open gap penalties throughout our experiments.

4.2.6 Varying the Mutation Data Matrix

The two commonly used families of data mutation matrices are the PAM series and the BLOSUM series, which are both supported by BLAST and discussed in Section 2.4 on page 43. The BLOSUM62 substitution matrix is the most commonly used for BLAST searches, however other scoring matrices are included in the default BLAST distribution:

- the BLOSUM45 matrix is constructed from alignments between distantly related proteins and is suitable for detecting distant homology;
- the BLOSUM80 matrix is constructed from closely related proteins and is suitable for detecting close homology; and
- the older PAM30 and PAM70 matrices are still occasionally used because they can provide better sensitivity for searches conducted with short queries.

We consider the effect of using these matrices in this section.

Table 4.7 shows a comparison of the various gapped alignment techniques for the four additional scoring matrices; results for the BLOSUM62, with $o_g = 11$, $e_g = 1$, $o_s = 7$, and

Scoring matrix	BLOSUM45		BLOSUM80	
Gapped costs (o_g,e_g)	14,2		10,1	
Semi-gapped costs (o_s,e_s)	10,2		5,1	
	ROC_{50}	Time	ROC_{50}	Time
Combined	0.331	6.58	0.332	4.07
Semi-gapped only	0.331	6.73	0.332	4.21
Restricted insertion only	0.331	10.59	0.332	6.11
Baseline	0.331	11.23	0.332	6.72
NCBI-BLAST	0.331	14.31	0.330	7.95

Scoring matrix	PAM30		PAM70	
Gapped costs (o_g,e_g)	10,1		9,1	
Semi-gapped costs (o_s,e_s)	3,1		4,1	
	ROC_{50}	Time	ROC_{50}	Time
Combined	0.238	1.75	0.292	2.91
Semi-gapped only	0.238	1.84	0.293	3.01
Restricted insertion only	0.238	3.20	0.294	7.04
Baseline	0.238	3.59	0.294	7.91
NCBI-BLAST	0.236	3.87	0.293	9.85

Table 4.7: Comparison of the gapped alignment techniques when used in combination with the scoring matrices included with BLAST. *For each matrix, the recommended gap penalties were used with* $C = 6.0$.

$e_s = 1$ are reported in Section 4.2.2. For each matrix, the recommended open gap and extend gap penalties taken from the NCBI online version[1] were used and are listed in the table. As previously, we report ROC values for searching SCOP, and average query evaluation times for searching GenBank NR.

The results show that our schemes are robust for all matrices. As for BLOSUM62, the accuracy of our schemes compared to our baseline and NCBI-BLAST is mostly indistinguishable. For PAM70, the combined approach has an ROC score that is 0.001 less than NCBI-BLAST, but it is staggeringly more than three times as fast; simple parameter tuning can alter this

[1] See `http://www.ncbi.nlm.nih.gov/BLAST/`

tradeoff as required. For BLOSUM80, NCBI-BLAST has an ROC score 0.002 less than the other approaches, which we have attributed to a minor difference in the implementation of the dropoff scheme. Overall, our semi-gapped schemes are relatively faster for detecting distant homologs.

4.2.7 Summary

In this section, we presented the results of experimental evaluations for our new alignment methods. Our semi-gapped alignment method, combined with an optimised rearrangement of the recurrence relations and our novel restrict insertion heuristic more than halve the time taken to identify high-scoring gapped alignments in BLAST while retaining the same accuracy. We have carefully parameterised the semi-gapped stage and chosen values for N and R that maximise accuracy and speed. We have also shown that lowering the open gap penalty improves the accuracy of the method. The results demonstrate that our schemes are robust across a range of scoring schemes and E-value cutoffs.

4.3 Conclusion

Very little work has addressed the fundamental algorithmic steps that algorithms such as BLAST use to accurately and efficiently compute gapped alignments. In this chapter, we proposed two improvements to gapped alignment in BLAST. The first improvement is an new stage in the BLAST algorithm called semi-gapped alignment that efficiently identifies candidate sequences with a broad similarity to the query. The novel semi-gapped alignment algorithm permits gaps, but only at specific positions in the two sequences. The second improvement is a heuristic scheme that dismisses unlikely evolutionary events, called adjacent gaps, to reduce computation involved in aligning sequences. We show experimentally that — together with an optimisation of the alignment recursion — these techniques halve the query evaluation time of the gapped alignment stages of BLAST with negligible effect on accuracy. We conclude that these steps are a valuable addition to BLAST and have integrated them into our own version of the tool called FSA-BLAST (which stands for *Faster Search Algorithm* - BLAST) and is available for download from `http://www.fsa-blast.org/`.

In the next chapter, we propose a new approach to the first, hit detection stage of BLAST. We focus on optimising the lookup table used to identify hits in protein searches and investigate the effect of parameter choices such as word length on this initial stage.

Chapter 5

Protein Hit Detection

The first stage of BLAST — and many other homology search tools — involves identifying *hits*: short, high-scoring matches between the query sequence and the sequences from the collection being searched. The hit detection stage represents a large proportion of the total search time; our results in Section 3.1.3 on page 55 show that this first stage consumes 37% of the total search time for protein queries, and 85% of the search time for nucleotide queries.

It is therefore not surprising that hit detection has received considerable attention at late with a range of new approaches to this first stage of homology search. In Section 3.2 on page 76 we describe several such approaches, including indexed-based methods and the use of discontinuous seeds to detect inexact matches between the query and collection subsequences. Unfortunately, these new approaches have limited application and are either unsuitable for searching large collections such as GenBank or are significantly less sensitive, for reasons presented in Section 3.2. Further, as discussed in Section 3.2.3, discontinuous seeds have only been applied in practice to nucleotide data and do not appear to be as effective for protein searches. As a result, the BLAST approach of scanning the entire collection for exact matches of a fixed length W remains the most successful approach to homology search.

The definition of a hit and how they are identified differs between protein and nucleotide searches, mainly because of the difference in alphabet sizes. For example, for BLAST nucleotide search, an exact match of length eleven is required. However, for protein search, it requires a match of length three and inexact matches are permitted. Indeed, BLAST uses an algorithmically different approach to hit detection for protein and nucleotide searches, as discussed in Section 3.1.3.

In this chapter, we investigate algorithmic optimisations that aim to improve the speed of

protein search. Specifically, we investigate the choice of data structure for the hit matching process, and experimentally compare an optimised implementation of the current NCBI-BLAST codeword lookup approach to the use of a deterministic finite automaton (DFA). Our DFA is highly optimised and carefully designed to take advantage of CPU cache on modern workstations. Our results show that the DFA approach reduces total search time by 6%–30% compared to codeword lookup, depending on platform and parameters. This represents a reduction of around 41% in the time required by BLAST to perform hit detection; this is an important gain, given the millions of searches that are executed each day. Further, the new approach can be applied to a range of similar protein search tools. We also explore the effect of varying the word length and neighbourhood threshold on the hit detection process.

This chapter is organised as follows. We describe the codeword lookup structure used for hit detection in BLAST and experiment with varying the word size and match threshold in Section 5.1. In Section 5.2 we describe our new approach based on a carefully optimised DFA structure. In Section 5.3, we compare our new DFA approach to the original codeword lookup scheme. Finally, we provide concluding remarks in Section 5.4. A preliminary version of the results and discussions presented in this chapter appeared in Cameron et al. [2006c].

5.1 Identifying Hits in BLAST

As discussed in Section 3.1.3, BLAST identifies high-scoring matches of a fixed length W between the query sequence and sequences in the collection as the first stage of search. A match is considered high-scoring if the pair of words (subsequences of length W) score at least the threshold constant T when aligned using a substitution matrix. Formally, BLAST identifies all hits between the query sequence q at position i and the current collection sequence s at position j such that

$$\sum_{0 \leq n < W} s(q_{i+n}, s_{j+n}) \geq T$$

where $s(q_i, s_j)$ is the score resulting from aligning the ith amino acid in q and the jth amino acid in s.

Hits are identified with the aid of a lookup table that is constructed from the query sequence and a substitution matrix. Given an alphabet of size a and a word length W the table contains a^W entries, where each entry represents a possible word $w = w_1 ... w_W$. Each entry specifies the location of words in the query that match w, that is, the entry contains a

list of query positions $I = i_1, ..., i_{|I|}$ such that for each position i,

$$\sum_{0 \leq n < W} s(q_{i+n}, w_n) \geq T$$

The lookup table is illustrated in Figure 3.3 on page 58.

During search, overlapping substrings or words of length W are extracted from the current collection sequence. Each word is used to lookup an entry in the table, which specifies the offset into the query, or *query position*, of zero or more hits. The query and collection sequence offsets $[i, j]$ of each hit are then passed on to subsequent stages of the BLAST algorithm.

The word length and threshold parameters, W and T, have considerable effect on the sensitivity of BLAST and amount of computation involved in the hit detection process and subsequent stages. In this section, we analyse the effect of varying parameter choices and illustrate that a longer word length can be used to achieve comparable accuracy with less computation. We then describe in detail the codeword lookup table used by NCBI-BLAST to identify hits efficiently, and show that the main drawback with large word lengths is an increase in the size of this data structure.

5.1.1 Varying the word size and threshold

The threshold parameter T that defines a high-scoring match between words affects the speed and accuracy of BLAST. Consider the example mutation data matrix and query sequence shown in Figure 3.3 on page 58 and a setting of $T = 7$. Observing the matrix, a match between the word `BA` and `BC` scores 8, since the intersection of the row and column labelled `B` is a score of 6 and between `A` and `C` is 2 in the example scoring matrix shown. Since the threshold is $T = 7$, a match between `BA` and `BC` is a hit, and so occurrences of `BA` or `BC` in the query match occurrences of `BA` or `BC` in the collection sequence. High values of T restrict the number of non-identical *neighbourhood words* that match a word with a score below the threshold T, resulting in less sensitive but faster search. Low values of T expand the number of neighbourhood words, with the opposite effects. Careful choice of T, for each W and scoring matrix pair, is crucial to BLAST performance.

NCBI-BLAST uses default values of $W = 3$ and $T = 11$ for protein search [Altschul et al., 1997]. To investigate the effect of varying these two parameters we conducted the following experiment: for each value of $W = 2, 3, 4, 5$, we identified a value of T that provides a similar degree of accuracy to the defaults of $W = 3$ and $T = 11$. We then used our own implementation of BLAST to perform 100 searches for each parameter pair between queries selected

Settings	Total Hits	Total Extensions		ROC_{50}
		(Ungapped)	(Gapped)	score
W = 2, T = 10	812,500,081	80,571,433	53,063	0.382
W = 3, T = 11	428,902,038	17,785,578	47,264	0.380
W = 4, T = 13	219,446,068	6,113,409	43,682	0.380
W = 5, T = 15	111,574,045	3,670,768	40,116	0.378

Table 5.1: A comparison of BLAST *search statistics for pairs of W and T that result in similar accuracy. The average number of hits, ungapped extensions, and gapped extensions and SCOP test ROC_{50} scores are shown.*

randomly from the GenBank non-redundant protein database and the entire database. We recorded the average number of hits, ungapped extensions, and gapped extensions for each parameter pair. The results of this experiment are shown in Table 5.1.

Accuracy was measured using the SCOP test that is discussed in Section 3.3.3. For the test, we used version 1.65 of the ASTRAL Compendium for Sequence and Structure Analysis [Chandonia et al., 2004]. The test provides a *Receiver Operating Characteristic* (ROC) score between 0 and 1, where a higher score reflects better sensitivity and selectivity. The version of the GenBank database used throughout this chapter was downloaded 17 November 2004 and contains 2,163,936 sequences in around 712 megabytes of sequence data. Our version of BLAST that was used for testing is FSA-BLAST that uses the semi-gapped and gapped alignment algorithms described in the previous chapter.

Table 5.1 shows that a longer word length W can be used to achieve comparable accuracy with far less computation. For example, the parameter settings $W = 5$ and $T = 15$ achieve similar accuracy to $W = 2$ and $T = 10$, while generating 86% fewer hits, 95% fewer ungapped extensions, and 14% fewer gapped extensions. We show in Section 5.3, however, that the reduction in computation for long word lengths does not necessarily translate to faster runtimes. This is because the data structures required for lookups with long word lengths are much larger and cannot be maintained in *CPU cache*.

CPU caches are typically no more than one megabyte in size, and store data that has been recently accessed (temporarily local) or is located near recently-accessed data (spatially local). CPU cache performance, design, and characteristics vary across platforms and architectures. However, in general, for memory-based tasks, data structures that are smaller and cluster related data make better use of cache and are faster on most architectures. Such data

Collection sequence:	TQACIV
Words:	TQA
	QAC
	ACI
	CIV

Figure 5.1: Words extracted from a collection sequence during search.

structures and algorithms are referred to as being *cache conscious*.

Next, we describe the codeword lookup table used by NCBI-BLAST to perform hit detection. We show that the structure is too large to fit into the CPU cache of a modern workstation for word sizes of $W = 4$ or greater. In Section 5.2 we present our cache-conscious data structure for the first stage of BLAST. Our aim in developing this new structure is to improve caching, resulting in longer word lengths and faster runtimes.

5.1.2 NCBI-BLAST Codeword Lookup

We have shown a schematic of the lookup table used by BLAST in Figure 3.3 on page 58. This section describes the implementation in more detail.

In NCBI-BLAST, each collection to be searched is pre-processed once using the *formatdb* tool and each amino-acid coded as a 5-bit binary value. The representation is 5 bits because there are 24 symbols — 20 amino-acids and 4 IUPAC-IUBMB amino-acid ambiguity character substitutions — and $24 < 2^5$. For fast table lookup, a codeword is constructed by reading and concatenating W 5-bit values together. The codeword is used as an offset into the codeword lookup table; this provides unambiguous, perfect hashing, that is, a guaranteed $O(1)$ lookup for matching query positions to each collection sequence word. The table contains space for $a \times 32^{(W-1)}$ slots, for an alphabet of size a and word length W, however only a^W of these correspond to valid codewords and are actually used. For $a = 24$ symbols and $W = 3$, a word is represented by 15 bits and the table contains space for $a \times 32^2 = 24,576$ slots; $24^3 = 13,824$ of these correspond to valid codewords and are actually used, whilst the remaining 10,752 slots represent unused space inside the lookup table.

During search, collection sequences are read code-by-code, that is, 5-bit binary values between 0 and 23 decimal inclusive are read into main-memory. Since codewords overlap, each codeword shares $W - 1$ symbols with the previous codeword. For example, when $W = 3$ each codeword shares two symbols with the previous codeword as shown in Figure 5.1. To construct the current codeword, 5 bits are read from the sequence, and binary mask (&),

bit-shift (<<), and OR (|) operations applied. These three binary operations are used to remove a residue from the start of the word, promote each of the remaining residues to its new position, and add a new residue to the end of the word. Consider the example collection sequence in Figure 5.1 that is processed using a word length of $W = 3$. After reading codes for the first three letters `T`, `Q`, and `A`, the current codeword contains a 15-bit representation of the first three-letter word `TQA`. The code for `C` is read next, and the codeword needs to be updated to represent the second word `QAC`. To achieve this, three operations are performed: first, a binary masking operation is used to remove the first 5 bits from the codeword, resulting in a 10-bit binary representation of `QA`; second, a left binary shift of 5 places is performed; and, last, a binary OR operation is used to insert the code for `C` at the end of the codeword, resulting in a binary representation of the new codeword `QAC`. The new codeword is then used to lookup an entry in the table.

Each slot in the lookup table specifies the location of zero, one, or more query positions. A query position is the offset into the query where a matching word occurs. We describe the design of the lookup table employed by NCBI-BLAST now. Each slot in the lookup table contains four integers. The first integer specifies the number of hits in the query sequence for the codeword, and the remaining three integers specify the query positions associated with up to three of these hits. In cases where there are more than three hits, the first query position is stored in the slot and the remaining query positions are stored outside the table, with their location recorded in the slot in place of the second query position. Figure 5.2 illustrates the design of the lookup table: the example shows a fraction of the table including two codewords that have two and zero query positions, and one codeword that uses the external structure to store a total of five query positions.

Storing up to three query offsets in each slot in the lookup table improves spatial locality and, therefore, takes advantage of CPU caching effects; a codeword with less than three hits can be processed without jumping to an external main-memory address. Assuming slots are within cache, the query positions can be retrieved without accessing main-memory. However, storing query positions in the table increases the table size: as the size increases, fewer slots are stored in the CPU cache.

In addition to the primary lookup table, NCBI-BLAST uses a secondary table to reduce search times. For each word, the secondary table contains one bit that indicates whether or not any hits exist for that word. If no hits exist, the search advances immediately to the next word without the need to search the larger primary table. This potentially results in faster search times because the smaller secondary table is compact (it stores a^W bits compared to

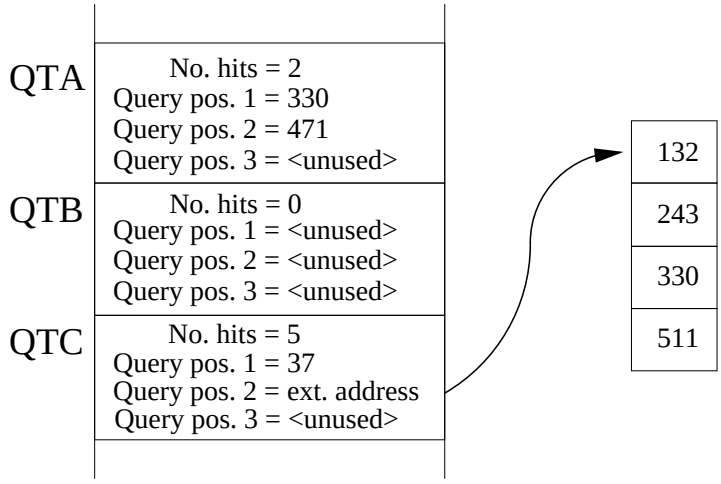

Figure 5.2: Three slots from the lookup table structure used by NCBI-BLAST. *In this example,* $W = 3$ *and the words* `QTA`, `QTB`, *and* `QTC` *have two, zero, and five associated query positions respectively. Since* `QTC` *has more than three query positions, the first is stored in the table and the remaining positions are stored at an external address.*

$a^W \times 4$ 32-bit integers) and can be accessed faster than the larger primary table.

The average size of the NCBI-BLAST codeword lookup table for values of W and T that were used previously in this chapter are shown in the column labelled *codeword lookup* of Table 5.2. For word sizes of $W = 2$ and $W = 3$ the primary table is less than half a megabyte and small enough to reside in the CPU cache of most modern workstations. However, as W increases, so does the table size: when $W = 4$, the table is around 12 Mb in size, much larger than the available cache on most modern hardware. This is probably why word sizes of $W = 4$ or larger are disabled by default in NCBI-BLAST. Further, we show in Section 5.3 that a word size of $W = 4$ provides slower searches than the default of $W = 3$ despite a reduction in computation as shown in Section 5.1.1. The results in Table 5.2 also show that the secondary table is considerably smaller than the primary table across the range of word lengths, so that codewords that do not generate a hit are less likely to result in a cache miss.

5.1.3 Summary

The first stage of BLAST involves identifying sort matching regions, also known as hits, between the query and sequences from the collection. A lookup structure that is constructed from the query sequence and scoring matrix is used to identify hits quickly. During search,

BLAST parameters		Codeword lookup		Deterministic finite
		Primary	Secondary	automaton
W = 2	T = 10	13 Kb	$\ll$ 1 Kb	2 Kb
W = 3	T = 11	392 Kb	3 Kb	22 Kb
W = 4	T = 13	12,329 Kb	96 Kb	257 Kb
W = 5	T = 15	393,374 Kb	3,072 Kb	3,011 Kb

Table 5.2: Average size of the codeword lookup and DFA structures for 100 queries randomly selected from the GenBank non-redundant protein database. Values of W and T with comparable accuracy were used. Experiments were conducted using our own implementation of the DFA structure and the codeword lookup structure used by NCBI-BLAST *version 2.2.10 with minor modifications to allow word sizes 4 and greater.*

words are extracted from the collection for which the lookup structure specifies the location of matching words in the query. In this section, we have described this process for detecting hits in detail. We have described the codeword lookup approach used by NCBI-BLAST and investigated the effect of varying the two key parameters in this process: the word length and neighbourhood threshold.

In the next section, we describe an alternative data structure — a deterministic finite automaton — for fast hit detection. We carefully optimise our new automaton to minimise its size and improve search performance.

5.2 Deterministic finite automaton

The original version of NCBI-BLAST [Altschul et al., 1990] used a deterministic finite automaton (DFA) [Sudkamp, 1997] similar to the one we describe next, but it was abandoned in 1997 for the lookup table approach described in the previous section. In this section, we propose a new cache-conscious DFA design for fast codeword lookup. We first describe the original DFA structure used by BLAST, and then describe our new cache-conscious DFA design and several optimisations that can be applied to the new structure. Results for our new DFA are presented in Section 5.3.

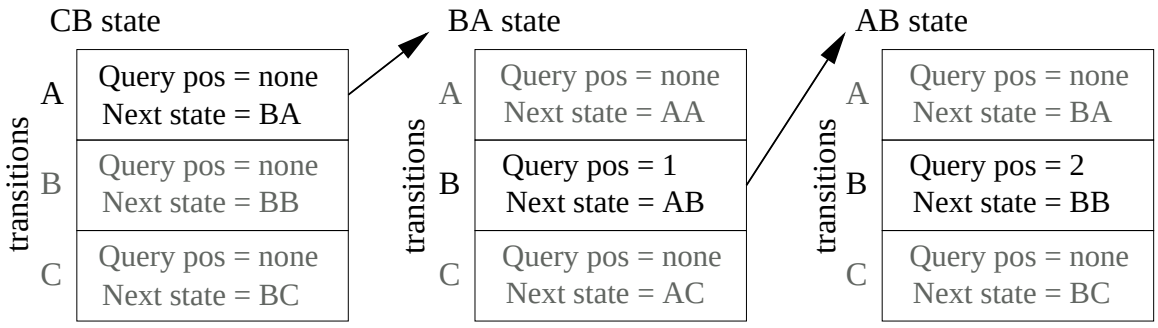

Figure 5.3: The original DFA structure used by BLAST. *In this example* $a = 3$ *and the query sequence is* `BABBC`. *The three states shown are those visited when the collection sequence* `CBABB` *is processed.*

5.2.1 Original automaton

The DFA approach is fundamentally different to the lookup table scheme. Rather than generating a unique numeric codeword for each word, the automaton represents the query as *states* and *transitions* between those states. The DFA employed by the original BLAST is as follows: each possible prefix of length $W - 1$ of a word is represented as a state, and each state has transitions to a possible next states. Associated with each transition is a list of zero or more query positions.

Figure 5.3 shows a portion of a DFA that has been constructed using a simplified alphabet with size $a = 3$, a word length of $W = 3$ and an example query sequence `BABBC`. In this example we assume that only identical words score above the threshold T, that is, we do not consider neighbourhood words. Three states with a total of nine transitions are shown in the figure. The `B` transition from the `BA` state contains the single query position $i = 1$ because the word `BAB` appears in the query beginning at the first character. Similarly, the `B` transition from the `AB` state contains the single query position $i = 2$ because the word `ABB` appears in the query beginning at the second character.

The structure is used as follows. Suppose the collection sequence `CBABB` is processed. After the first two symbols `C` and `B` are read, the current state is `CB`. Next, the symbol `A` is read and the transition `A` from state `CB` is followed. The transition does not contains any query positions and the search advances to the `BA` state. The next symbol read is `B` and the `B` transition is followed from state `BA` to state `AB`. The transition contains the query position $q = 1$, which is used to record a hit at query and collection sequence offsets $[1, 2]$. Finally, the symbol `B` is read and this produces a single hit at offsets $[2, 3]$ because the `B` transition out

of state `AB` contains the query position $i = 2$ that matches the collection sequence at offset $j = 3$. The automaton contains a state for every possible prefix of length $W - 1$, regardless of whether any words with that prefix appear in the query or not.

The structure we have described is that used in the original version of BLAST. The original DFA implementation used a linked list to store query positions and stored redundant information[1], making it unnecessarily complex and not cache-conscious. This is not surprising: in 1990, general-purpose workstations did not have onboard caches and genomic collections were considerably smaller, making the design suitable for its time.

However, despite the shortcomings of the original DFA implementation, the automaton structure has several important advantages. First, the DFA is more compact. The lookup table structure has unused slots, since eight of the 5-bit codes are unused. The table contains space for $a \times 32^{(W-1)}$ entries however only a^W of these correspond to valid codewords and are actually used. For the default word length of $W = 3$ and an alphabet size of $a = 24$, this means that 10,752 out of 24,576 slots or 44% of the overall structure is unused. The DFA does not suffer from this problem because the scheme does not rely on codewords. Further, a range of optimisations can be applied to the DFA which we discuss in Section 5.2.3.

5.2.2 New automaton

A drawback of the original DFA is that is requires an additional pointer for each word to the next state, resulting in a significant increase in the size of the lookup structure. To examine this effect, we experimented with 100 query sequences randomly selected from the GenBank database and found that pointers consume on average 61% of the total size of the structure when default parameters are used and the relevant optimisations from Section 5.2.3 are applied to the original DFA. However, it is possible to reduce the number of pointers as we discuss next.

The next state in the automaton is dependent only on the suffix of length $W - 1$ of the current word and not the entire word. We can therefore optimise the structure further: we let each state correspond to the suffix of length $W - 2$ of the current word, and each transition correspond to the suffix of length $W - 1$. Each transition has two pointers: one to the next state, and one to a collection of words that share a common prefix of length $W - 1$. The words are represented by entries that each contain a reference to a list of query positions. When a symbol is read, both pointers are followed: one to locate the query positions for the

[1] Based on our own analysis of NCBI-BLAST 1.1

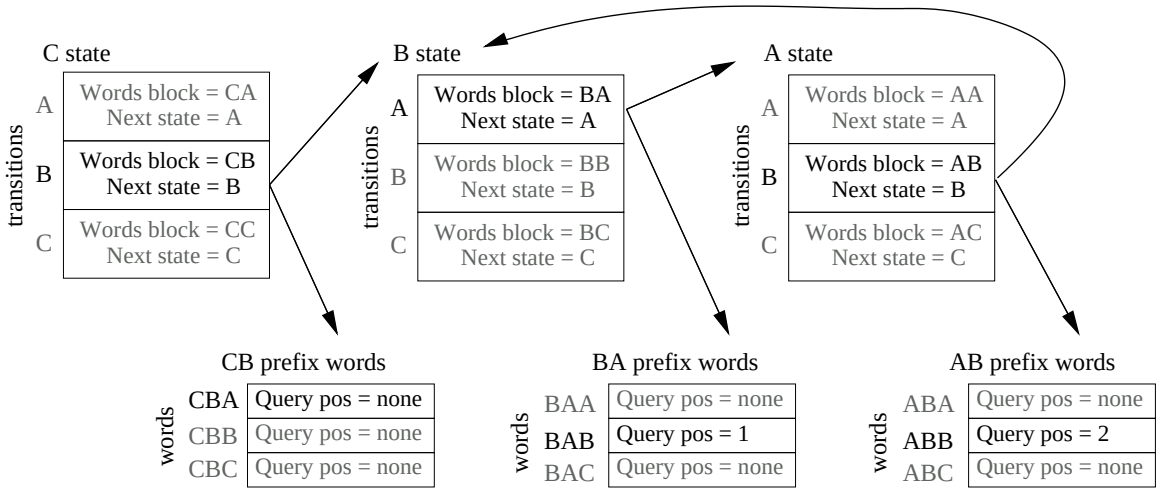

Figure 5.4: The new DFA structure. In this example $a = 3$ and the query sequence is `BABBC`*. The figure shows the portion of the new structure that is traversed when the example collection sequence* `CBABB` *is processed.*

new word and the other to locate the next state in the structure.

This new DFA is illustrated in Figure 5.4. Again, we use the example query sequence `BABBC` and the diagram only shows the portion of the structure that is traversed when the collection sequence `CBABB` is processed. Consider now the processing of the collection sequence. After the first symbol `C` is read, the current state is `C`. Next, the symbol `B` is read and the `B` transition is considered. It provides two pointers: the first to the new current state, `B`, and the second to the `CB` prefix words. The next symbol read is `A` and two events occur: first, the word `CBA` from the `CB` prefix words is checked to obtain the query positions for that word, of which there are none in this example; and, second, the current state advances to `A`. Next, the symbol `B` is read, the word `BAB` in the collection of `BA` prefix words is accessed and a single hit is recorded with query position $i = 1$. The current state then advances to `B`. Finally, the symbol `B` is read, the word `ABB` of the `AB` prefix words is consulted and a single hit is recorded with query position $i = 2$.

This new DFA structure requires more computation to traverse than the original: as each symbol is read, two pointers are followed and two array lookups are performed. In contrast, the original DFA structure requires one pointer to be followed and one array lookup to be performed. However, the new DFA is considerably smaller and more cache-conscious, since the structure contains significantly fewer pointers. Despite each state containing two

pointers, there are $a^{(W-2)}$ instead of $a^{(W-1)}$ states. Further, this rearrangement affords additional optimisations that we describe next.

5.2.3 Optimising the automaton

In this section, we present several optimisations that can be applied to our novel DFA structure described in the previous section. Our optimisations are designed to reduce the overall size of the structure and increase internal locality with the aim of improving cache performance. Importantly, our strategies reduce the DFA size without any computational penalty.

Storing query positions

We have devised a new system for storing query positions in the automaton structure. Unlike the more rigid codeword lookup table, the location of each entry in the DFA is not fixed and we take advantage of this flexibility. Consider the new DFA structure illustrated in Figure 5.4. Each state has a transitions that must be grouped together into an array. Similarly, words that share a common prefix must be grouped together into an array with a entries. However, because these states and blocks are located by following pointers in the automaton, they may be placed anywhere in memory. This allows flexibility in their arrangement that is not possible when the location of each entry is governed by its associated codeword. Further, it permits query positions to be embedded in the DFA itself.

We take advantage of this in our DFA structure and we propose that query positions are stored outside of each entry, immediately preceding each block. That is, query positions are stored before each collection of words with a common prefix. This reduces the size of each entry — only one value must be recorded per word to indicate the location of the list of query positions — but minimises the likelihood of a cache miss when accessing that list because it is located nearby. We have implemented our approach as follows. Lists of query positions for every entry in the block are recorded before that block, and query position lists are terminated with a zero value. The distance between the first entry in the block and the start of the query positions list is recorded for each word, where a zero value indicates an empty list.

This new arrangement is illustrated on the left-hand side of Figure 5.5. In this example, the alphabet size is $a = 3$ and the block of words with a `AB` prefix is shown. The word `ABA` occurs at offset 16 in the query: the `ABA` entry contains a list offset of two indicating that the list of query positions for this word commences two positions before the start of the block,

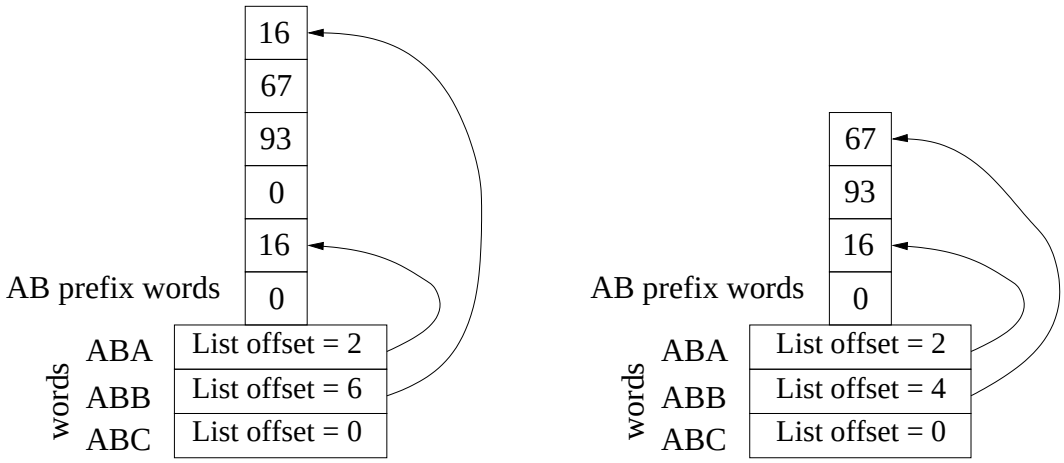

Figure 5.5: Example of query position reuse. The word `ABB` *produces a hit at query positions 16, 67, and 93, the word* `ABA` *produces a hit at query position 16 and the word* `ABC` *does not produce any hits. A simple arrangement is shown on the left. A more space efficient arrangement that reuses query positions is shown on the right.*

where the zero-terminated list {16} appears. The word `ABB` occurs at offsets 16, 67, and 93 in the query: the list offset of six points to the list of query positions {16, 67, 93}. Finally, the word `ABC` does not appear in the query, which is indicated by the list offset value of zero.

As part of this new structure, we have developed a technique for reusing query positions between words with a common prefix. An existing list of query positions $M = m_1, ..., m_{|M|}$ can be reused to store a second list $N = n_1, ..., n_{|N|}$ in memory if $|N| \leq |M|$ and N is the suffix of M, that is, $M_{|M|-i} = N_{|N|-i}$ for all $0 \leq i \leq |N|$. Because the order of the query positions within an entry is unimportant, the lists can be rearranged to allow more reuse. We have employed a greedy algorithm that produces a near-optimal arrangement for minimising memory usage. The algorithm processes the new lists from shortest to longest, and considers reusing existing lists in order from longest to shortest. An example of query position reuse that results from our method is shown in Figure 5.5. A simple and less space efficient arrangement is shown on the left-hand side. On the right-hand side, the query positions are rearranged to permit reuse between the words `ABA` and `ABB`. The list {16, 67, 93} is reordered to {67, 93, 16} so that the last value in the list can be reused to produce the zero-terminated list {16}. Using this approach, the number of list items and zero-terminators to be stored for the block of words is reduced from six to four.

To measure the effectiveness of our list reuse scheme, we experimented with 100 query

sequences randomly selected from GenBank. We observed an average reduction in list size of 39% for our reuse scheme compared to a baseline where no query position reuse was performed. This confirms that the approach successfully reduces the overall size of the DFA.

Utilising background amino acid frequencies

We have also optimised the arrangement of the overall DFA structure to improve cache performance, by clustering more frequently accessed states together. This approach is possible using the automaton because words do not map to codewords, that is, to offsets within the structure, resulting in flexibility in the arrangement of states in the structure.

We have chosen to cluster together frequently-accessed states, maximising the chances that these states will be cached. In the DFA we propose, the most-frequently accessed states are clustered at the centre of the structure; to do this we use the Robinson and Robinson background amino-acid frequencies [Robinson and Robinson, 1991].

Similarly, entries within each prefix block can be arranged from most- to least-frequently occurring amino-acid residue, improving caching effects and minimising the size of the offset to the query positions for more commonly accessed states. Within each group of entries that share a $W-1$ residue prefix, entries with more common suffices are positioned at the beginning of the block, minimising the distance between more common entries and the associated list of query positions. We accomplish this in our DFA by assigning binary values to amino-acids in descending order of frequency.

Further optimisations

The new DFA also offers the ability to reuse a single block of entries for two or more different prefixes. If the lists of query positions is the same for each of the a suffix symbols for a pair of prefixes, then a single block of entries can be used for both. In this event, the *words block* field of the two transitions point to a single block of words. This is illustrated in Figure 5.6, using an example where a single block is used for both the prefixes `BB` and `CB`. In this example, the words `BBA`, `CBA`, `BBC`, and `CBC` do not produce any hits. The words `BBB` and `CBB` both produce a single hit at query position $i = 35$, indicating that both words produce a high-scoring match at this position in the query. Because the query position lists are the same for each suffix symbol for the `BB` and `CB` prefixes, a single block can be used for both. This reuse leads to a further reduction in the overall size of the DFA structure: in our experiments with 100 random queries described previously we found that the scheme on

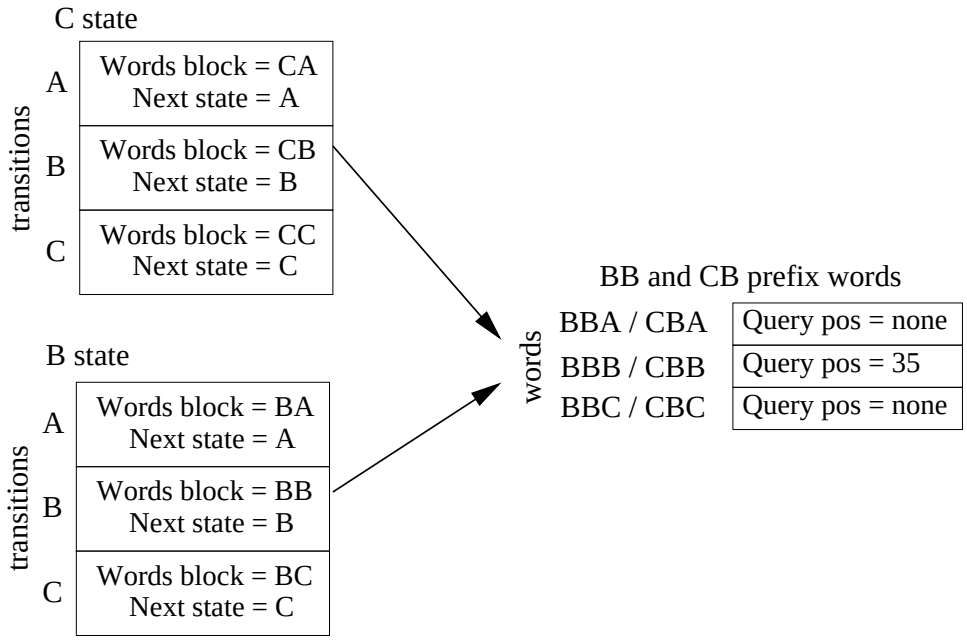

Figure 5.6: Illustration of block reuse in the DFA. A single block of entries, shown on the right, is used for both the `BB` *and* `CB` *prefixes.*

average reduces the number of blocks by 16%.

To further optimise the new structure for longer word lengths, we have reduced the alphabet size from $a = 24$ to $a = 20$ during construction of the structure by excluding the four ambiguity characters: `V`, `B`, `Z`, and `X`. We have observed that these four characters are highly infrequent, contributing a total of less than 0.1% of symbol occurrences in the GenBank non-redundant database. To do this, we replace each ambiguity character with an amino-acid symbol that is used only during the first stage of BLAST. This has a negligible effect on accuracy: Table 5.3 shows there is no perceivable change in ROC score for the SCOP test, despite a small change in total hits between the queries and collection sequences. The approach of replacing ambiguity characters with bases is already employed by BLAST for nucleotide searches, as originally proposed by Williams and Zobel [1997].

Our final optimisation is to store query positions as 16-bit integers where possible, instead of 32-bit integers as used in NCBI-BLAST. In the case where 32-bit integers are required — because the query exceeds 65,536 symbols in length — we use those instead.

The size of this optimised deterministic finite automaton for values of $W = 2, 3, 4, 5$ is shown in Table 5.2. The new structure is considerably smaller than the NCBI-BLAST codeword

	No ambiguity substitution		Ambiguity substitution	
	Total hits	ROC_{50}	Total hits	ROC_{50}
W = 2, T = 10	812,500,081	0.382	815,203,770	0.382
W = 3, T = 11	428,902,038	0.380	429,296,279	0.380
W = 4, T = 13	219,446,068	0.380	219,705,636	0.380
W = 5, T = 15	111,574,045	0.378	111,785,042	0.378

Table 5.3: Effect on search accuracy of substituting non-ambiguity characters for ambiguity characters during the first stage of BLAST *search. ROC_{50} scores were measured using the SCOP database and number of hits was measure by searching 100 randomly selected queries against the entire GenBank non-redundant database. Our* BLAST *implementation was used.*

table: for the default word length of $W = 3$ it is roughly 6% of the size of the codeword lookup table. The difference in size is even greater for larger word lengths: when $W = 4$ it is roughly 2% of the size of the lookup table and when $W = 5$ it is less than 1% of the size of the original data structure. Importantly, the new DFA is less than half a megabyte in size when a word length of $W = 4$ is used, making it small enough to fit into the available cache on most modern processors. In the next section, we show that this reduction in size leads to significantly faster search times due to better cache performance.

5.2.4 Summary

In this section, we described a new approach to hit detection in BLAST that employs a deterministic finite automaton (DFA) for fast word matching. We have presented a new automaton structure that requires more computation to traverse than the original, but contains fewer pointers and is significantly smaller. We have applied several optimisations to our new DFA to further reduce its size. As a result, our new data structure is roughly 6% of the size of the codeword lookup table when default parameters are employed.

In the next section, we evaluate our new automaton and compare the performance of this structure to the original codeword lookup approach used by NCBI-BLAST.

5.3 Results

This section presents the results of our experiments with various implementations of the first stage of BLAST.

We present a comparison of the NCBI-BLAST codeword lookup approach, our optimised implementation of the NCBI-BLAST approach, and an implementation of our optimised DFA scheme. All code was compiled using the same compiler flags as NCBI-BLAST. The NCBI-BLAST implementation is version 2.2.10 with a minor modification to permit experimentation with word lengths $W = 4$ and greater. All experiments used default SEG filtering [Wootton and Federhen, 1993] of the query sequences and NCBI-BLAST default parameters except where noted.

Collections and queries are as described in Section 5.1.1. The best elapsed time of three runs was recorded for each query, and then query times averaged across 100 queries. All experiments were carried out on machines under light load, with no other significant processes running. Four modern workstations were used in the experiments: an Intel Pentium 4 2.8 GHz with 16Kb L1 cache, 1Mb L2 cache and 2 Gb of RAM; an Intel Xeon 2.8GHz with 16Kb L1 cache, 512Kb L2 cache, 1Mb L3 cache and 2 Gb of RAM; an Apple PowerMac G5 dual processor 2.5GHz with 64Kb L1 cache, 512Kb L2 cache and 1.5 Gb of RAM; and, a Sun UltraSPARC-IIIi 1280 MHz with 96Kb L1 cache, 1Mb L2 cache and 1 Gb of RAM.

Table 5.4 shows a comparison of our implementation of the NCBI table-based approach to the original NCBI-BLAST version. With our optimisations — including minor changes to reduce the computation involved in constructing codewords and accessing the primary and secondary lookup tables — elapsed query times for stage one are typically around 70% to 80% of the NCBI-BLAST times. The exception is the PowerMac G5, where our optimisations result in an 8% speed up. We believe the different performance on the PowerMac is due to a relative difference in fundamental costs — of shifts, additions, binary OR, and increments — between it and the other platforms. We use our implementation as a baseline in the experiments reported in the remainder of this section.

Figure 5.7 shows a comparison of the optimised table-based and DFA schemes using our own implementation of BLAST. These results show overall BLAST search times for each of the W and T parameters pairs with similar accuracy, except for $W = 5$ and $T = 15$ which did not produce a runtime below 50 seconds on any of the architectures tested. The DFA is significantly faster: for the default $W = 3$ setting, it results in 10% faster average runtimes, and is around 15% faster on the commonly-used Intel platforms. Given that the hit detection process accounts for 37% of the average search time, this is equivalent to a 41% speedup in the first stage of BLAST. Importantly, because the compact DFA structure caches effectively, it is practical for $W = 4$, where the elapsed query times are almost identical to $W = 3$ and 35%–50% faster than the table-based scheme for the same setting; the DFA is therefore a

Machine	NCBI-BLAST	Optimised Codeword	
	(secs)	(secs)	(%)
Intel Pentium 4	10.92	8.63	79%
Intel Xeon	10.60	8.26	78%
Sun UltraSPARC	18.58	13.27	71%
PowerMac G5	9.56	8.75	92%

Table 5.4: A comparison of BLAST *stage one times between* NCBI-BLAST *and our own implementation of table-based codeword lookup hit detection, using default parameters of* $W = 3$ *and* $T = 11$*. The percentage of the* NCBI-BLAST *runtime is also shown.*

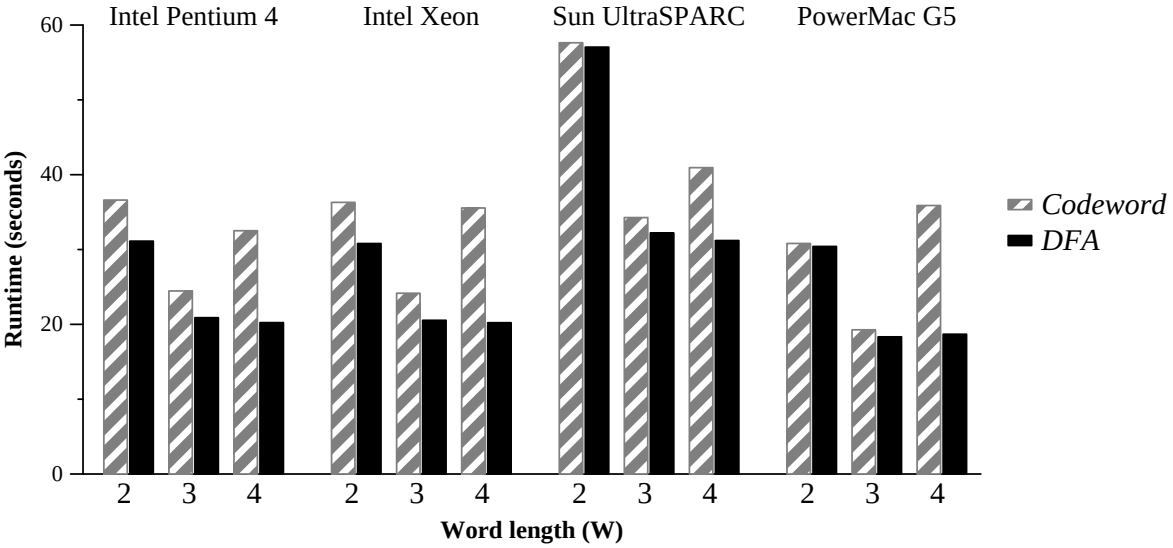

Figure 5.7: A comparison of BLAST *total search times for table-based codeword lookup and optimised deterministic finite automaton (DFA) designs. Experiments were conducted using* W *and* T *parameters pairs with similar accuracy on four different hardware architectures.*

practical structure for $W = 4$, a parameter disabled by default in NCBI-BLAST.

One of the main innovations in the 1997 BLAST paper [Altschul et al., 1997] was the introduction of a two hit mode of operation, where two hits on the same diagonal instead of only one hit are required to trigger an ungapped extension. Altschul et al. [1997] compared the sensitivity of these two different modes of operation towards high-scoring alignments, and concluded that the two hit mode is more sensitive and faster. However, their investigation only measured search sensitivity and not overall accuracy to homologous relationships and

considered only the default word length of $W = 3$.

We have conducted our own, more detailed investigation of these two different modes of operation across a range of word lengths using the SCOP database. Figure 5.8 shows a comparison between the one hit and two hit modes of operation using our new implementation of BLAST and our optimised DFA structure. The results show overall BLAST search times for values of W and T with comparable accuracy. For two hit mode we used the same values of W and T listed in Table 5.2. For one hit mode we used parameters $W = 3$, $T = 13$ that result in a ROC_{50} score of 0.384, and parameters $W = 4$, $T = 15$ that result in a ROC_{50} score of 0.383. Results for word lengths of $W = 2$ and $W = 5$ are not shown because they did not produce runtimes below 60 seconds for one hit mode of operation on any of the architectures tested.

The results confirm that BLAST is faster when run in two hit mode, in agreement with the 1997 BLAST paper [Altschul et al., 1997]. However, we note that for $W = 4$ the speed difference between one hit and two hit is significantly smaller than for $W = 3$. For the Intel platform in particular, the one hit mode of operation with a word length of $W = 4$ is roughly as fast and slightly more accurate than the default word length of $W = 3$ using the two hit mode.

Table 5.5 shows an overall comparison between NCBI-BLAST and FSA-BLAST: our own implementation of the BLAST algorithm. FSA-BLAST uses the new optimised DFA and our improvements to the gapped alignment stages of BLAST described in Chapter 4. Our implementation is around 30% faster on Intel and Sun platforms, and 20% faster on the Apple PowerMac G5. Around 15% of this speedup can be attributed to our improvements to the gapped alignment stages, while the remainder is due to the work described in this chapter. Importantly, there is no significant effect on accuracy: the ROC_{50} score for NCBI-BLAST is 0.379 compared to 0.380 for our implementation.

5.4 Conclusion

We have proposed, explained, and optimised structures for the first, hit detection phase in BLAST with the aim of improving overall BLAST runtimes for protein search. Hit detection matches words extracted from collection sequences against a structure derived from the query sequence, and BLAST uses an algorithmically different approach to hit detection for protein and nucleotide data. For protein searches, NCBI-BLAST uses a table-based lookup approach, a structure we have investigated and optimised for our experiments.

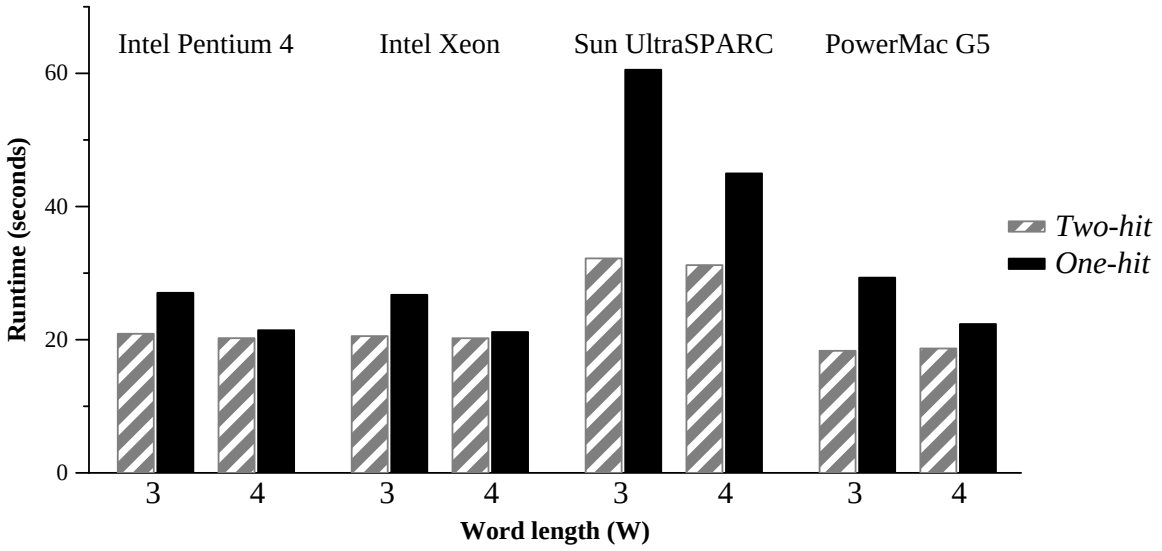

Figure 5.8: A comparison of BLAST *total search times for one hit and two hit modes of operation, using our implementation of* BLAST *and the optimised DFA structure. Experiments were conducted using W and T parameters pairs with similar accuracy on four different hardware architectures.*

	NCBI-BLAST	FSA-BLAST	
	(secs)	(secs)	(%)
Intel Pentium 4	30.58	20.89	68%
Intel Xeon	30.22	20.54	68%
Sun UltraSPARC-IIIi	46.42	32.21	69%
PowerMac G5	22.83	18.33	80%

Table 5.5: Runtime comparison between NCBI-BLAST *and* FSA-BLAST *using default parameters. The percentage of* NCBI-BLAST *runtime is also shown.*

We have explored varying word lengths and parameter settings in an attempt to optimise the first stage of search. Our results show that longer word lengths can result in comparable accuracy and a substantial reduction in computation. However, longer word lengths also result in larger lookup structures, which reduces the effectiveness of CPU cache and results in poor search performance in practice. It is therefore important to minimise the size of data structures used to perform hit detection.

We have proposed using a deterministic finite automaton (DFA) for fast, cache-conscious matching. Our scheme is optimised based on properties of the matching process and designed to make effective use of modern hardware. Our experiments show this approach works well: it typically improves overall BLAST search times by around 15% compared to our implementation of the NCBI-BLAST approach; our implementation of the table-based scheme is in turn around 20% faster than the NCBI implementation. Further, our scheme is practical for BLAST searches with a word length of four, a parameter value not supported by NCBI-BLAST.

We have also considered the effectiveness of the two hit mode of operation, where two hits on the same diagonal are required to trigger an ungapped extension. Our results confirm that the two hit mode offers better accuracy and search times than the original one hit approach, although the performance difference between the two schemes is smaller when the non-default word length of four is employed.

Our improvements to hit detection can also be applied to other variants of BLAST that use the same underlying protein comparison algorithm, including PSI-BLAST, BLASTX, TBLASTN and TBLASTX, and to other protein search tools that use word matches to trigger alignment.

We have integrated the deterministic finite automaton into our new implementation of BLAST. When combined with our improvements to the gapped alignment stages of BLAST that were described in the previous chapter, our methods are 20-30% faster than NCBI-BLAST for protein searches with no significant effect on accuracy.

In the next chapter, we present several new methods that enable faster nucleotide search by comparing collection sequences in their compressed form. Our new approaches to each stage of BLASTN more than halve overall nucleotide search times. We also present new approaches to hit detection for nucleotide data.

Chapter 6

Compressed Sequence Comparison

Amino-acid or protein search is preferred by biologists over nucleotide search: protein sequence collections are well-annotated, the collections are small and targeted, and rich tools and scoring schemes are available for the search process. Nucleotide search is less preferable than protein search, but often used: very large databases are available, many queries are from non-coding regions (and so do not have a protein equivalent), genome to genome comparisons are beginning to yield interesting results, and the queries return much broader result sets. As a result, nucleotide searches represent a sizeable proportion of the queries posed by biologists. Our analysis of usage data for the online BLAST service offered by the NCBI in Section 3.1.3 on page 72 reveals that around 57% of searches conducted by users were BLASTN searches between a nucleotide query and nucleotide collection. Further, we found that nucleotide searches represent a very significant investment of computing resources, equating to roughly 39% of the overall processing time involved in searches against the GenBank NR protein or nucleotide databases.

It is therefore not surprising that fast methods for nucleotide sequence comparison have received considerable attention at late. Methods that rely on spaced seeds, such as MEGABLAST [Zhang et al., 2000] and PATTERNHUNTER [Ma et al., 2002; Li et al., 2004], or construct a main-memory index of the collection, such as BLAT [Kent, 2002] and SSAHA [Ning et al., 2001], are specifically geared towards nucleotide search. Many of these methods are ideal for querying small collections, making them suitable for tasks such as whole-genome alignment, but fail to provide fast, sensitive search of large collections. Therefore, BLASTN remains the only practical tool for accurately searching any significant fraction of the GenBank database.

Surprisingly, little attention has been paid to the BLASTN, its innovations and its optimi-

sations, and the approach varies considerably from the description of the BLAST algorithm presented in the 1997 paper [Altschul et al., 1997]. As discuss in Section 3.1.3, BLAST uses an algorithmically different approach to amino acid and nucleotide comparisons. For BLASTN searches, matches between highly-similar, non-identical words are not considered during the first stage and only one hit, rather than two nearby hits on the same diagonal, is required to trigger an ungapped alignment. Further, a longer word length is commonly employed, and matches between DNA sequences with different orientations are considered. Importantly, BLAST records nucleotide sequences using the *byte packed* compression scheme, which we describe next and is the focus of this chapter. Indeed, the BLASTN and BLASTP algorithms are implemented separately, even though both tools are accessible through the BLASTALL application interface of NCBI-BLAST.

In this chapter, we propose innovations in BLASTN searching. Each of our schemes is based on the simple, practical byte packed compression scheme employed by BLAST to store nucleotide collection sequences that was described in Section 3.1.3. Using this approach, each of the four nucleotide bases is stored as a two-bit binary value, permitting four bases to be stored per byte; the ambiguity codes are stored in a separate structure. This compression scheme allows for very fast searching: compressed sequences are faster to read from disk than uncompressed sequences, they require less main-memory to store, and they can be compared without decompression. This latter point is important and unique to our work: our innovations allow a compressed query sequence to be compared to a compressed collection sequence in each of the first three stages of the BLAST algorithm. Specifically, our approach permits hit detection and ungapped hit extension without sequence decompression. We propose two techniques for performing gapped alignments on sequences without decompression: *bytepacked alignment* and *table-driven alignment*. Both techniques are used as a new stage before gapped alignment, with the result that only the final gapped alignment stages — which are performed for less than 1% of collection sequences — requires a decompression step.

Our work on the hit detection and ungapped hit extension stages focuses largely on re-engineering the BLASTN approach. We propose practical improvements to the NCBI BLASTN algorithms and data structures, and show that our compression based schemes improve search times significantly without affecting accuracy. For the first stage — where exact subsequences, typically of length $N = 11$, are identified between the query and each collection sequence — our approach reduces search times by almost 50%. For the second stage — where hits are extended using an ungapped alignment algorithm that does not permit in-

sertions or deletions — our approach reduces alignment times by around 43%. Together, since both stages consume around 90% of the entire search process, our schemes reduce total BLASTN search times by around 43%.

Our major innovations in this chapter are two novel approaches to gapped alignment. The first scheme — which we refer to as *bytepacked alignment* — allows the computation of alignment scores between a compressed query sequence and a compressed collection sequence. To do this, we compress the query sequence into four compressed representations, one for each possible position of a two-bit nucleotide code in a byte. We are then able to compute alignments between two sequences based on an ungapped extension from the previous stage, beginning at any offset without decompression. The bytepacked alignment method is heuristic because each collection sequence is compressed only once, requiring that insertions and deletions only occur at one in four offset positions. (However, insertions and deletions are permitted anywhere in the query because it is compressed into four representations.) Regular gapped alignment is then performed for high-scoring alignments using uncompressed sequences to compute the optimal alignment. Overall, with very little modification to the underlying BLASTN parameters, our bytepacked alignment scheme is around 78% faster than the NCBI BLASTN gapped alignment stage. Importantly, there is no significant difference in accuracy.

The second scheme — which we refer to as *table-driven alignment* — aligns sequences using a novel variant of the Four Russians [Wu et al., 1996] approach. This involves dividing the alignment matrix into blocks and using precomputed values from a lookup table to calculate alignment scores for each block. We use the Four Russians approach to calculate alignment scores for four adjacent characters, or a single byte, in the compressed sequence at a time. Using this approach, we are able to compare the query to sequences in the collection without decompression. When applied to BLAST, our table-driven alignment approach reduces the time taken to perform gapped alignment by 72%, and unlike bytepacked alignment, the approach is guaranteed to find the optimal alignment between two sequences.

Overall, our improvements to the hit detection, ungapped alignment, and gapped alignment stages more than double the speed of BLASTN with no significant effect on accuracy. Further, our improvements can be applied to other tools that use a two bits per base representation of nucleotide sequences.

This chapter is structured as follows. In Section 6.1, we describe our approaches for comparing compressed sequences. We present accuracy and performance results for our approaches in Section 6.2. Finally, we provide concluding remarks in Section 6.3. The

results and discussions presented in this chapter are based on Cameron and Williams [2006].

6.1 Novel bytepacked approaches

During nucleotide search NCBI-BLAST retrieves each sequence in the collection from disk, where it is stored using the bytepacked representation, and decompresses it partially or entirely to perform each of the four search stages. In this section, we describe new approaches to each stage of BLAST that permit sequences in the collection to be processed in their compressed form. We have investigated new methods for performing the first three stages of BLAST, including approaches to hit detection, ungapped alignment, and gapped alignment that can be applied to bytepacked collection sequences.

There are two major advantages in processing collection sequences in their compressed form. First, not every sequence in the collection needs to be decompressed, that is, converted from its on-disk bytepacked representation to a string of characters where each nucleotide base is represented by a single character. BLAST uses a filtering approach to search, where each stage takes longer to process a collection sequence than the previous but fewer sequences are processed in later stages. Our novel approaches to alignment in BLAST permit us to delay the decompression of sequences until later in the search process, and as a result fewer sequences need to be decompressed. Second, we are able to reduce the computation required to compare the query to collection sequences by processing the sequences in their compressed form. We present new methods for aligning collection sequences four bases at a time, effectively reducing sequence length by a factor of four and increasing the size of the sequence alphabet from 4 to 256. As a result, aligning bytepacked collections sequences has the potential to reduce search times by up to 75% compared to aligning the original sequences.

6.1.1 Stage 1: Hit detection

As described in Section 3.1.3 on page 55, BLASTN performs nucleotide hit detection by searching for occurrences of n matching characters between the query and collection sequences, then extending this initial match in each direction to determine if it forms part of an W-basepair *hit*. To identify matches quickly, BLASTN uses a lookup table to find byte-aligned exact matches of length n and then examines four bases from the collection sequence on each side of the match. This involves reading the compressed or *packed* bytes on each side of the initial match, unpacking them, and aligning the adjacent nucleotide bases with the respective query characters. If $W-n$ adjacent bases are found to match then a W-base hit has been identified;

the coordinates of the hit are then passed onto the second stage and an ungapped extension is performed.

We have optimised the hit detection process by developing a faster method for extending the initial hit of length n in each direction to check for a W-basepair hit. Rather than decompress bytes from the collection sequence that are adjacent to the initial match, our approach uses a pair of fast lookup tables and a special representation of the query to extend the hit without decompressing the collection sequence and in fewer operations.

Given a query sequence $q = q_1...q_{|q|}$, let us define a special byte packed representation of q, where each overlapping quadruplet from the query is packed into a byte, that is $Q = \{q_{[1:4]}, q_{[2:5]}, ..., s_{[|q|-3:|q|]}\}$. This representation can be used to extract a portion of the query starting at any offset in compressed form by selecting every fourth byte. For example, consider the query sequence `ACTTGACAGTAGGACC`. The special overlapping byte packed representation of this sequence is $Q = \{ACTT, CTTG, TTGA, TGAC, ..., GACC\}$. To extract a substring from the query of length 12 starting from the second character in compressed form, we extract the second, sixth, and tenth bytes from Q; this provides the string `CTTGACAGTAGG` in a byte packed representation.

We can use the special byte packed representation of the query to perform fast comparisons between four adjacent characters from the query sequence and four characters from a collection sequence. Given a collection sequence $s = s_1...s_{|s|}$, let us define $\overrightarrow{M}(q_{[i:i+3]}, s_{[j:j+3]})$ as the number of matching characters between the two bytes $q_{[i:i+3]}$ and $s_{[j:j+3]}$ going from the first character to the last before the first mismatch. Similarly, we define $\overleftarrow{M}(q_{[i:i+3]}, s_{[j:j+3]})$ as the number of matching characters between the bytes going from last character to the first before the first mismatch. For example, $\overrightarrow{M}(ACTT, ACAT) = 2$ and $\overleftarrow{M}(ACTT, ACAT) = 1$. Both functions can be computed quickly by performing a binary XOR operation between the pair of bytes; this provides a single value between 0 and 255 that specifies which character positions within the bytes differ. The result can then be used to fetch pre-computed values for $\overrightarrow{M}$ and $\overleftarrow{M}$ in a specially designed lookup table.

Given an initial hit of length n between bases $q_i \dots q_{i+n-1}$ from the query and bases $s_j \dots s_{j+n-1}$ from the collection sequence we use the new query representation Q and lookup tables for $\overrightarrow{M}$ and $\overleftarrow{M}$ to extend the hit. The new length of the hit N is calculated using the equation:

$$N = n + \overleftarrow{M}\left(q_{[i-4:i-1]}, s_{[j-4:j-1]}\right) + \overrightarrow{M}\left(q_{[i+n:i+n+3]}, s_{[j+n:j+n+3]}\right)$$

If $N \geq W$ then the location of the hit is passed on to the second stage where an ungapped extension is performed.

We have also optimised the codeword lookup table used to identify the initial n-character match. The NCBI-BLAST lookup table contains four 32-bit integer fields for each entry; one to store the number of hits, and the remaining to store up to three query positions in the table entry itself. If a word produces more than three hits then a list of query positions is stored outside of the table, and the table entry instead contains a pointer to the external list. As a result, if a word produces three hits or less then the list of query positions can be accessed without the cache penalty associated with jumping to an external address. This is the same table design used by BLAST for protein searches and is described in detail in Section 5.1.2.

The table design is reasonably efficient for the first stage of protein searches, where BLAST considers inexact but high-scoring matches as well as exact matches between words as hits. For protein data, our experiments in Section 3.1.3 revealed that BLAST identifies on average 229 hits per collection sequence and many words produce at least one hit. This contrasts with nucleotide searches where, as discussed in Section 3.1.3, BLAST identifies on average 3.5 hits per collection sequences and the lookup table is considerably more sparse. We conducted a simple experiment to illustrate this by constructing a BLAST word lookup table using default parameters for 100 nucleotide queries chosen randomly from the GenBank NR database. We found that on average 97.1% of words generated zero hits, 2.6% generated a single hit, and less than 0.3% generated more than one hit.

To address the differing characteristics of nucleotide searches, we have designed a new lookup table that is optimised for nucleotide search. The table records a single integer for each entry; a positive value provides the query position of a single hit, a negative value provides the location of an external, zero-terminated list of query positions, and a zero value indicates that there are no hits for that word. We use 16-bit integers when sufficient, that is, when the query is less than 32,768 bases in length, otherwise 32-bit integers are used. We have also found that the auxiliary lookup table described in Section 5.1.2 does not improve nucleotide search times and we do not employ the auxiliary table in our own implementation. We also considered employing our deterministic finite automaton scheme described in Chapter 5 to nucleotide data, however a preliminary experimental investigation revealed that the approach is unsuitable for processing compressed nucleotide sequences.

6.1.2 Stage 2: Ungapped alignment

As described in Section 3.1.3, NCBI-BLAST performs an ungapped extension in each direction from the initial hit until the score decreases by more than the dropoff parameter. During the extension process, BLAST keeps track of the best alignment score observed so far. To perform ungapped extensions on compressed nucleotide sequences, each byte from the collection sequence is unpacked as required, and the sequence is aligned one basepair at a time. Figure 6.1 provides a pseudo-code description of the ungapped extension algorithm used by NCBI-BLAST for aligning uncompressed collection sequences. The algorithm presented here is similar to the protein alignment routine presented in Figure 3.5 on page 62; the most significant difference is that sentinal codes are employed during protein sequence alignment to automatically terminate the extension process at either end of the sequences. Starting at the location of the hit that triggered the extension $[i, j]$, pairs of bases from the query and collection sequences are progressively aligned. The variable *bestscore* records the best alignment score so far, and the alignment process terminates at the end of either sequence or if the alignment score decreases by more than *dropoff*. Note that the psuedo-code given here performs the forward extension of an alignment only; some minor variations are required to perform the backwards extension of a hit.

We have developed a new ungapped alignment algorithm for BLASTN that permits alignment of compressed collection sequences. Using our approach, ungapped alignment is performed four bases at a time. The pseudo-code description of our new algorithm is shown in Figure 6.2. In addition to the $\overrightarrow{M}$ and $\overleftarrow{M}$ lookup tables described previously, the algorithm uses a third table to compute $\overline{\overline{M}}(q_{[i:i+3]}, s_{[j:j+3]})$, which we define as the total score for matching bases between a pair of bytes. For example, $\overline{\overline{M}}(ACTT, ACAT) = 2$ if a match score of 1 and mismatch score of -1 is used. The function $\overline{\overline{M}}$ is also calculated by performing a binary XOR between a pair of bytes and consulting a specially designed lookup table with 256 entries for pre-computed values of $\overline{\overline{M}}$.

The new algorithm aligns collection sequences one byte at a time using the $\overline{\overline{M}}$ lookup table. The algorithm also considers the alignment of individual bases at the start and end of the alignment. The partial alignment of the next byte is considered during the extension process when this may increase the optimal alignment score, that is, when $score > bestscore - (3 \times matchscore)$. This partial alignment is considered by consulting the $\overrightarrow{M}$ lookup table, and the variable $finescore$ records the score resulting from the partial alignment of the next byte. Again, some minor variations are required to perform the backwards extension including the

```
/* Input: q, s, i, j, dropoff */

UngappedExtension
int score = 0
int bestscore = 0
int i_best = 0, j_best = 0

while i ≤ |q| and j ≤ |s|
    if q_i = s_j then
        score ⟵ score + matchscore
    else
        score ⟵ score − mismatchscore
    if score > bestscore then
        bestscore ⟵ score
        i_best ⟵ i
        j_best ⟵ j
    else if bestscore − score > dropoff then
        stop
    increment i
    increment j

/* Output: i_best, j_best, bestscore */
```

Figure 6.1: Original ungapped extension algorithm used by NCBI-BLAST *for aligning nucleotide sequences.*

/* Input: q, s, i, j, dropoff */

BytePackedUngappedExtension
int score = 0
int bestscore = 0
int $\mathrm{i}_{best} = 0$, $\mathrm{j}_{best} = 0$

while i $\leq |q|$ **and** j $\leq |s|$
    if score > bestscore − (3 × matchscore) **then**
        finescore ⟵ score + $\overrightarrow{M}(\mathrm{q}_{[i:i+3]}, \mathrm{s}_{[j:j+3]})$
        if finescore > bestscore **then**
            bestscore ⟵ finescore
            i_{best} ⟵ i
            j_{best} ⟵ j
    else if bestscore − score > dropoff **then**
        stop
    score ⟵ score + $\overline{\overrightarrow{M}}(\mathrm{q}_{[i:i+3]}, \mathrm{s}_{[j:j+3]})$
    increment i by 4
    increment j by 4

/* Output: i_{best}, j_{best}, bestscore */

Figure 6.2: Improved ungapped extension algorithm for aligning compressed collection sequences four bases at a time.

use of the $\overleftarrow{M}$ instead of the $\overrightarrow{M}$ lookup function.

Our approach is similar to the ungapped alignment method used by SENSEI [States and Agarwal, 1996], which also aligns sequences one byte at a time with the aid of a lookup table. However, their approach only considers alignments that start and end on the byte boundary and so may result in suboptimal alignments. Unreported experiments found that the SENSEI approach leads to a loss in search performance or accuracy when used to perform the second stage of BLAST.

We have employed another minor optimisation to the ungapped extension process that reduces the number of bytes processed. The region covered by the initial match detected in Stage 1 is already known to contain $\frac{n}{4}$ matching bytes and we avoid realigning this region when performing an ungapped alignment.

6.1.3 Stage 3: Gapped alignment

Computing optimal gapped alignments between a query sequence and compressed collection sequences is significantly more complicated than computing ungapped alignments; there are several ways to align a packed byte from the collection with the query sequence when gaps are also considered. For example, optimal gapped alignments may include insertions or deletions in the middle of the packed byte. To illustrate this, consider the example gapped alignment between query sequence `ATGCAGTT` and collection sequence `ATGGTT` at the top-left of Figure 6.3. The first four characters of the collection sequence, `ATGG`, are represented by a single packed byte and the optimal alignment involves two insertions in the middle of the byte. A scheme that aligns sequences one byte at a time without considering insertions in the middle of a byte does not produce this optimal alignment. We discuss Figure 6.3 in more detail later in this section.

In this section, we propose two new approaches to gapped alignment that address this problem, *bytepacked alignment* and *table-driven alignment*. Bytepacked alignment restricts the location of gaps in an alignment so that they only occur on the collection sequence byte boundary. As a result, collection sequences can be aligned a byte at a time, however the approach generates suboptimal alignments. Table-driven alignment considers all possible alignments between a single base from the query sequence and a packed byte from the collection sequence through the use of a specially designed lookup table. Although table-driven alignment is lossless, our approach is only suitable for aligning sequences using non-affine gap costs. As we show later, both techniques provide a good approximation of the gapped

alignment score and work well when employed as a filtering step between the ungapped and gapped alignment stages of BLAST.

The advantages of our new approaches are two-fold. First, both techniques process the collection sequence one byte at a time, rather than one base at a time, providing a significant reduction in processing when compared to gapped alignment. Second, collection sequences do not need to be decompressed to perform bytepacked or table-driven alignment. Because both techniques provide an additional filtering step between ungapped alignment and gapped alignment, fewer collection sequences need to be decompressed before being realigned using gapped alignment. We consider each in turn.

Bytepacked alignment

In this section we propose our novel bytepacked alignment scheme. With restrictions, it enables bytepacked alignment to be performed on compressed collection sequences. Specifically, gaps can only start and end on the collection sequence byte boundary, that is where $j \equiv 0, \text{modulo}\, 4$, and as a result bytepacked alignment provides an approximation of the optimal gapped alignment score.

Bytepacked alignment is performed in a similar manner to gapped alignment. The key difference is that bytepacked alignment uses one row in the dynamic programming matrix for each packed byte in the collection sequence, rather than one row for each individual base. By restricting the start and end of gaps to lie on the byte boundary, it is possible to consider four bases at a time during alignment, as illustrated in Figure 6.4. The y-axis of the figure represents the sequence in its compressed form and the query is represented as a series of overlapping quadruplets along the x-axis; this is the same representation of the query, Q, described previously. Three events are considered for each cell in the matrix; a match of four bases — or a single byte — from the query sequence and collection sequence, a single insertion in the collection sequence, or a series of four insertions in the query sequence.

Figure 6.3 illustrates bytepacked alignments for two example sequence pairs, with the optimal gapped alignments and corresponding bytepacked alignments shown at the top and bottom of the figure respectively. The example on the left-hand side illustrates the case where the alignment contains insertions in the collection sequence. In Figure 6.3 (a) the optimal gapped alignment contains two insertions in the collection sequence that move the alignment from one diagonal to another. Bytepacked alignment still permits the insertions, however they must occur on the sequence byte boundary as illustrated in Figure 6.3 (b). The example

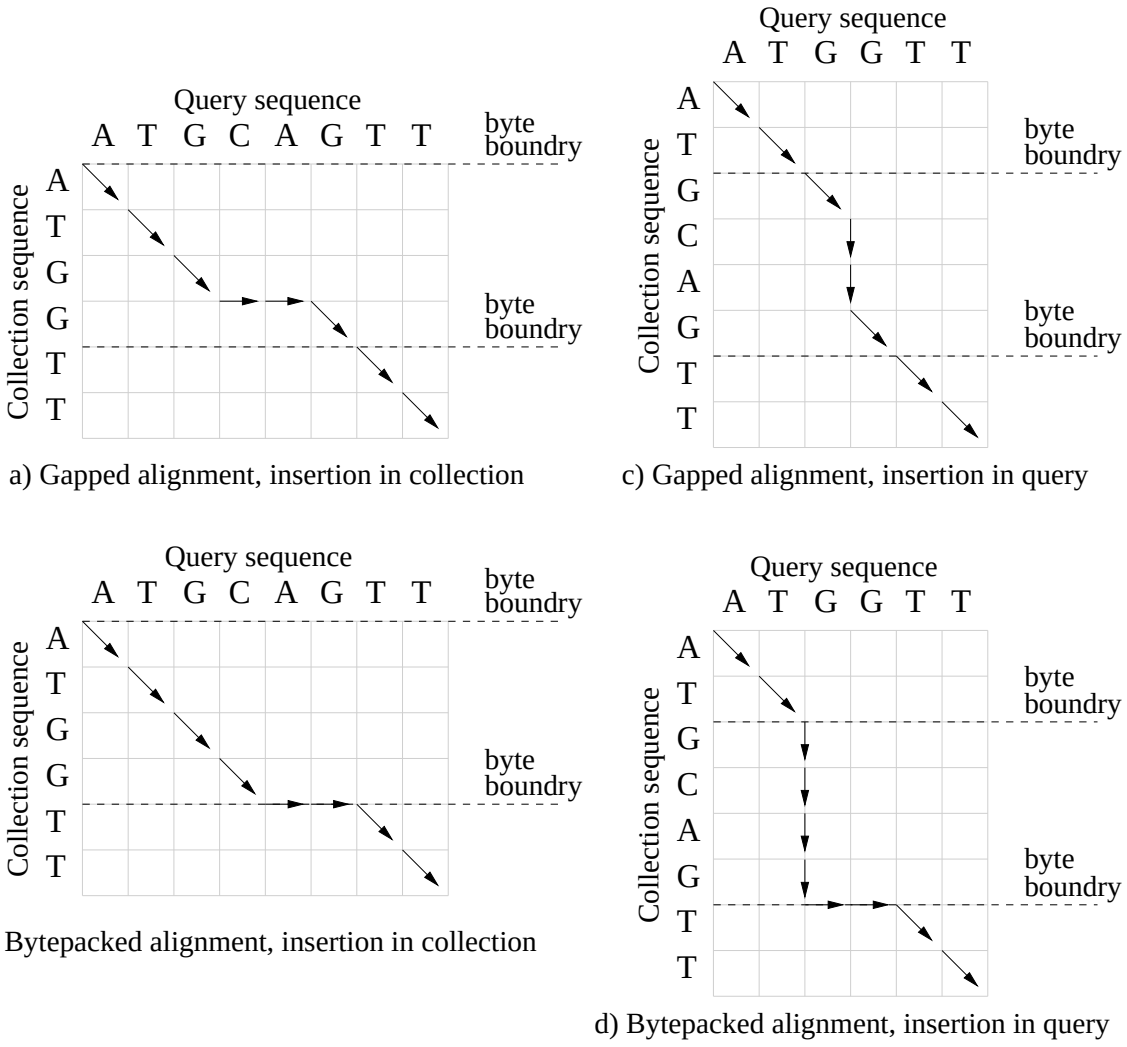

Figure 6.3: Effect of the bytepacked alignment constraints on an pair of example alignments. a) illustrates a gapped alignment containing insertions in the collection sequence and b) shows the equivalent bytepacked alignment with the insertions shifted to the collection sequence byte boundary c) illustrates a gapped alignment containing insertions in the query and d) shows the equivalent bytepacked alignment that uses adjacent gaps to change diagonal.

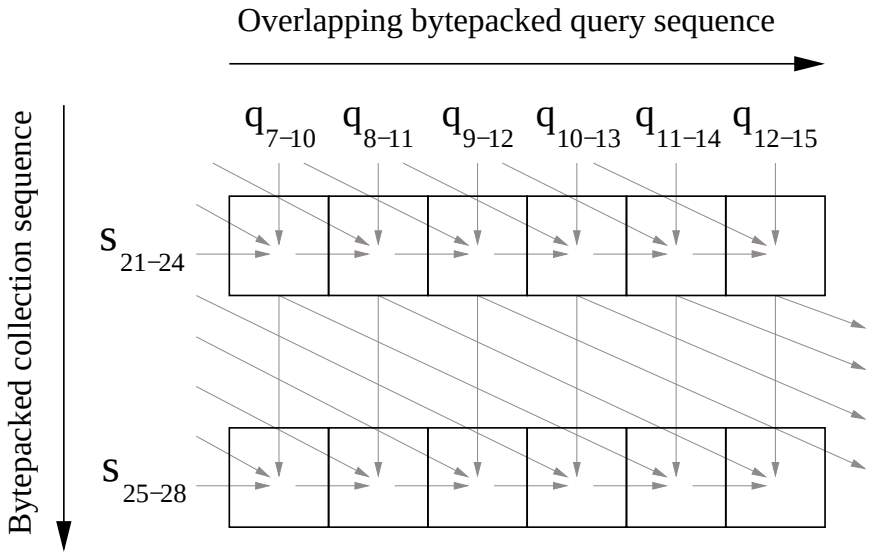

Figure 6.4: Portion of the dynamic programming matrix used to perform bytepacked alignment. A diagonal arrow represents a match between four bases, or a single byte, from the query and collection sequences, a horizontal arrow represents a single insertion in the collection sequence, and a vertical arrow represents four insertions in the query.

on the right-hand side illustrates the case where the optimal alignment contains insertions in the query sequence. In Figure 6.3 (c) the optimal alignment contains two insertions in the query. However, gaps can only start and end on the collection sequence byte boundary in a bytepacked alignment and, as a result, only insertions in the query of length i where $i \equiv 0, \text{modulo}\, 4$ are permitted. Therefore, the bytepacked alignment must contain a pair of *adjacent gaps* — one in the query of length four followed immediately by another in the collection sequence of length two — to move the alignment to the new diagonal in Figure 6.3 (d).

To minimise the scoring penalty associated with adjacent gaps in bytepacked alignments we have employed the two state variation of the Gotoh gapped alignment algorithm [Durbin, 1998] that was described in Section 2.2.5 on page 33. The two state variation produces identical alignment scores to the original approach except for alignments that contain adjacent gaps. In the two state variation a single open gap penalty is incurred for the pair of adjacent gaps, whereas two open gap penalties are applied in the original approach. By reducing the scoring penalty for adjacent gaps, bytepacked alignment provides a better approximation of gapped alignment scores.

The alignment algorithm described in Section 2.2.5 that supports affine gap costs and is employed by BLAST records three values for each cell in the dynamic programming matrix; $B(i,j)$ is the highest score for any alignment ending at $[i,j]$, $I_q(i,j)$ is the highest score for any alignment ending at $[i,j]$ with an insertion in the query, and $I_s(i,j)$ is the highest score for an alignment ending at $[i,j]$ with an insertion in the collection sequence. The two state variation combines $I_q(i,j)$ and $I_s(i,j)$ into a single maximum value, $I(i,j)$, that represents the best score for an alignment ending at $[i,j]$ with an insertion in either direction. The following recurrence relations are used to perform bytepacked alignment between a compressed collection sequence s and the query sequence q that is represented using the special overlapping quadruplet representation Q:

$$
\begin{aligned}
M(i,j) = &\quad B(i-4,j-4) + \overline{\overline{=}}_M(q_{[i-3:i]}, s_{[j-3:j]}) \\
I(i,j) = &\quad \max \begin{cases} M(i-1,j) - d \\ I(i-1,j) - e \\ M(i,j-4) - d - 3e \\ I(i,j-4) - 4e \end{cases} \\
B(i,j) = &\quad \max \begin{cases} I(i,j) \\ M(i,j) \end{cases}
\end{aligned}
$$

where the temporary scalar $M(i,j)$ represents the best score for any alignment ending at $[i,j]$ with four matching bases. In addition to the recurrence relations, initialisation rules are used to handle boundary conditions in the matrix; all cells where $j = 0$ or $-3 \geq i \geq 0$ are initialized to $-\infty$, except for the starting point $[0,0]$ which is initialised to zero.

Bytepacked alignment places restrictions on the location of gaps in the alignment. However, we expect the restrictions to have a minor effect on alignment scores for similar reasons to those presented in Section 4.1.2 in relation to semi-gapped alignment:

1. Bytepacked alignment still permits gaps, but forces them to occur in a suboptimal location or with a different arrangement. In cases where the optimal alignment contains a gap in the collection sequence, bytepacked alignment shifts the start and end location by no more than 2 bases to ensure the gap occurs on the collection sequence byte boundary. Unless the optimal gap is adjacent to high-scoring bases, the shift will not significantly change the alignment score. In cases where the optimal alignment contains a gap of length G in the query sequence, the corresponding bytepacked alignment will

contain a gap of length $\lceil \frac{G}{4} \rceil \times 4$ in the query and a gap of length $4 - G$ modulo 4 in the collection sequence. Assuming gap lengths are uniformly distributed, this results in an average additional gap penalty of $3e$ when the two state algorithm is used. If the recurrence relations described in Section 2.2.5 were used instead, the average increase in gap penalty would be $d+2e$. This illustrates the advantage of the two state variation, since d is usually much larger than e.

2. The scoring penalty associated with a gap typically outweighs any additional penalty incurred by opening the gap at a suboptimal location or increasing the length of the gap. Further, the additional penalty can be compensated for by decreasing the open gap penalty, as shown in our previous work with protein alignments in Chapter 4. We report experiments with varying the open gap penalty used for bytepacked alignment in Section 6.2.

In terms of computational cost, bytepacked alignment represents a significant saving when compared to gapped alignment. The number of cells in the dynamic programming matrix is reduced approximately by a factor of four and the amount of computation per cell is almost unchanged. Some additional computation is required to match a pair of bytes, instead of a pair of characters, when calculating $\overline{\overline{M}}(q_{[i-3:i]}, s_{[j-3:j]})$ for each cell. However, $\overline{\overline{M}}$ can be calculated quickly by performing a binary XOR and using a specially designed lookup table to calculate the score. Further, our results in Section 6.2 show that bytepacked alignment offers a substantial speed gain.

Similar to our semi-gapped alignment technique that is described in Section 4.1.2, we have found that bytepacked alignment is best employed as an additional filtering step between the ungapped and gapped alignment stages of BLASTN. Therefore, we need a method for deciding which bytepacked alignments should be passed on to the gapped alignment stage. Once again, we have chosen to perform gapped alignment on collection sequences with a bytepacked alignment score above $R \times S2$, where S2 is the nominal score required to achieve the E-value cutoff and $0 < R \leq 1$. Figure 6.5 illustrates the new process for scoring sequences using bytepacked alignment. In the example, an ungapped extension is performed first, followed by a bytepacked alignment where gaps can start and end only on the byte boundary. The bytepacked alignment scores above $R \times S2$ and the collection sequence is unpacked and realigned using gapped alignment. The choice of R affects the speed and sensitivity of BLAST, and we report experiments with varying values of R in Section 6.2.

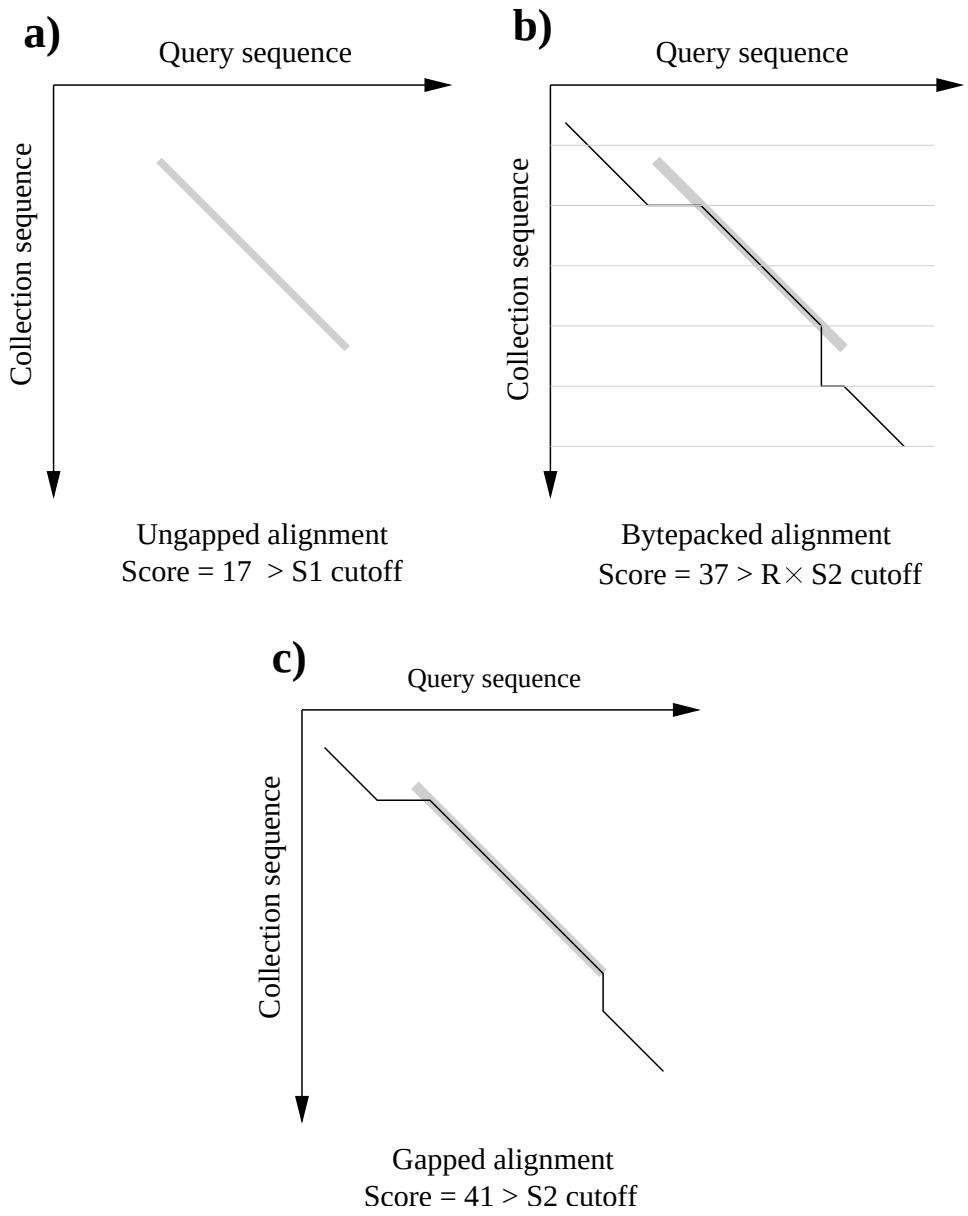

Figure 6.5: Process for scoring sequences using bytepacked alignment. First, an ungapped extension is performed and the resulting alignment scores above the S1 cutoff (a). Next, a bytepacked alignment is performed where gaps can start and end only on byte boundaries, illustrated by grey horizontal lines (b). The resulting alignment scores above $R \times S2$ and is passed onto the last stage, where an unrestricted gapped alignment is performed (c).

Table-driven alignment

In this section, we propose our novel table-driven alignment, which is an alternative to the bytepacked alignment approach. The approach is based on the Four Russians concept [Wu et al., 1996] that involves dividing a problem into subsections and solving each subsection using precomputed answers found in a lookup table. In the context of sequence alignment, the dynamic programming matrix is divided into blocks of adjacent cells that are processed simultaneously. The values of proceeding cells are used as input into the table and alignment scores for each cell in the current block are provided as output.

The Four Russians concept has previously been applied to general string matching problems [Wu et al., 1996], regular expression pattern matching [Myers, 1992], protein sequence alignment [Myers and Durbin, 2003], and the alignment of nucleotide sequences compressed using the Lempel-Ziv compression scheme [Crochemore et al., 2002]. The latter application is closely related to our approach, however it is unclear if Lempel-Ziv compression is suitable for use with the first two stages of BLAST. Instead, we have applied the Four Russians approach to the alignment of byte packed nucleotide sequences permitting four rows of the alignment matrix to be processed at a time. This is achieved by dividing the alignment matrix into subsections of size 1×4 that correspond to the alignment of one base from the query and one packed byte from the collection sequence. Similar to the other techniques discussed in this chapter, this provides two major advantages over gapped alignment; first, the collection sequence does not need to be decompressed for the alignment to be performed, and second, a significant reduction in computation is achieved by using a lookup table to process four characters from the collection sequence at a time.

Figure 6.6 illustrates the table-driven alignment approach to processing four cells in the matrix at a time with consideration of all possible alignments, including gaps, between a single query base q_{a+1} and a packed byte $s_{[b+1:b+4]}$ from the collection sequence. A byte packed collection sequence and uncompressed query sequence are used as input. On the left-hand side of Figure 6.6, the alignment matrix is divided into blocks of adjacent cells. Values for the preceeding grey cells have already been computed, and values for the four empty cells shown on the right-hand side of the figure are calculated by consulting a lookup table. The input into the table consists of values for the six neighbouring grey cells and a *match vector* that specifies which bases in the packed byte $s_{[b+1:b+4]}$ match the query base q_{a+1}. The arrows in the right-hand side of the figure represent the match, mismatch and insertion events that are considered for each block.

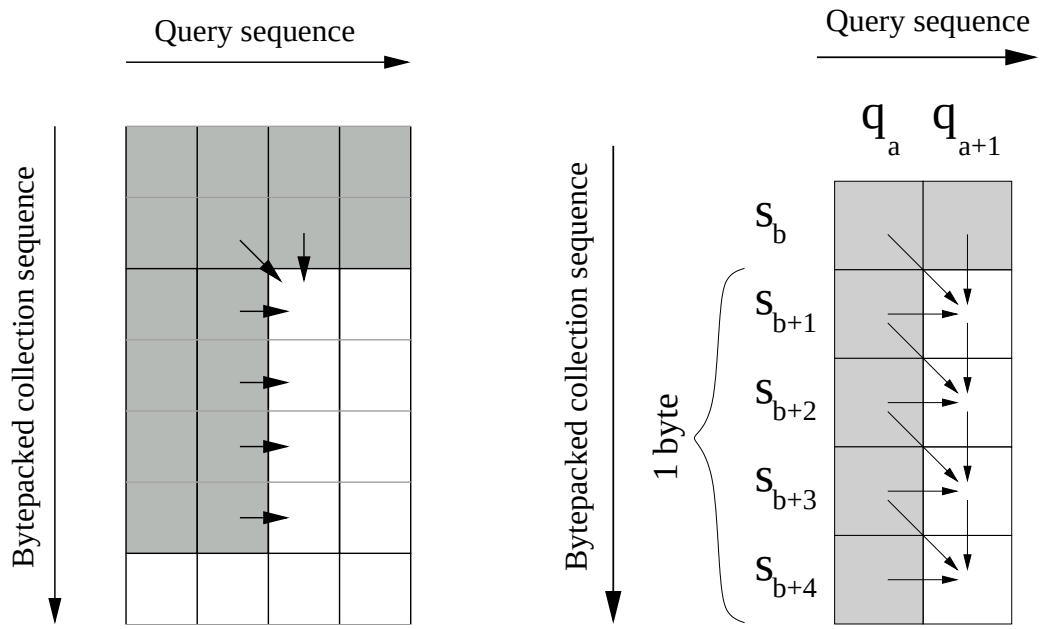

Figure 6.6: Illustration of table-driven alignment. The alignment matrix is divided into blocks of four adjacent cells, and alignment scores for each block are calculated using a lookup table. Previously computed values in the six neighbouring grey cells and a *match vector* that specifies whether the query base q_{a+1} matches each base in the collection sequence byte $s_{[b+1:b+4]}$ are used as input into the table, which provides values for the four white cells.

A significant problem in implementing the Four Russians approach is keeping the lookup table small; a large table will not fit into CPU cache, leading to poor performance due to an increase in latency associated with accessing main memory. Gotoh's alignment algorithm [1982] that is described in Section 2.2.5 records three values, $B(i,j)$, $I_x(i,j)$ and $I_y(i,j)$, for each cell. The values for six cells are used as input into the table, as illustrated in Figure 6.6, resulting in a total of 18 distinct inputs and a prohibitively large table. Therefore, we have made two optimisations to reduce the size of the table.

Our first optimisation is to use the simplified recurrence relations presented in Section 2.2.3 that do not allow for affine gap costs when performing the alignment:

$$B(i,j) = \max \begin{cases} B(i,j) + s(q_i, s_j) \\ B(i-1,j) - e \\ B(i,j-1) - e \end{cases}$$

where e is the cost of each insertion, and no penalty is applied for the opening of a gap. As

a result, only one value is recorded for each cell in the matrix; $B(i, j)$ represents the highest score for any alignment ending at $[i, j]$. This reduces the number of scores used as input into the lookup table from 18 to 6. Our results in Section 6.2 show this approach still provides a good approximation of affine gap cost alignment scores, and works well in practice.

Our second optimisation is to use the difference between neighbouring values rather than absolute values as input into the table, an approach best described by Jones and Pevzner [2004]. Encoding differences rather than absolute values reduces the size of the lookup table, because the differences are limited to a smaller range. Let us define the difference in score $\Delta B(i_C, j_C, i_D, j_D) = B(i_D, j_D) - B(i_C, j_C)$ where $[i_C, j_C]$ and $[i_D, j_D]$ are immediately neighbouring cells such that $i_D = i_C, j_D = j_C + 1$ or $j_D = j_C, i_D = i_C + 1$. When the above recurrence relations are used, the value of ΔB is limited such that $-e \leq \Delta B(i_C, j_C, i_D, j_D) \leq e + r$ where r is the alignment score for two matching bases, $r > 0$. When the default BLAST values of $m = 1$ and $e = 2$ are used, the difference between any cell and its immediately proceeding neighbour is no less than -2 and no more than 3, a range of 6 possible values.

To illustrate how our second optimisation is applied to table-driven alignment, let us return to the illustration in Figure 6.6. The cell $[a, b]$ acts as a reference, with other inputs and outputs encoded relative to its value. The remaining values in the q_a column are encoded as $\Delta B(a, j - 1, a, j)$ where $b + 1 \geq j \geq b + 4$. Similarly, the value for cell $[a + 1, b]$ is encoded as $\Delta B(a, b, a + 1, b)$. This provides a total of 5 inputs into the table, each with $2e + r + 1$ possible values. The only remaining input necessary for calculating scores for the empty cells is the match vector, the 4-bit value specifying if the query base q_a matches or mismatches each of the bases $s_{b+1} \ldots s_{b+4}$. Using this optimised design, the lookup table contains a total of $(2e + r + 1)^5 \times 2^4$ or 124,416 entries when default BLASTN parameters are used.

We have carefully implemented the Four Russians approach to minimise the amount of computation per block. The match vector is generated efficiently by performing a binary XOR between the packed byte $s_{[b+1:b+4]}$ from the collection sequence and a special byte packed representation of q_{a+1} that contains the single query base repeated four times, $q_{[a+1,a+1,a+1,a+1]}$. The result of the binary XOR operation is then used to fetch the match vector from a small, specially designed lookup table. We have also used the *carry in*, *carry out* method when implementing our Four Russians approach [Wu et al., 1996]. A *row carry in* codeword specifies the four values $\Delta B(a, j - 1, a, j) \mid b + 1 \geq j \geq b + 4$ and is used as input into the lookup table. The output of the table, which we refer to as *row carry out*, includes a codeword formatted in the same manner as row carry in and specifies the four values $\Delta B(a + 1, j - 1, a + 1, j) \mid b + 1 \geq j \geq b + 4$. The row carry out codeword can then be

used as a row carry in when processing the next column in the dynamic programming matrix without the need to decode it; this provides a significant computational saving. Similarly, the value $\Delta B(a, b, a+1, b)$ is the *column carry in* used as input into the table, and the table outputs a *column carry out* specifying the value of $\Delta B(a, b+4, a+1, b+4)$. The column carry out can then be used as the column carry in when processing the next four rows in the current column.

Unlike our bytepacked alignment scheme, table-driven alignment considers all possible paths through the alignment matrix without any restriction on the locations of gaps. Although the table-driven approach does not support affine gaps costs, we set the table-driven alignment insertion penalty e_t to equal the gapped alignment extension penalty $e_t = e_g$. Assuming the dropoff regions explored by gapped alignment and table-driven alignment are the same, the latter produces alignment scores equal to or greater than those produced by the former. Any table-driven alignment that scores above the nominal score required to achieve cutoff $S2$ is rescored using gapped alignment, guaranteeing no loss in accuracy when table-driven alignment is used. The main drawback with table-driven alignment is its reliance on fast access to main-memory to consult the lookup table when calculating values for each subsection of the alignment matrix. This relies upon the lookup table fitting into CPU cache for reasonable performance.

6.1.4 Summary

In this section, we have proposed novel approaches for comparing nucleotide sequences. Our schemes permit collection sequences to be processed in their compressed byte packed form and compare four bases at a time with the aid of numeric comparisons and specially designed lookup tables. We have described novel approaches to the first two stages of BLASTN that produce identical results to the conventional approach employed by NCBI-BLAST where collection sequences are decompressed before each stage of the search. Our most significant contribution are two new, alternative approaches to gapped alignment. The first permits insertions and deletions to start and end on the byte boundry only, allowing matches between four bases at a time. The second employs the Four Russians technique of Wu et al. [1996] to align four consecutive bases, with consideration for gaps, using a lookup table of precomputed answers.

In the next section, we present an experimental evaluation of our novel approaches by integrating them into our own implementation of BLAST.

6.2 Results

In this section we compare NCBI-BLAST to FSA-BLAST, our own version of BLAST that incorporates the improvements to nucleotide search described in this chapter. First, we describe the collections and performance and accuracy measures used for our experiments. We then present overall results for our approaches, including bytepacked and table-driven alignment. Finally, we present results for varying parameter choices for our schemes.

6.2.1 Test collection, environment and measurements

Annotated protein sequence collections such as the SCOP [Murzin et al., 1995; Andreeva et al., 2004] and PIR [Wu et al., 2003] databases provide sequence classifications that are ideal for evaluating the retrieval effectiveness of protein homology search tools using assessment techniques such as those discussed in Section 3.3.3. Unfortunately, no similarly annotated collections exist for measuring the accuracy of nucleotide search tools and their ability to identify related sequences. Therefore, we have employed the approach described by Li et al. [2004] for measuring retrieval effectiveness by comparing search results to the complete set of alignments identified by the exhaustive Smith-Waterman algorithm [1981].

A reduced version of the GenBank non-redundant (NR) nucleotide database[1] is the largest collection available for search through the NCBI online BLAST interface, and the default collection for search. For our evaluation, we used a copy of the NR database downloaded on 4 April 2005. To minimise the time spent generating Smith-Waterman alignments for our experiments, we used only half of the NR database by randomly extracting sequences from the collection and creating our own test collection, which we refer to as NR/2. The NR/2 collection contains 6,862,797,036 basepairs in 1,511,546 sequences, ranging in length from 6 to 36,192,742 basepairs. We use the NR/2 database to measure both search runtime and accuracy in our experiments.

A set of 50 test queries were randomly extracted from the NR database. Queries longer than 10,000 basepairs — typically entire genomes or chromosomes — were excluded from the selection process; BLAST searches with longer queries are too slow to be practical and less sensitive genome search tools such as BLAT and MEGABLAST are better suited to such searches. Queries were pre-filtered using our own implementation of the DUST filter, identical in output to the version of the filter used by NCBI-BLAST. Each query was searched against the

[1] This does not include EST, STS, and GSS sequences, environmental samples, or phase 0, 1, or 2 HTGS sequences.

entire NR/2 database using our own implementation of the Smith-Waterman algorithm that is distributed with FSA-BLAST. We use our implementation of Smith-Waterman instead of the SSEARCH tool distributed with FASTA to ensure identical scoring and statistical calculations between our baseline and BLAST [Karlin and Altschul, 1990; Altschul and Gish, 1996].

To evaluate the accuracy of BLASTN, we used each of our 50 test queries to search the NR/2 collection using default parameters, including $v = 250$ where a maximum of v alignments are displayed to the user. We compared the alignments returned by BLAST to the best L alignments returned by Smith-Waterman, ranked by nominal score. We used $L = v$ by default, except for queries where the set of v best alignments was ambiguous because alignments had identical scores. For these queries we set L to the smallest value such that $L \geq v$ and $a_L > a_{L+1}$ where a_i is the score of the i^{th} alignment.

For each query we measured search accuracy with the commonly-used *Receiver Operating Characteristic* (ROC) [Gribskov and Robinson, 1996], using the best L Smith-Waterman alignments as the set of true positives. The ROC score is calculated as:

$$ROC = \frac{1}{nL} \sum_{1 \leq F \leq n} u^F$$

where F is the ranked position of the F^{th} false positive in the list of alignments returned by BLAST, u^F is the number of true positives that are ranked ahead of the F^{th} false positive, and n is the number of false positive alignments returned by BLAST. We only consider an alignment to be positive if the score returned by BLAST is at least half the optimal Smith-Waterman alignment score, an approach also used by Li et al. [2004]. The final reported ROC score is calculated by taking the average across all queries where $L > 1$. This provides a good measure of the sensitivity of BLAST and the level of accuracy for identifying the best v alignments.

For our timing experiments, we compared each of the 50 test queries to the NR/2 database and recorded the best elapsed time of three runs. We report the average search time for 50 queries throughout this section. Experiments were carried out on a Pentium 4 2.8GHz workstation with 2 Gb of main-memory under light load. We used version 2.2.10 of NCBI-BLAST as our baseline and all code was compiled with default NCBI-BLAST compiler flags and optimisations. All experiments were conducted using default BLAST parameters including a gapped alignment trigger score of 25.0 bits, an ungapped extension dropoff of 20.0 bits, a gapped extension dropoff of 30.0 bits, and an E-value cutoff of 10.0. Stage 3 search times include table-driven and bytepacked alignment times when these approaches were used.

	NCBI-BLAST	FSA-BLAST	
		Bytepacked alignment	Table-driven alignment
Stage 1 (secs)	21.85	11.22	11.22
Stage 2 (secs)	1.23	0.70	0.70
Stage 3 (secs)	2.25	0.37	0.46
Stage 4 (secs)	0.34	0.19	0.26
Total (secs)	25.67	12.48	12.64
ROC	0.973	0.974	0.973

Table 6.1: Average runtime in seconds for each stage of BLAST and ROC search accuracy when searching the NR/2 database. All alignment techniques use default parameters.

6.2.2 Overall results

An overall comparison between NCBI-BLAST and our own implementation of BLAST based on the improvements described in this chapter is presented in Table 6.1. Our results show that FSA-BLAST is more than twice as fast as NCBI-BLAST for nucleotide searches regardless of whether table-driven alignment or bytepacked alignment is employed. Our new method for extending hits without decompressing collection sequences and our new lookup table design reduce average search times by 10.63 seconds, equivalent to a 49% reduction in Stage 1 search time. Our new algorithm for performing ungapped alignment using compressed collection sequences results in a further saving of 0.53 seconds per query, with Stage 2 times reduced by 43%. Our bytepacked alignment scheme results in a 78% reduction in time for the gapped alignment stages of BLAST, and our table-driven alignment scheme results in an 72% reduction in time for the final two stages. Although both schemes align collection sequences one byte at a time, bytepacked alignment provides slightly better performance than table-driven alignment; this is likely due to the latency incurred by table-driven alignment when accessing the in-memory lookup table. Importantly, there is no significant change in accuracy between NCBI-BLAST and FSA-BLAST when either scheme is employed. (A wilcoxon sign-ranked test on pairs of ROC scores for each query produced a p-value of $p \leq 1.0$.)

Further experiments reveal that table-driven and bytepacked alignment are indeed accurate approximations of gapped alignment and effective for detecting potentially high-scoring alignments. When bytepacked alignment is employed on average only 1,039 out of 107,076 bytepacked alignments score above $R \times S2$ and are realigned using gapped alignment. Roughly

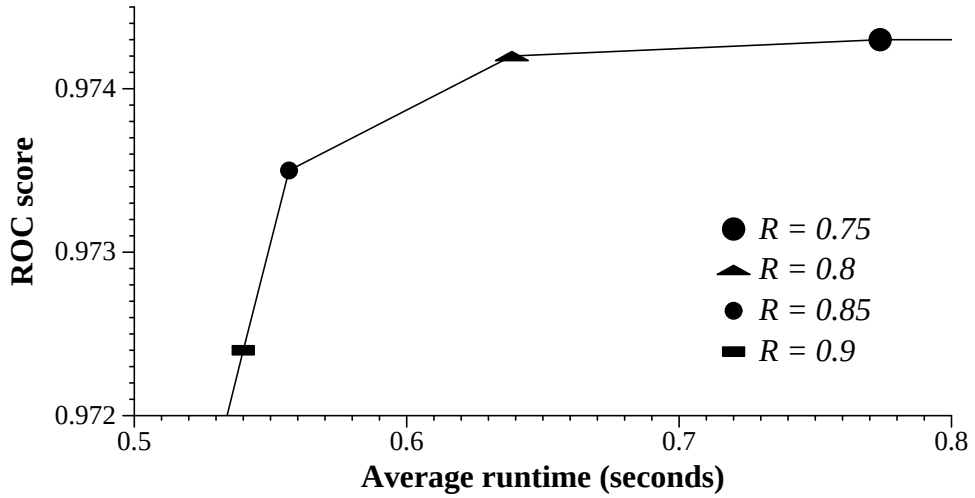

Figure 6.7: Average runtime and ROC search accuracy for searches against NR/2 database using bytepacked alignment and varying values of R.

99% of high-scoring ungapped alignments are discarded by the bytepacked approach. Since bytepacked alignment does not lead to a loss in search accuracy (Table 6.1), this indicates that it provides a good approximation of optimal alignment scores. When table-driven alignment is employed on average only 322 out of the 107,076 table-driven alignments score above $S2$, with less than 0.5% of table-driven alignments rescored using the gapped alignment approach. The result indicates that alignment using non-affine gap penalties can provide a good approximation of the optimal alignment score with affine gap costs.

Experiments were conducted using default bytepacked alignment parameters $o_b = 0$ and $R = 0.85$, where o_b is the open gap penalty used for bytepacked alignment and R controls the rescoring of collection sequences using gapped alignment as discussed in Section 6.1.3. We present the results of our experiments with varying values of R and o_b next.

6.2.3 Varying R

The bytepacked alignment scheme is parameterised by the constant R that provides a trade-off between alignment speed and accuracy. As discussed in Section 6.1.3, only collection sequences with a bytepacked alignment score above $R \times S2$, where S2 is the nominal score required to achieve cutoff and $0 < R \leq 1$, are realigned using gapped alignment. Figure 6.7 shows the effect of varying R on the runtime and accuracy of FSA-BLAST when bytepacked alignment is employed. As R decreases, more collection sequences are rescored using gapped

Open gap penalty (o_b)	0	1	2	3	4	5
R	0.85	0.80	0.75	0.75	0.55	0.55
ROC	0.974	0.974	0.973	0.973	0.973	0.973
Time (seconds)	0.28	0.30	0.32	0.32	0.33	0.34

Table 6.2: Average query evaluation time (stages 3 and 4 only) for varying bytepacked alignment open gap penalties. For each penalty, and value of R that produces an equivalent ROC score was chosen.

alignment resulting in improved accuracy and longer search times. A larger value of R results in fewer gapped alignments and reduced search times at the expense of search accuracy, with BLAST more likely to miss high-scoring alignments. The highlighted value $R = 0.85$ provides a good compromise between search speed and accuracy, and we have chosen this value as our default setting.

6.2.4 Varying the Open Gap Penalty

Bytepacked alignment places restrictions on the location of gaps, resulting in suboptimal alignments as discussed in Section 6.1.3. Therefore, in most cases where the optimal alignment contains insertions or deletions, the bytepacked alignment score and gapped alignment score differ. To compensate for this, we consider lowering the open gap penalty used for bytepacked alignment, with the aim of providing a better approximation of the gapped alignment score. Let us define o_b and e_b as the open and extend gap penalties used for bytepacked alignment. Because gap length has little effect on the constraints imposed by bytepacked alignment, we let $e_b = e$. Next, we select pairs of values for o_b and R with similar accuracy and measure the average search time. The results for this experiment are shown in Table 6.2. The results indicate that a lower open gap penalty provides faster average search times with comparable accuracy, and that the minimum penalty $o_b = 0$ provides best results; we use this value as our default. Note that an open gap penalty of zero implies a non-affine gap cost, with the added advantage of simplifying the alignment algorithm and reducing the computation per cell in the alignment matrix. We have exploited this in our implementation of bytepacked alignment.

6.2.5 Summary

In this section, we evaluated our novel schemes for comparing byte packed nucleotide sequences in BLAST. When intergrated into FSA-BLAST, we found that our improvements to the first and second stages reduce average query evaluation times by 43%. Further, our bytepacked alignment and table-drive alignment methods, when employed as additional filtering steps between the ungapped and gapped alignment stages, reduced the time required to align sequences by 78% and 72% respectively. We also experimented with varying the R parameter that controls the minimum bytepacked alignment score required to trigger a gapped alignment, and found that a lower bytepacked alignment open gap penalty provides faster runtimes with comparable search accuracy.

6.3 Conclusion

With such a large number of nucleotide searches being performed each day, it is important that BLASTN is both efficient and accurate, especially given the rapid grown of sequence databanks such as GenBank. In this chapter, we have described several improvements to BLASTN that permit faster nucleotide searches. Our schemes rely on comparing nucleotide sequences using a simple byte packed representation, leading to reduced search times in two ways: first, collection sequences can be processed using their on-disk representation without the need to decompress them, and second, sequences can be aligned much faster by comparing four nucleotide bases at a time.

We have presented optimisations to the first two stages of BLAST that result in a 43% reduction in average search time. We have also described two new approaches to gapped alignment that allow the alignment of compressed collection sequences. Our first alignment algorithm, bytepacked alignment, places restrictions on the location of gaps permitting four bases to be aligned at a time. Although the scheme is a heuristic, our results indicate that when carefully applied to BLAST, bytepacked alignment reduces the time taken to perform gapped alignment by 78% with no significant effect on accuracy. Our second alignment algorithm, table-driven alignment, uses the Four Russians approach of dividing the alignment matrix into sections and solving each section using precomputing values from a lookup table. Table-driven alignment is lossless, and reduces the time taken by BLAST to perform gapped alignment by 72%.

When combined, our improvements to BLASTN more than halve average query evaluation times for nucleotide searches. Further, the ideas described in this chapter are not only

applicable to BLAST. Indeed, several other search tools use byte packed compression to store collection sequences including PATTERNHUNTER [Li et al., 2004], BLAT [Kent, 2002], MEGABLAST [Zhang et al., 2000], SENSEI [States and Agarwal, 1996], and SSAHA [Ning et al., 2001]. We expect that many of the schemes presented in this chapter would yield similar speed improvements when applied to these algorithms.

In the next chapter, we describe new methods for managing internal redundancy that is prevalant in genomic collections such as GenBank.

Chapter 7

Managing Redundancy

This chapter contains material that appeared in Cameron et al. [2006b], Bernstein and Cameron [2006] and Cameron et al. [2006a] and is based on research conducted in collaboration with fellow PhD candidate Yaniv Bernstein. Yaniv Bernstein adapted the DECO package to process genomic sequence data and provided an implementation of the slotted SPEX approach. I implemented the clustering algorithm and adapted BLAST to search clusters defined by our method. We both contributed equally to the invention of the slotted SPEX algorithm, our scheme for managing redundancy using union-sequences and wildcards, and to the experimental evaluations presented in this chapter.

Genomic data banks often contain a large amount of internal redundancy with the presence of near-duplicate entries [Holm and Sander, 1998; Park et al., 2000b; Li et al., 2001b; Rapp and Wheeler, 2005]. Highly-similar sequences may appear in collections for several reasons, including the existence of closely-related homologues or partial sequences, sequences with expression tags, fusion proteins, post translational modifications, and sequencing errors.

Database redundancy has several pernicious effects on genomic search; a larger database that contains redundant entries takes longer to query, produces repetitive results for any over-represented domain that matches the query, skews measures of the statistical significance of alignments [Park et al., 2000b], and can lead to profile saturation in iterative search tools such as PSI-BLAST [Park et al., 2000b; Li et al., 2002].

The issue of redundancy in genomic collections is often addressed by creating representative-sequence databases (RSDBs); culled collections in which no two sequences share more than a given level of identity. As discussed in Section 3.3.4 on page 100, such databases have been

shown to significantly improve profile training in iterative search tools such as PSI-BLAST by reducing over-representation of certain protein domains and consequently minimising profile saturation.

However, there are two major drawbacks with existing methods for managing redundancy in large collections. First, methods for identifying near-identical entries are either too slow, or require too much main-memory to be practical for large collections such as GenBank. The most successful existing algorithms use a form of all-against-all comparison that has time complexity quadratic in the number of sequences in the database [Grillo et al., 1996; Holm and Sander, 1998; Park et al., 2000b; Li et al., 2001a;b]. As discussed in Section 3.3.4, They do not scale well with collection size; despite efficient methods for discounting highly dissimilar sequence-pairs most of these algorithms still have a fundamental $O(n^2)$ time complexity in the size of the collection, rendering them increasingly infeasible as genomic databases continue their exponential growth. Other successful approaches based on suffix structures [Manber and Myers, 1993; Gusfield, 1997] do not suffer from the quadratic complexity problem [Gracy and Argos, 1998; Malde et al., 2003], however suffix structures have significant memory overheads and long construction times, making them unsuitable for large genomic collections as discussed in Section 3.3.4 on page 100.

In the first part of this chapter, we describe and apply *fingerprinting* techniques [Manber, 1994; Brin et al., 1995; Heintze, 1996; Broder et al., 1997; Shivakumar and Garcia-Molina, 1999] to isolate candidate pairs of highly similar sequences. Our approach is fast, scales with time linear to the collection size, and has modest memory requirements. We describe our method for applying fingerprinting to genomic sequences and find that it is remarkably accurate for this task. We also apply fingerprinting to the creation of representative-sequence databases [Holm and Sander, 1998; Park et al., 2000b; Li et al., 2001b]. We are able to process the GenBank non-redundant database in around 1.5 hours, while the fastest existing approach, CD-HIT [Li et al., 2001b], requires over 9 hours for the same task. Importantly, there is no detectable change in accuracy.

The second major drawback with existing methods for managing redundancy is that they are less suitable for regular search algorithms such as BLAST [Altschul et al., 1990; 1997] and FASTA [Pearson and Lipman, 1988; 1985] and the final iteration of iterative tools such as PSI-BLAST because, by definition, RSDBs are not comprehensive. This leads to search results that are both less accurate, because the representative sequence for a cluster may not be the one that aligns best with a given query, and less authoritative, because the user is only shown one representative sequence from a family of similar sequences.

In the second part of this chapter, we describe a sequence clustering methodology that lacks the drawbacks of previous approaches for creating a representative-sequence database. Most of the approaches described in Section 3.3.4 identify clusters of near-duplicate sequences, choose one sequence from each cluster as a representative to the database, and delete the other sequences. In contrast, we generate a special union-sequence for each cluster that—through use of wildcard characters—represents all of the sequences in the cluster simultaneously. Through careful choice of wildcards, we are able to achieve near-optimal alignments while still substantially reducing the number of sequences against which queries need to be matched. Further, we store all sequences in a cluster as a set of edits against the union-sequence. This achieves a form of compression and allows us to retrieve cluster members for more precise alignment against a query should the union-sequence achieve a good alignment score. Thus, both space and time are saved without loss in accuracy.

Our method supports two modes of operation: users can choose to see all alignments or only the best alignment from each cluster. In the former mode, the clustering is transparent and the result is comparable to searches when the collection has not been clustered. In the latter mode, the search output is similar to the result of searching a culled representative database, except that our approach is guaranteed to display the best alignment from each cluster and is also able to report the number of similar alignments that have been suppressed.

To investigate the effectiveness of our clustering approach we have integrated it with our freely available open-source software package, FSA-BLAST. When applied to the GenBank non-redundant (NR) database, our method reduces the size of sequence data in the NR database by 27% and improves search times by 22% with no significant effect on overall search accuracy.

This chapter is organised into two main components as follows. In Section 7.1 we describe existing fingerprinting techniques and introduce our novel slotted SPEX algorithm for efficiently identifying near-duplicate sequences in a large collection. We assess the accuracy of our fingerprinting method towards high-scoring pairwise alignments and show that our approach is faster than existing methods for creating a representative database. In Section 7.2 we present our new scheme for managing redundancy that uses a union-sequence containing special wildcard characters to represent a cluster of similar sequences. We describe our methodology for building clusters and present methods for selecting and scoring wildcards. Finally, we present experimental results for our new method. In Section 7.3 we conclude this chapter with some final remarks.

7.1 Fingerprinting for near-duplicate detection

There are several applications for which it is necessary to perform an all-against-all similarity comparison; that is, identify arbitrary pairs of highly-similar sequences in a collection. For example, the assembly of EST (expressed sequence tag) data involves arranging a collection of overlapping sequences into a longer consensus sequence [Burke et al., 1999; Malde et al., 2003]. Unlike in previous chapters, for this application there is no apparent query sequence: rather, we are interested in similarity between any pair of sequences in the collection. Another application where an all-against-all comparison is required is the construction of a representative-sequence database (RSDB), where highly redundant sequences are removed from a collection resulting in faster, more sensitive search for distant homologies [Park et al., 2000b; Li et al., 2002] using search algorithms such as PSI-BLAST [Altschul et al., 1997]. The RSDB can be simply constructed by extracting each sequence from the collection in turn, and comparing it to an initially empty representative collection. If the sequence is highly similar to a sequence in the representative collection then it is deleted, otherwise it is added to the new collection.

Although it is possible to use an algorithm such as BLAST to detect all pairs of similar sequences in a collection, such an approach is prohibitively time-consuming. In Section 3.3.4 we discuss several past solutions that use a simple pairwise approach to identify pairs of similar sequences including Holm and Sander [1998] and Li et al. [2001b]. These schemes use fast BLAST-like heuristics to compare each sequence in the collection to the entire collection. The representative-sequence database tool CD-HIT [Li et al., 2001b] is the fastest approach based on this method. However, despite fast methods for comparing each sequence pair, such approaches require time that is quadratic in the size of the collection and are increasingly infeasible as genomic collections continue to grow. We show in Section 7.1.5 that the CD-HIT tool requires over 9 hours to process the current GenBank non-redundant protein database.

We propose a method that takes advantage of the following trend for sequence databases such as GenBank: of all the sequence-pairs in a collection, only a very small fraction have any significant level of mutual redundancy. In this section we describe a novel, alternative approach for identifying highly-similar sequences in a collection called *fingerprinting* that has linear time complexity and modest memory requirements.

This section is organised as follows. First, we describe document fingerprinting methods that are commonly applied to Information Retrieval. Next, we present the SPEX algorithm [Bernstein and Zobel, 2004] that provides efficient, lossless fingerprinting of large collections

Original document	Collection of six word chunks
	`[the quick brown fox jumped over]`
the quick brown fox jumped	`[quick brown fox jumped over the]`
over the lazy dog	`[brown fox jumped over the lazy]`
	`[fox jumped over the lazy dog]`

Figure 7.1: Set of chunks that are six words in length for a document containing the text "the quick brown fox jumped over the lazy dog"

in Section 7.1.2. In Section 7.1.3 we adapt existing fingerprinting methods to genomic data with our new slotted SPEX method. In Section 7.1.4 we assess the accuracy of our approach for identifying pairs of sequences with greater than 90% identity; we find that it is remarkably accurate for this task. In Section 7.1.5 we apply fingerprinting to the creation of representative-sequence databases or RSDBs [Holm and Sander, 1998; Park et al., 2000b; Li et al., 2001b]. We are able to construct an RSDB for the GenBank non-redundant database in around 1.5 hours, while the fastest existing approach, CD-HIT [Li et al., 2001b], requires over 9 hours for the same task. Importantly, we observe no significant effect on accuracy.

7.1.1 Document fingerprinting

Document fingerprinting [Manber, 1994; Brin et al., 1995; Heintze, 1996; Broder et al., 1997; Shivakumar and Garcia-Molina, 1999] is an effective and scalable technique for identifying pairs of documents within large text collections that share portions of identical text. Fingerprinting has been used for several applications, including copyright protection [Brin et al., 1995], document management [Manber, 1994] and web search optimisation [Broder et al., 1997; Fetterly et al., 2003; Bernstein and Zobel, 2005].

The fundamental unit of document fingerprinting techniques is the *chunk*, a fixed-length unit of text such as a series of consecutive words or a sentence. The full set of chunks for a given document is formed by passing a *sliding window* of appropriate length over the document; this is illustrated in Figure 7.1 for a chunk length of six words.

The set of all chunks in a collection can be stored in an inverted index [Witten et al., 1999], similar to that described in Section 3.2.1, and the index can be used to calculate the number of shared chunks between pairs of documents in a collection. Two identical documents will naturally have an identical set of chunks. As the documents begin to diverge, the proportion

of chunks they share will decrease. However, any pair of documents sharing a run of text as long as the chunk length will have at least one chunk in common. Thus, the proportion of common chunks is a good estimator of the quantity of common text shared by a pair of documents. The quality of this estimate is optimised by choosing a chunk length that is long enough so that two identical chunks are unlikely to coincidentally occur, but not so long that it becomes too sensitive to minor changes. In the duplicate detection DECO package, for example, the default chunk length is eight words [Bernstein and Zobel, 2004].

In theory, one could simply store the full set of chunks for each document in a collection, and directly compute the degree of text reuse between documents as above. However, such an approach is highly inefficient and slow. Therefore, practical document fingerprinting techniques apply a number of optimisations to make the process more efficient. First, chunks are generally hashed before storage in order to make their representation more compact. The use of hashing is fairly straightforward and has few attendant problems; as long as the hash is of a sensible size, there will be few collisions and the performance of the algorithm will be largely unaffected. Second, some sort of *selection heuristic* is normally applied so that only some chunks from each document are selected for storage. The choice of selection heuristic has a very significant impact on the general effectiveness of the fingerprinting algorithm. Most fingerprinting algorithms have used simple feature-based selection heuristics, such as selecting chunks only if their hash is divisible by a certain number, or selecting chunks that begin with certain letter-combinations. (See Bernstein and Zobel [2004] for an overview of existing chunk selection methods.)

However a major drawback with these chunk selection schemes is that they are lossy; if two documents share chunks, but none of them happen to satisfy the criteria of the selection heuristic, the fingerprinting algorithm will not identify these documents as sharing text. For example, consider the processing of two documents with a chunk length of one word and a chunk selection heuristic that only selects words starting with a vowel. Given a pair of highly similar documents with the following text:

Document 1 `The yellow cat ran between the tall trees.`
Document 2 `The brown cat ran between the tall towers.`

Not a single chunk would be selected from either of these two documents because all of the words contained in them start with a consonant. As a result, the similarity between these two documents would be overlooked. As the degree of desired selectivity increases, this problem only worsens.

7.1.2 The SPEX chunk selection algorithm

Bernstein and Zobel [2004] introduced the SPEX chunk selection algorithm, which allows for *lossless* selection of chunks, based on the observation that singleton chunks (chunks that only occur once and represent a large majority in most text collections) do not contribute to identifying text reuse between documents. The SPEX algorithm takes advantage of the following observation: if any subchunk (subsequence) of a chunk is unique, the chunk as a whole is unique. For example, an occurrence of the chunk *two brown cats* must be unique in the collection if the subchunk *brown cats* is also unique. Using a memory-efficient iterative hashing technique, SPEX is able to select only those chunks that occur multiple times in the collection. Using SPEX can yield significant savings over selecting every chunk without any degradation in the quality of results [Bernstein and Zobel, 2004; 2005].

The SPEX algorithms works as follows. In the first iteration, each chunk that is one word in length in the document is processed. A hashtable records for each chunk whether it appears multiple times, just once, or not at all. In the second iteration, the entire document is processed again this time considering chunks that are two words in length by passing a sliding window over the document. Each two word chunk is only inserted into the hashtable if both one word subchunks appear multiple times in the collection. For example, the two word chunk *brown cats* is only processed if both *brown* and *cats* are non-unique. A second hashtable records whether each two word chunk appears multiple times, just once, or not at all. The process is then repeated for increasing chunk lengths until the final desired length is reached.

Figure 7.2 provides a pseudocode sketch of how SPEX identifies duplicate chunks of length `finalLength` within a collection of documents (or genomic sequences) S. The algorithm iterates over chunk lengths from 1 to `finalLength`, the final chunk length desired. At each iteration, SPEX maintains two hashtables (referred to as `lookup` in the figure): one recording the number of occurrences of each chunk for the previous iteration, and one for the current iteration. As we are only interested in knowing whether a chunk occurs multiple times or not, each entry in `lookup` takes one of only three values: zero, one, or more than one (2^+). This allows four hashtable entries to be fit into a single byte, significantly reducing the size of the table. A chunk is only inserted into `lookup` if its two subchunks of length `chunkLength - 1` both appear multiple times in the hashtable from the previous iteration.

Figure 7.3 illustrates how the SPEX algorithm works when applied to the pair of documents shown at the top of the figure. First, one word chunks are counted as shown in the far left

column. The words `yellow`, `brown`, `trees`, and `towers` appear only once in the collection and the remaining words appear at least twice. In the second iteration, two work chunks are extracted from the collection and inserted into the hashtable. However, chunks such as `The yellow` that contain at least one unique subchunk from the previous iteration are not processed, nor are they inserted into the hashtable, as indicated by the letters NP. As a result, fewer entries are made into the hashtable and the number of collisions is minimised. Finally, three word chunks are extracted in the third iteration, and any chunk that contains a unique subchunk of length two is dismissed. The end result after three iterations is that all three word chunks that are shared between the two documents (that is, `cat ran between`, `ran between the`, and `between the tall`) are identified.

Although collisions caused by different chunks hashing to the same value are not resolved, Bernstein and Zobel [2004] report that this has minimal impact on the performance of the algorithm. Collisions introduce false positives to the process, that is, cause unique words to be considered as non-unique. However, the iterative process tends to remove such false positives and helps prevent the hashtables from reaching too high a load. As a result, the SPEX algorithm is able to process quite large collections of text and indicate reasonably quickly whether a given chunk occurs multiple times, and consumes a relatively modest amount of memory [Bernstein and Zobel, 2004].

Bernstein and Zobel [2004] have written a fingerprinting package, DECO, that implements the SPEX algorithm. They show that DECO is effective at identifying documents that share text [Bernstein and Zobel, 2004] and have successfully applied it for detecting and filtering redundant documents from search results on web data [Bernstein and Zobel, 2005].

7.1.3 Fingerprinting for genomic sequences

The SPEX algorithm (and, indeed, any fingerprinting algorithm) can be trivially adapted for use with genomic sequences by simply substituting documents with sequences. However, the properties of a genomic sequence are quite different from those of a natural language document, and these differences present a number of challenges to the SPEX algorithm. The most significant difference is the lack of any unit in genomic data analogous to natural language words, given that a genomic sequence is represented by an undifferentiated string of characters with no natural delimiters such as whitespace, commas or other punctuation marks.

The lack of words in genomic sequences has a number of immediate impacts on the

```
for chunkLength ⟵ 1 to finalLength
  foreach sequence in the collection
    foreach chunk of length chunkLength in sequence
      if chunkLength = 1 then
        increment lookup[chunk]
      else
        subchunk1 ⟵ chunk prefix of length chunkLength - 1
        subchunk2 ⟵ chunk suffix of length chunkLength - 1
        if lookup[subchunk1] = 2+ and lookup[subchunk2] = 2+ then
          increment lookup[chunk]
```

Figure 7.2: The SPEX *algorithm.*

Document 1	`The yellow cat ran between the tall trees.`
Document 2	`The brown cat ran between the tall towers.`

Iteration 1 (One word)		Iteration 2 (Two words)		Iteration 3 (Three words)	
`The`	2^+	`The yellow`	NP	`The yellow cat`	NP
`yellow`	1	`The brown`	NP	`The brown cat`	NP
`brown`	1	`yellow cat`	NP	`yellow cat ran`	NP
`cat`	2^+	`brown cat`	NP	`brown cat ran`	NP
`ran`	2^+	`cat ran`	2^+	`cat ran between`	2^+
`between`	2^+	`ran between`	2^+	`ran between the`	2^+
`the`	2^+	`between the`	2^+	`between the tall`	2^+
`tall`	2^+	`the tall`	2^+	`the tall trees`	NP
`trees`	1	`tall trees`	NP	`the tall towers`	NP
`towers`	1	`tall towers`	NP		

Figure 7.3: Three iterations of the main loop in the SPEX *algorithm. The pair of documents at the top are processed to identify duplicate chunks of one, two and three words in length. For each chunk,* 1 *denotes that it appears once only in the collection,* 2^+ *denotes that it appears multiple times, and* NP *indicates that the chunk was not processed because it contains a unique subchunk.*

operation and performance of the SPEX algorithm. First, the granularity of the sliding window must be increased from word-level to character-level. An increased granularity results in a far greater number of chunks in a genomic sequence than in a natural-language document of similar size. This increase becomes apparent when one considers that — assuming an average word length of five — the SPEX sliding window increments over a total of six bytes, including a whitespace, of a natural language document at each move. By contrast, the base-pairs in nucleotide sequence data each represent only two bits, or 0.25 of a byte, of data. Thus, a nucleotide sequence can be expected to produce roughly 24 times as many chunks as a natural language document of the same size. As a result, the SPEX algorithm is less efficient and scalable for genomic data than for natural language documents.

Processing genomic sequences also involves performing more iterations as part of the SPEX algorithm. To identify chunks that are eight words in length SPEX must perform eight iterations; for a similar-length chunk containing 30 characters of sequence data, SPEX would need to perform 30 iterations. This obviously increases the total running time significantly.

The distribution of subsequences within genomic data is also less highly skewed than the distribution of words in English text. Given a collection of natural language documents, we expect some words (such as 'and' and 'or') to occur extremely frequently, while other words (such as perhaps 'alphamegamia' and 'nudiustertian') will be *hapax legomena*: words that occur only once. This permits the SPEX algorithm to be effectual from the first iteration by removing word-pairs such as 'nudiustertian news'. In contrast, given a short string of characters using the amino acid alphabet of size 20, we expect that it is far less likely that the word will occur only once in any collection of nontrivial size. Thus, the first few iterations of SPEX are likely be entirely ineffectual.

One simple solution to these problems is to introduce 'pseudo-words', effectively segmenting each sequence by moving the sliding window several characters at a time. However, this approach relies on sequences being aligned along segment boundaries. This assumption is not generally valid and makes the algorithm highly sensitive to insertions and deletions. Consider, for example, the following sequences given a chunk length of four and a window increment of four:

	Sequence	Chunks
Sequence 1	AB**CDEFGHIJKL**MNOP	ABCD EFGH IJKL MNOP
Sequence 2	AAB**CDEFGHIJKL**MNOP	AABC DEFG HIJK LMNO
Sequence 3	GHAA**CDEFGHIJKL**MQ	GHAA CDEF GHIJ KLMQ

```
1  chunkLength ⟵ finalLength - Q × (numIterations - 1)
2  for iteration ⟵ 1 to numIterations
3     foreach sequence in the collection
4        foreach chunk of length chunkLength in sequence
5           if lookup[chunk] ≠ 0 then
6              increment lookup[chunk]
7           else
8              count number of subchunks of length chunkLength - Q
9                    where lookup[subchunk] = 2+
10             if count ≥ 2 or iteration = 1 and
11                number of chunks processed since increment lookup ≥ Q then
12                increment lookup[chunk]
13    increment chunkLength by Q
```

Figure 7.4: The slotted SPEX *algorithm*

Despite all three of these sequences containing an identical subsequence of length 11 (in bold above), they do not share a single common chunk. This strong correspondence between the three sequences will thus be overlooked by the algorithm.

We propose a hybrid of regular SPEX and the pseudo-word based approach described above that we call slotted SPEX. Slotted SPEX uses a window increment greater than one but is able to 'synchronise' the windows between sequences so that two highly-similar sequences are not entirely overlooked as a result of a misalignment between them. Although slotted SPEX is technically a lossy algorithm, it provides a guarantee that a pair of sequences with a minimum length match will share at least one common chunk. We discuss this guarantee in more detail later in this section.

Figure 7.4 describes the slotted SPEX algorithm. As in standard SPEX, we pass a fixed-size window over each sequence with an increment of one. However, unlike SPEX, slotted SPEX does not consider inserting every chunk into the hashtable. In addition to decomposing the chunk into subchunks and checking that the subchunks are non-unique, slotted SPEX also requires that one of two initial conditions are met. First, that it has been at least `Q` window increments since the last insertion; or second, that the current chunk already appears in the hashtable. The parameter `Q` is the *quantum*, which can be thought of as the window increment used by the algorithm. Slotted SPEX guarantees that at least every $\texttt{Q}^{th}$ overlapping

substring from a sequence is inserted into the hashtable. The second precondition — that the chunk already appears in the hashtable — provides the synchronisation that is required for the algorithm to work reliably.

Let us illustrate the operation of slotted SPEX with an example. Using the same set of sequences as above, a quantum `Q` = 4 and a chunk length of four, slotted SPEX produces the following set of chunks:

	Sequence	Chunks
Sequence 1	AB**CDEFGHIJKL**MNOP	ABCD **EFGH IJKL** MNOP
Sequence 2	AAB**CDEFGHIJKL**MNOP	AABC ABCD **EFGH IJKL** MNOP
Sequence 3	GHAA**CDEFGHIJKL**MQ	GHAA CDEF **EFGH IJKL**

For the first sequence, the set of chunks produced does not differ from the naïve pseudo-word technique. Let us now follow the process for the second sequence. The first chunk, `AABC`, is inserted as before. When processing the second chunk, `ABCD`, the number of chunks processed since the last insertion is one, fewer than the quantum `Q`. However, the condition `lookup[chunk]` $\neq$ `0` on line 5 of Figure 7.4 is met: the chunk has been previously inserted. The corresponding entry in the hashtable is therefore incremented, effectively synchronising the window of the sequence with that of the earlier, matching sequence. As a result, every $\texttt{Q}^{th}$ identical chunk will be identified across the matching region between the two sequences. In this example, the slotted SPEX algorithm selects two chunks of length four that are common to all sequences.

Unlike the original SPEX algorithm which incremented the chunkLength by one for each iteration, the slotted SPEX algorithm uses a larger increment that equals the quantum `Q`. In the slotted SPEX approach, a match of length `chunkLength + Q` between two sequences is guaranteed to contain at least two matches of length `chunkLength` identified during the previous iteration. As a result, slotted SPEX must increase the chunk length by at least the quantum `Q` between iterations to ensure the scheme is lossless.

In comparison to the ordinary SPEX algorithm, slotted SPEX requires fewer iterations, consumes less memory and builds a smaller index, as shown in Section 7.1.4. This makes it suitable for the higher chunk density of genomic data. While slotted SPEX is a lossy algorithm, it does offer the following guarantee: for a window size `finalLength` and a quantum `Q`, any pair of sequences with a matching subsequence of length `finalLength + Q - 1` or greater will have at least one identical chunk selected. As the length of the match grows, so will the

guaranteed number of common chunks selected. Thus, despite the lossiness of the algorithm, slotted SPEX is still able to offer strong assurance that it will reliably detect highly similar pairs of sequences.

We have modified the DECO software to be able to read genomic data files and have implemented the slotted SPEX algorithm using default parameters of `finalLength` = 30 and `Q` = 9.

7.1.4 Fingerprinting for identity estimation

In this section, we analyze the performance of slotted SPEX for distinguishing sequence pairs with a high level of identity from those that do not.

Following Holm and Sander [1998] and Li et al. [2001b], we calculate the percentage identity between a pair of sequences by performing a banded Smith-Waterman alignment [Sankoff and Kruskal, 1983; Chao et al., 1992] that is described in Section 2.3.1 on page 39 using a band width of 20, match penalty of 1, and no mismatch or gap penalty. The percentage identity I for the sequence pair s_i, s_j is calculated as $I = S(s_i, s_j)/L(s_i, s_j)$ where $S(s_i, s_j)$ is the alignment score and $L(s_i, s_j)$ is the length of the shorter of the two sequences. This score can be functionally interpreted as being the proportion of characters in the shorter sequence that match identical characters in the longer sequence. We define similar sequence pairs as those with at least 90% identity ($I \geq 0.9$); this is the same threshold used in Holm and Sander [1998] and is the default parameter used by CD-HIT [Li et al., 2001b].

For experiments in this section we use version 1.65 of the ASTRAL Compendium [Chandonia et al., 2004]. We do not use the associated SCOP family classifications in our experiments, rather we use the ASTRAL database because it is a relatively small yet complete database that allows us to experiment with a wide range of parameterisations. The ASTRAL database contains 24,519 sequences, equating to 300,578,421 unique sequence-pair combinations. Of these, 139,716 — less than 0.05% — have an identity of 90% or higher by the above measure, despite the database having a high degree of internal redundancy as shown in Section 3.3.3 on page 97. A vast majority of sequence pairs in any database can be assumed to be highly dissimilar.

Although we do not expect fingerprinting to be as sensitive as a computationally intensive dynamic-programming approach such as Smith-Waterman, we hope that the method will effectively distinguish sequence-pairs with a high level of identity from the large number of pairs that have very low identity. Our aim is to use document fingerprinting to massively

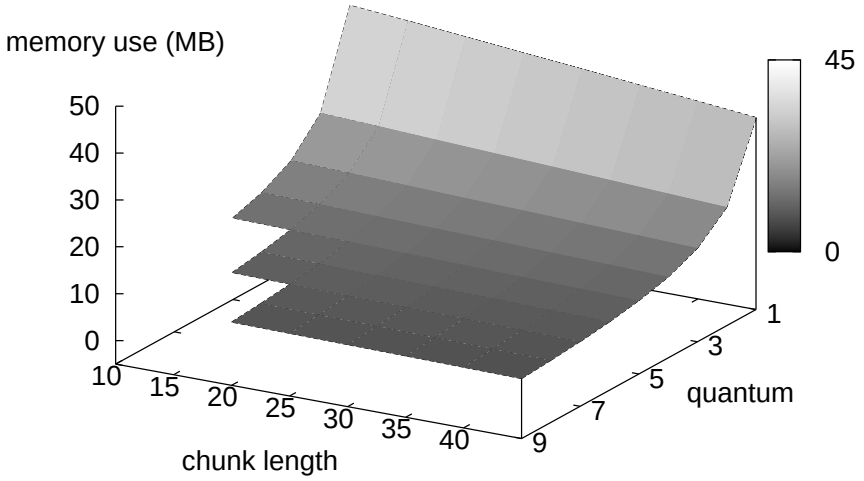

Figure 7.5: SPEX *index size as a function of final chunk length and quantum when processing the ASTRAL collection.*

reduce the search space within which more sensitive analysis must be pursued. For example, even if fingerprinting identifies three times as many false positives (dissimilar sequences) as true positives (similar sequences), less than 0.2% of all sequence pairs in the ASTRAL collection would need to be aligned.

In choosing a chunk length and quantum to use with the slotted SPEX algorithm we are making a compromise between execution speed, memory use and accuracy. In general we would expect memory use to decrease as chunk length and quantum increase, but accuracy will likewise decrease. As is generally the case in classification tasks, an increase in sensitivity (proportion of true positives) can be obtained at the expense of a decrease in specificity (proportion of true negatives).

In order to find a good compromise between resource consumption and effectiveness, we have experimented with different parameter combinations. Figure 7.5 shows the SPEX index size for varying chunk lengths and quantums. The results show that increasing the chunk length does not result in a large reduction in index size, however increasing the quantum results in a marked and consistent decrease in the size of the index.

Figure 7.6 plots the average precision [Buckley and Voorhees, 2000] as a function of chunk

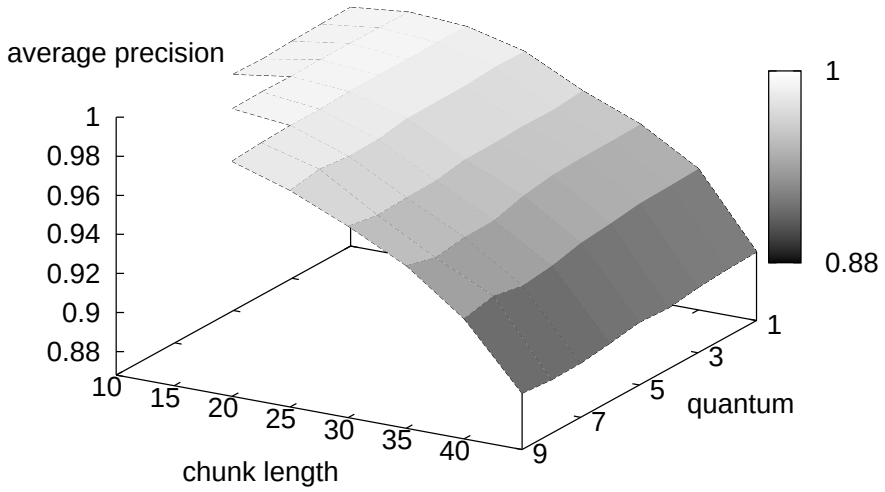

Figure 7.6: Average precision as a function of final chunk length and quantum when processing the ASTRAL collection.

length and quantum. Average precision is a measure of retrieval accuracy that is similar to the ROC measure described in Section 3.3.3 on page 97 and is commonly used in the field of information retrieval[1]. The average precision measure was calculated by sorting pairs in decreasing order of SPEX score — the number of matching chunks divided by the length of the shorter sequence — and using sequence pairs with an identity of 90% or above as the set of positives. We observe that increasing the chunk length results in a small loss in accuracy, however increasing the quantum has almost no effect on average precision. This indicates that slotted SPEX — even with a high quantum — is able to estimate sequence identity nearly as well as the regular SPEX algorithm with reduced costs in memory use, index size and index processing time.

The result in Figures 7.5 and 7.6 make a strong case for using a shorter chunk length; however, shorter chunks place a greater loading on the hashtable. With larger collections, memory bounds can lead to an unacceptably large hashtable load resulting in a large number of collisions and poor performance. Thus, when shorter chunk lengths are employed the

[1]Chen [2004] reports that the ROC and average precision measures are roughly equally effective for evaluating retrieval effectiveness.

approach is less scalable. Similarly, longer quanta are in general beneficial to performance. However, a larger quantum reduces the number of iterations possible in slotted SPEX. Thus, a very high quantum can result in more collisions in the hashtable due to fewer iterations, suggesting once again that a compromise is required. Guided by these observations along with the other data, chunk lengths of 25 or 30 with a quantum of 5 to 9 appear to provide a good compromise between the various considerations in all-against-all identity detection for large collections.

The high average precision results indicate that slotted SPEX provides an accurate prediction of whether sequence pairs have a high level of identity. What is particularly surprising is that the average precision stays high even with reasonably long chunk lengths and high quanta. Earlier efforts by Holm and Sander [1998] and Li et al. [2001b], and Li et al. [2001a] are extremely rigorous and rely upon short matching chunks, typically less than ten characters in length, between sequence-pairs before proceeding with alignment. Every sequence-pair that may *possibly* exceed the identity threshold is fully aligned. This limits the possibilities for the preprocessing step: for example, a sequence-pair with 90% identity is not guaranteed to have any common chunks of length greater than nine; thus nine was the natural upper bound on chunk length. Li et al. [2001b] found that loosening this rigour has only minor negative impact on result quality. Our results extend on this observation, as the chunk lengths we use are considerably longer than for any previous work.

In our experiments we have focused on identifying sequence-pairs with greater than 90% identity, and we have shown that fingerprinting is effective at this task. However, it is probable that fingerprinting will prove less useful as the identity threshold is lowered and number of similar sequence pairs increases.

7.1.5 Removing redundant sequences

Holm and Sander [1998], Park et al. [2000b] and Li et al. [2001b] have all investigated techniques for creating *representative-sequence databases* (RSDBs), culled collections where no two sequences share more than a given level of identity. RSDBs are typically constructed by identifying clusters of similar sequences and retaining only one sequence from each cluster, the cluster representative. Such databases are more compact, resulting in faster search times. More significantly, they have been demonstrated to improve the sensitivity of distant-homology search algorithms such as PSI-BLAST [Li et al., 2002; Park et al., 2000b].

The most recent and efficient technique for constructing an RSDB, CD-HIT [Li et al.,

2001b], uses a greedy incremental approach based on an all-against-all comparison. The algorithm starts with an empty RSDB. Each sequence is processed in decreasing order of length and compared to every sequence already inserted into the RSDB. If a high-identity match is found, where I exceeds a threshold, the sequence is discarded; otherwise it is added to the RSDB. To reduce the number of sequence pairs that are aligned, CD-HIT first checks for short matching chunks — typically of length four or five — between sequences before aligning them. The approach is still fundamentally quadratic in time complexity.

We have replicated the greedy incremental approach of CD-HIT, but use fingerprinting with slotted SPEX as a preprocessing step to dramatically reduce the number of sequence comparisons performed. Slotted SPEX is used to build a list of candidate sequence pairs, where the SPEX score exceeds a specified threshold. We only perform alignments between sequence pairs in the candidate list. This is significantly faster than comparing each sequence to all sequences in the RSDB.

We have employed the SPEX algorithm to remove redundant sequences as follows. First, SPEX is used to identify short substrings or chunks that appear more than once in the collection. The SPEX algorithm outputs a inverted index that contains a postings list for each chunk that appears more than once in the collection. Each postings list contains a list of sequence number and sequence offset pairs. For example, consider the following postings list:

```
QTWR (7,13),(9,4),(11,8)
```

where the chunk `QTWR` appears three times in the collection, once in sequence 7 at offset 13, once in sequence 9 at offset 4, and once in sequence 11 at offset 8. Only the first occurrence of the chunk in each sequence is considered.

We use the inverted index to find pairs of highly-similar sequences in the collection. We proceed through each list in the index linearly, maintaining an *accumulator* for every sequence pair that co-occurs in at least one of the postings lists. The accumulator records the number of matching chunks that co-occur in the two sequences (see Section 3.2.1 for a more detailed description of accumulators). Each postings list is expanded to provide a list of sequence pairs. For example, the expansion of the postings list above would result in the following pairs of sequences: sequences 7 and 9, sequences 7 and 11, and sequences 9 and 11.

Once we have processed every postings list in the index, each pair of sequences with an accumulator shares at least one chunk. This provides a list of candidate sequence pairs. This

information is used to identify near-duplicate sequences by aligning each of the candidate pairs. A new representative collection is then created where no pair of sequences share more than X% identity, where X is an input parameter. This is achieved using the same approach as existing RSDB tools such as CD-HIT: each sequence is processed in order from longest to shortest and is only added to the new representative database if it does not share above X% identity with a sequence already present in the new collection.

One hurdle with our approach, however, is that large collections often include a small number of chunks with a very high frequency of occurrence. For example, the subsequence `WRKLVDFRELNKRTQDFWEVQLGIPHPAGL` appears in over 20,000 sequences relating to human immunodeficiency virus in the GenBank non-redundant database. Expanding the postings lists of frequently occurring chunks involves processing a large number of sequence-pairs. For example, expanding a postings lists with 500 entries involves considering 124,750 possible sequence pairs. In order to minimise the number of times we must expend such an effort, we delay the expansion of long postings lists—those with more than M entries where $M = 20$ by default—until a sequence in the list is processed, that is, the sequence is considered for inclusion in the new representative database. We call this approach *deferred postings list expansion*. Each time a redundant sequence is identified and deleted from the new representative database it is also removed from any long postings lists, substantially reducing the number of sequence pairs in each list that must be subsequently processed. A reference between each sequence and the long postings lists it appears in is maintained so that entries can be removed quickly.

Using this approach, our algorithm has a worst-case time complexity of $O(n^2)$ for a collection of n sequences, in the event that a single chunk appears in every sequence in the collection yet no sequences are removed to create the new representative database, that is, no pair of sequences in the collection have at least X% identity. In practice, however, such a scenario is unlikely when sensible parameters are employed to process real sequence data, because sequence pairs that share a long matching region are typically similar globally. Further, our experimental results presented next demonstrate that our algorithm scales roughly linear with collection size.

We measured the performance and scalability of our approach by comparing it to CD-HIT — which is freely available for download — using several releases of the comprehensive Genbank non-redundant (NR) protein database over time[2]. We used the CD-HIT default

[2]Ideally, we would have had more datapoints for this experiment. However, old releases of the NR database are not officially maintained, and thus we could only find four different releases of the database.

threshold of $T = 90\%$ and the four releases of GenBank NR database from July 2000 until August 2005 described in Table 7.1. For tests with CD-HIT we used default parameters except for `max_memory` which we increased to 1.5 Gb. For our approach, we used a final chunk length `finalLength` of 25, a quantum of 9 and 3 iterations. Our threshold for identifying a candidate pair is one matching chunk between the pair. We use this low threshold because we have found that it provides improved accuracy with a negligible increase in execution time. In our experiments with the ASTRAL database described previously and our chosen default parameters, slotted SPEX identifies only 10,143 false positives out of 147,724 sequence pairs identified.

The results in Table 7.1 show no significant difference in representative collection size between our method and CD-HIT, indicating the two approaches are roughly equivalent in terms of sensitivity to pairs of highly-similar sequences. Figure 7.7 shows the runtime for our approach and CD-HIT for the releases of GenBank tested. A visual inspection reveals that our approach scales roughly linearly with the size of the collection while CD-HIT is superlinear. When processing the recent August 2005 collection, our approach is more than six times faster than CD-HIT.

7.1.6 Summary

The identification of highly-similar sequence pairs in genomic collections has several important applications in bioinformatics. Previous solutions to this problem involve either an all-against-all comparison with $O(n^2)$ complexity or the use of suffix structures that suffer from large main-memory overheads or long construction times. Therefore, existing approaches are not suitable for processing large collections such as GenBank.

We have applied document fingerprinting techniques to genomic data with the aim of more efficiently identifying pairs of similar sequences in large collections. We have described a new algorithm called slotted SPEX that requires less main-memory and CPU resources when processing genomic collections. We show that slotted SPEX is highly accurate for identifying high-identity sequence pairs, even with long chunk lengths and large quanta. We have also tested the effectiveness of our slotted SPEX approach for removing redundant sequences from large collections. When processing the recent GenBank non-redundant protein database our scheme is more than six times faster than the previous fastest approach, CD-HIT, with no significant change in accuracy. Further, our approach scales approximately linearly with collection size.

	Original	Size reduction	
Release date	Size (Mb)	CD-HIT	Our approach
16 July 2000	157	61.71 Mb (39.56%)	61.72 Mb (39.57%)
22 May 2003	443	164.38 Mb (37.33%)	165.07 Mb (37.48%)
30 June 2004	597	217.80 Mb (36.71%)	218.76 Mb (36.87%)
18 August 2005	900	322.98 Mb (36.08%)	324.92 Mb (36.30%)

Table 7.1: Reduction in collection size for CD-HIT *and our approach for various releases of the GenBank NR database.*

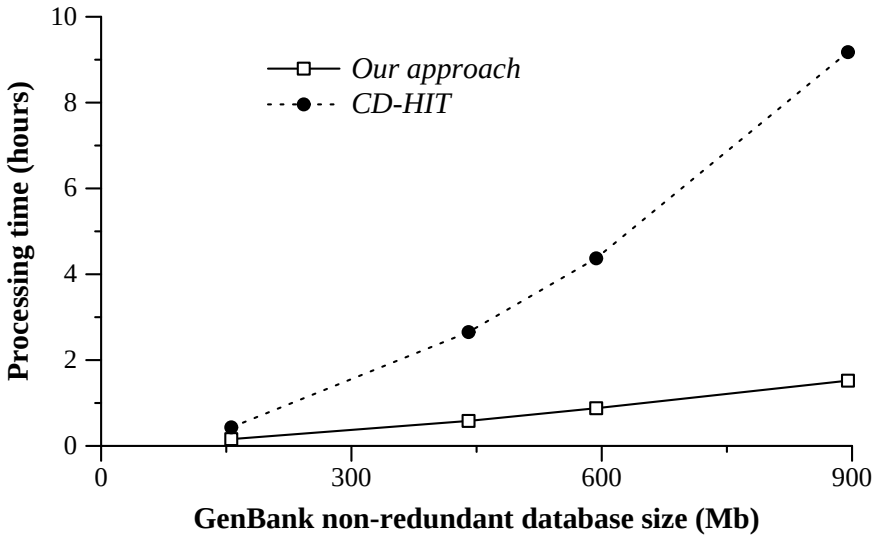

Figure 7.7: Time required to identify and remove redundant sequences from various releases of the GenBank NR database.

In the next section, we present a new approach to managing redundancy in genomic collections. Our scheme clusters highly-similar sequences into groups using the fingerprinting techniques described in this section. Each cluster is then represented using a special union-sequence that contains wildcard characters to represent all of the sequences in the cluster simultaneously.

7.2 A new approach for managing redundancy

The most commonly employed approach to reducing redundancy in genomic collections is to remove near-duplicate entries from the collection resulting in a smaller, representative version of the original database. Algorithms such as NRDB90 [Holm and Sander, 1998], RSDB [Park et al., 2000b], CD-HI [Li et al., 2001a] and CD-HIT [Li et al., 2001b] identify pairs of highly-similar sequences in the original collection and create a representative sequence database (RSDB) where no two sequences share more than a certain level of similarity.

Representative collections can be used to perform profile training in iterative search tools to reduce the chance of profile saturation and improve search accuracy [Park et al., 2000b; Li et al., 2002]. However, representative collections are not ideal for use with regular pairwise genomic search tools such as BLAST because they are not comprehensive. Searches against a RSDB are less accurate because a deleted sequence from the original collection may share greater similarity with the query than its retained representative. Further, search results are less authoritative because the user is only presented with a single alignment from a cluster of related sequences.

In this section, we propose a new method for managing redundancy that is suitable for search algorithms such as BLAST. Our method identifies clusters of related sequences using fingerprinting techniques described in the previous section. Each cluster is represented by a union-sequence that contains special wildcard characters to denote residue positions where the member sequences differ. During search, the query is initially compared to the union-sequence of each cluster, and the member sequences are only aligned to the query if the union-sequence produces a high-scoring alignment. Our method reduces overall search time because the vast majority of union-sequences do not produce a statistically significant alignment, resulting in fewer sequence comparisons. We have integrated our approach into our own implementation of BLAST and found that our method reduces search times against the GenBank database by 22% with no affect on search accuracy.

This section is organised as follows. First, we describe our new method for clustering

sequences through the use of wildcards and union-sequences in Section 7.2.1. We explain how clusters are derived using fingerprinting techniques in Section 7.2.2. In Section 7.2.3 we describe our methodology for scoring alignments between residues in the query and wildcards in a union-sequence during search. In Section 7.2.4 we present two approaches for selecting a good set of wildcards to be used for clustering. Finally, we analyse the effect of our clustering method on BLAST search times and present other experimental results in Section 7.2.5.

7.2.1 Clustering using wildcards

Let us define $E = e_1, ..., e_{|E|}$ as an ordered collection of sequences where each sequence is a string of residues drawn from the alphabet R, that is $e_i = r_1...r_{|e_i|}$. Our approach transforms the collection into a new representation by grouping sequences into clusters. The collection is therefore represented as a set of clusters C, where each cluster contains a union-sequence U and edit information for each member of the cluster. The union-sequence is a string of residues and special wildcard characters $U = u_1...u_{|U|} | u_j \in R \cup K$, where K is the set of characters used to represent the available wildcards.

Rather than use a single wildcard character to create the clusters, we use a set of n wildcards, W, where each wildcard has a different meaning and is represented by a distinct character in K. By default we use a set of size $n = 7$, for the reasons discussed in Section 7.2.4. Each wildcard is defined as a distinct set of residues, and can only be substituted for those residues. The ordered set of wildcards W used to perform the clustering is defined as $W = \{w_1, ..., w_n \mid w_x \subseteq R\}$. By convention, the last wildcard in this set w_n is assumed to be the *default wildcard* w_d that can represent any residue; that is, $w_n = w_d = R$. We present two different sets of wildcards for use in our experiments that are listed in Table 7.2 on page 211, and we discuss the selection of wildcards in Section 7.2.4.

Figure 7.8 shows an example cluster of heat shock proteins that was constructed using our clustering method. The union-sequence is shown at the top and cluster members are aligned below. Columns where the member sequences differ from each another and a wildcard has been inserted are shown in bold face. In this example, $W = \{w_d\}$ — that is, only the default wildcard is used and it is represented by an asterisk.

When a cluster is written to disk, the union-sequence — shown at the top of the figure — is stored in its complete form, and each member of the cluster is recorded using edit information. The edit information for each member sequence consists of start and end offsets that specify a range within the union-sequence, and a set of residues that replace the wildcards

```
          10        20        30        40          50          (position)
           |         |         |         |           |
KNQVAMN * QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV * QAEVDV * RFRSNT * ER  (union-seq)
        P QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV I QAEV                  (gi 156103)
KNQVAMN P QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV I QAEV                  (gi 156105)
          QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV V QAEVDV L RFRSNT K ER  (gi 156121)
KNQVAMN P QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV V QAEVDV L RFRSNT K    (gi 552059)
KNQVAMN P QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV I QAEVDV Q RFRSNT R    (gi 552055)
KNQVAMN P QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV I QAEVDV Q RFRSNT R E  (gi 552057)
        P QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV V QAEVDV L RFRSNT K ER  (gi 156098)
          QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV V QAEVDV L RFRS         (gi 156100)
             VFDAKRLIGRKFDEPTVQADMKHWPFKV I QAEVDV Q RFRSNT R E   (gi 156111)
        N QNTVFDAKRLIGRKFDEPTVQADMKHWPFKV I QAEVDV Q RFRSNT R    (gi 552056)
```

Figure 7.8: Example cluster of heat shock proteins from GenBank NR database. The union-sequence is shown above, followed by ten member sequences with GI accession numbers shown in brackets. Residue positions within the union-sequence are shown at the top.

in that range. For example, the first member of the cluster with GI accession 156103 would be represented by the tuple (8,44,PI); the member sequence can be reconstructed by copying the substring between positions 8 and 44 of the union-sequence and replacing the wildcards at union-sequence positions 8 and 40 with characters P and I respectively. Note that our clustering approach does not permit gaps; this is because insertions and deletions are heavily penalised during alignment, and our approach relies upon conformity between the alignment score produced by a cluster member and the alignment score produced by the corresponding representative union-sequence. If our scheme were to permit gaps, the union-sequence representing a cluster and the members of that cluster would produce greatly varying alignment scores, resulting in reduced search accuracy.

Our clustering method is designed so that each union-sequence aligns to the query with a score that is typically equal to or higher than the best score for aligning the query to members of the cluster. This relies upon an effective system for scoring matches between query residues and wildcards, which we discuss in detail in Section 7.2.3. During search, the query is compared to the union-sequence of each cluster, and if the union-sequence produces a statistically significant alignment, then the members of the cluster are restored from their compressed representations and aligned to the query. Our approach supports two modes of operation: users can choose to see all high-scoring alignments, or only the best alignment from each cluster. The latter mode reduces redundancy in the results.

7.2.2 Clustering algorithm

In this section we describe our approach to efficiently clustering large sequence collections. Our approach is based on the slotted SPEX fingerprinting algorithm described in Section 7.1, and has linear-time performance and low main-memory overheads.

For each chunk, the slotted SPEX algorithm outputs a *postings list* of sequences that contain the chunk and the offset into each sequence where the chunk occurs. Our clustering algorithm uses these lists to identify candidate pairs: pairs of sequences that share at least one chunk. Each pair of sequences is aligned using the similarity score measure we describe next — this measure is specific to our new clustering strategy and represents the number and type of wildcards that would be inserted into the new union-sequence representing that pair of sequences. This measure is designed to limit the number of wildcard characters that occur in union-sequences. Highly similar candidate pairs with a similarity score below threshold T are recorded.

Given the list of candidate pairs, we use a variation on single-linkage hierarchical clustering [Johnson, 1967] to identify clusters. Our clustering methods works as follows. First, each sequence is initially considered as a cluster with one member. Candidate pairs are then processed in increasing order of similarity score, from most- to least- similar, and the pair of clusters that contains the highly-similar candidate sequences are potentially merged.

In general, given two candidate clusters C_X and C_Y with union-sequences X and Y respectively, the following process is used to determine whether the clusters should be merged:

1. X and Y are aligned and the sequence space partitioned into a prefix, an overlap region, and a suffix.

2. The union sequence candidate U for the new cluster is created by replacing each mismatched residue in the overlap region with a suitable wildcard w.

3. The union-sequence candidate U is accepted if the mean alignment score increase $\bar{Q}$ in the overlap region is below a specified threshold T — this prevents union-sequences from containing too many wildcards and reducing search performance.

If the clusters are merged, a new cluster C_U is created consisting of all members of C_X and C_Y. This process is repeated for all candidate pairs. When inserting wildcards into the union-sequence, if more than one wildcard w from the set of wildcards W is suitable then the one with the lowest expected match score $e(w) = \sum_R s(w, r)p(r)$ is selected, where $p(r)$ is the background probability of residue r [Robinson and Robinson, 1991], and $s(w, r)$ is the

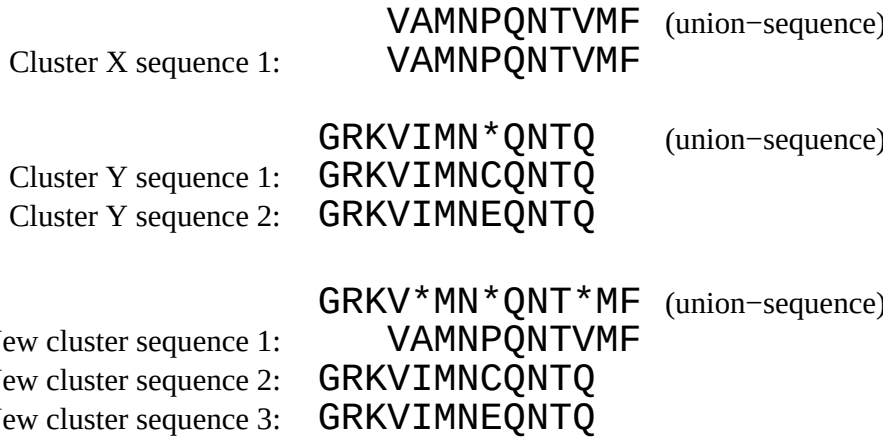

Figure 7.9: Merge of two example clusters. Cluster X contains a single sequence and cluster Y contains two sequences. A new cluster is created that contains members of both clusters and has a new union-sequence to represent all three member sequences.

alignment score for matching wildcard w to residue r. Note that we have not yet defined how wildcards are chosen; this is discussed further in Section 7.2.4. We discuss how the wildcard alignment vectors $s(w, \cdot)$ are calculated in Section 7.2.3.

The mean alignment score increase, Q, for a wildcard w is calculated as

$$Q(w) = \sum_{R} s(w, r)p(r) - \sum_{R \times R} s(r_1, r_2)p(r_1)p(r_2)$$

where $s(r_1, r_2)$ is the score for matching a pair of residues as defined by a scoring matrix such as BLOSUM62 [Henikoff and Henikoff, 1992]. $\sum_R s(w, r)p(r)$ represents the average score resulting from aligning a randomly selected query residue r to the wildcard w. $\sum_{R \times R} s(r_1, r_2)p(r_1)p(r_2)$ represents the average score for aligning a randomly selected query residue with a random collection residue. Therefore, the value of $Q(w)$ provides an estimate of the average increase in alignment score for a single residue position that one can expect when an arbitrary collection residue is replaced with the wildcard w.

Figure 7.9 illustrates the process of merging clusters. In this example, cluster X (which contains one sequence) and cluster Y (which contains two sequences) are merged. A new cluster containing the members of both X and Y is created, with a new union-sequence that contains wildcards at residue positions where the three sequences differ.

The above approach works extremely well for relatively small databases; however, as discussed in Section 7.1.5 some chunks appear frequently in larger collections resulting in

long postings lists. Unfortunately, the *deferred postings list expansion* technique described in Section 7.1.5 cannot be applied to our clustering method, which relies upon a complete, sorted list of candidate pairs for the clustering process. Instead, we have chosen to process frequently occurring chunks —those with more than M occurrences in the collection, where we use $M = 100$ by default for our clustering strategy—in a different, top-down manner before proceeding to the standard hierarchical clustering approach discussed previously.

The top-down approach identifies clusters from a list of sequences l that contain a frequently-occurring chunk as follows:

1. All sequences in l are loaded into main-memory and aligned with each other.
2. An exemplar sequence is selected; this is the sequence with a highest average percentage identity to the other sequences in l.
3. A new cluster C is created with the exemplar sequence as its first member.
4. Each sequence in l is compared to the union-sequence of the new cluster. Sequences where $\bar{Q} < T$ are added to the cluster in order from most- to least- similar using the approach we describe above.
5. All of the members of the new cluster C are removed from l and the process is repeated from step 1 until $|l| < M$.

The top-down clustering is illustrated in Figure 7.10. In this example, a list of five sequences that contain the chunk `RTMCS` is process using the top-down method. The sequence S_1 has the highest average percentage identity to the other sequences in l and is selected as the exemplar. A new cluster is created with S_1 as the first member, and sequences S_2 and S_4 are subsequently added. The three members of the new cluster are removed from l, and the process is repeated until $|l| < M$.

Once the postings lists has been processed using the top-down method the shortened list is processed using the hierarchical clustering method described above. While the top-down process is laborious, it is performed for fewer than 0.2% of postings lists when clustering the version of the GenBank non-redundant database described in Section 7.2.5 with default parameters.

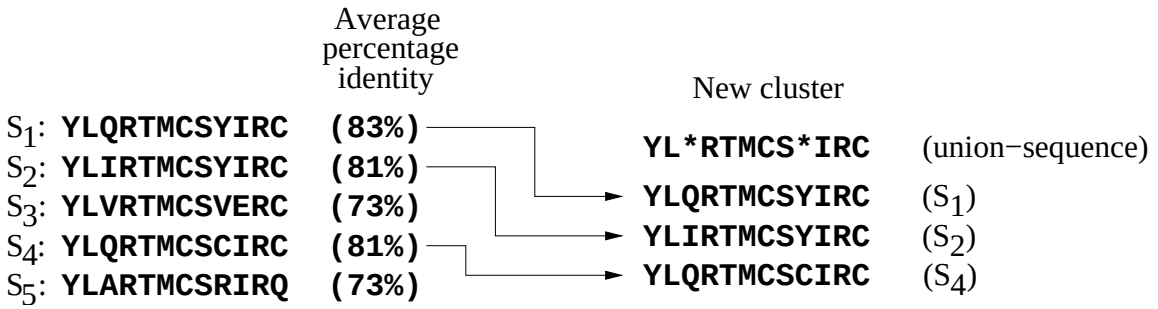

Figure 7.10: Illustration of top-down clustering where sequences $l = \{S_1, S_2, S_3, S_4, S_5\}$ contain the chunk `RTMCS`*. Each sequence is compared to every other sequence in the list and the sequence with the highest average percentage identity (S_1) is selected as the first member of a new cluster. Sequences S_2 and S_4 are highly similar to S_1 and are also included in the new cluster. The remaining sequences $l = \{S_3, S_5\}$ are used to perform another iteration of top-down clustering if $|l| \geq M$.*

7.2.3 Scoring wildcards

We have modified BLAST to work with our clustering algorithm as follows. Instead of comparing the query sequence to each member of the database, our approach compares the query only to the union-sequence representing each cluster, where the union-sequence may contain wildcard characters. If a high-scoring alignment between the union-sequence and query is identified, the members of the cluster are reconstructed and aligned to the query. In this section we discuss how, given a set of wildcards W, we determine the scoring vectors $s(w, \cdot)$ for each $w \in W$ that are used during search. The scoring vector constructed using our approach for one of the wildcard sets derived in Section 7.2.4 is shown in Figure 7.11.

Ideally, we would like the score between a query sequence Q and a union-sequence U to be precisely the highest score that would result from aligning Q against any of the sequences in cluster C_U. This would result in no loss in sensitivity and no false positives. Unfortunately, such a scoring scheme is not likely to be achievable without aligning against each sequence in every cluster, defeating much of the purpose of clustering in the first place.

To maintain the speed of our approach, scoring of wildcards against residues must be on the basis of a standard scoring vector $s(w, \cdot)$ and cannot take into consideration any data about the sequences represented by the cluster. Thus, scoring will involve a compromise between sensitivity (few false negatives) and speed (few false positives). We describe two such compromises below, and finally show how to combine them to achieve a good balance

Wildcard residues	Scoring vector																							
	L	A	G	S	V	E	T	K	D	P	I	R	N	Q	F	Y	M	H	C	W	B	Z	X	U
L,V,I,F,M	3	-1	-3	-2	3	-2	-1	-2	-3	-2	3	-2	-3	-1	2	0	4	-2	-1	-1	-3	-2	-1	-4
G,E,K,R,Q,H	-2	-1	1	0	-2	3	-1	4	0	-1	-3	3	0	3	-3	-1	-1	3	-3	-2	0	2	-1	-4
A,V,T,I,X	1	2	-1	0	3	-1	3	-1	-2	-1	3	-2	-2	-1	-1	-1	0	-2	-1	-3	-2	-1	-1	-4
S,E,T,K,D,N	-2	0	0	2	-2	2	2	2	4	-1	-2	0	4	1	-3	-2	-1	0	-2	-3	3	1	-1	-4
L,V,T,P,R,F,Y,M,H,C,W	1	-1	-2	-1	1	-1	1	-1	-2	2	0	1	-1	0	2	3	2	2	2	3	-2	-1	-1	-4
A,G,S,D,P,H	-2	3	3	3	-1	0	0	0	2	2	-2	-1	1	0	-2	-2	-1	1	-1	-3	1	0	0	-4
All residues	1	1	1	2	1	2	1	1	1	1	1	1	2	2	1	1	1	2	1	1	1	1	-1	-3

Figure 7.11: Scoring vectors for the minimised alignment score wildcards in Table 7.2. The set of residues represented by each wildcard is given in the left-hand column. The scoring vector provides an alignment score between each of the twenty-four amino acid symbols and that wildcard.

of sensitivity and speed.

During clustering, wildcards are inserted into the union-sequence to denote residue positions where the cluster members differ. Given a collection of union-sequences, let us order the wildcards that have been inserted into them during clustering by some arbitrary approach. Next, let us define $S = s_1...s_y \mid s_k \in W$ where S is the ordered sequence of y wildcards. Each occurrence of a wildcard s_k is used to represent a set of residues that appear in the cluster members at that position. We define $o_k \subseteq R$ as the set of residues represented by the k^{th} occurrence of a wildcard in the collection and $O = o_1...o_y \mid o_k \subseteq R$ as the ordered sequence of represented residue sets. That is, s_k is the k^{th} wildcard in the collection, which has been inserted into a union-sequence to replace the set of residues o_k in the cluster members. For example, if we consider the wildcard at union-sequence position 8 in Figure 7.8 on page 203 to be the first wildcard in the collection, then $s_1 = w_d$ (because the default wildcard has been inserted into the union-sequence) and $o_1 = \{P, N\}$. Since a wildcard can only be used to replace residues that it represents, s_k must be selected during clustering such that $o_k \subseteq s_k$.

Our first scoring scheme, s_{exp}, builds the scoring vector for each wildcard w by considering the actual occurrence pattern of residues represented by the wildcard w throughout the collection. Formally, we calculate the expected best score s_{exp} as:

$$s_{exp}(w, r) = \frac{\sum_{k \in P_i} \max_{f \in o_k} s(r, f)}{|P_i|}$$

where P_i is the set of ordinal numbers of all substitutions using the wildcard w_i:

$$P_i = \{j \mid j \ \in \mathbb{N}, j \leq y \ , \ s_j = w_i\}$$

This score can be interpreted as the mean score that would result from aligning residue r against the actual residues represented by the wildcard w. This score has the potential to reduce search accuracy; however, it distributes the scores well, and our experimental results in Section 7.2.5 indicate that it provides an excellent tradeoff between accuracy and speed.

The second scoring scheme, s_{opt}, calculates the optimistic alignment score of the wildcard w against each residue. The optimistic score is the highest score for aligning residue q to any of the residues represented by wildcard w. This is calculated as follows:

$$s_{opt}(w, r) = \max_{f \in w} s(r, f)$$

The optimistic score guarantees no loss in sensitivity: the score for aligning against a union-sequence U using this scoring scheme is at least as high as the score for any of the sequences represented by U. The problem is that in many cases the score for U is significantly higher, leading to false-positives where the union-sequence is flagged as a match despite none of the cluster members being sufficiently close to the query. This results in substantially slower search.

The expected and optimistic scoring schemes represent two different compromises between sensitivity and speed. We can adjust this balance by combining the two approaches using a mixture model. We define a mixture parameter, λ, such that $0 \leq \lambda \leq 1$. The mixture-model score for aligning wildcard w to residue r is defined as:

$$s_\lambda(w, r) = \lambda s_{opt}(w, r) + (1 - \lambda) s_{exp}(w, r)$$

The score $s_\lambda(w, r)$ for each w, r pair is calculated when the collection is being clustered and then recorded on disk in association with that collection. During a BLAST search, the wildcard scoring vectors are loaded from disk and used to perform the search. We report experiments with varying values of λ in Section 7.2.5. An example set of scoring vectors $s(w_i, \cdot)$ that were derived using our approach is shown in Figure 7.11.

7.2.4 Selecting wildcards

Having defined a system for assigning a scoring vector to an arbitrary wildcard, we now describe a method for selecting a set of wildcards to be used during the clustering process.

Each wildcard w is defined by a set of residues $w \subseteq R$ where each wildcard can only be used in place of the residues it represents when inserted into a union-sequence. Our wildcard scoring scheme that is described in Section 7.2.3 is dependent on the set of residues represented by w, so that each wildcard has a unique scoring vector. A set of wildcards, $W = \{w_1, ..., w_n\}$ is used during clustering. We assume that the last of these wildcards w_n is the default wildcard that can be used to represent any of the 24 residue and ambiguous codes, that is $w_n = R$. The remaining wildcards must be selected carefully; large residue sets can be used more frequently but provide poor discrimination with higher average alignment scores and more false positives. Similarly, small residue sets can be used less frequently, thereby increasing the use of larger residue sets such as the default wildcard.

The first aspect of choosing a set of wildcards to use for substitution is to decide on the size of this set. It would be ideal to use as many wildcards as necessary, so that for each substitution $s_i = o_i$. However, each wildcard must be encoded as a different character and this approach would lead to an very large alphabet. An enlarged alphabet would in turn lead to inefficiencies in BLAST due to larger lookup and scoring data structures. Thus, a compromise is required. BLAST uses a set of 20 character codes to represent residues, as well as four IUPAC-IUBMB ambiguous residue codes and the end-of-sequence sentinel code that was described in Section 3.1.3 on page 55, resulting in a total of 25 distinct codes. Each code is represented using 5 bits, permitting a total of 32 codes; this leaves 7 unused character codes. We have therefore chosen to use $|W| = 7$ wildcards.

We have investigated two different approaches to selecting a good set of wildcards. The first approach to the problem treats it as an optimisation scenario, and works as follows. We first cluster the collection as described in Section 7.2.2 using only the default wildcard, $W = \{w_d\}$. We use the residue-substitution sequence O from this clustering to create a set W^* of candidate wildcards. Our goal can then be defined as follows: we wish to select the set of wildcards $W \subseteq W^*$ such that the total average alignment score $A = \sum_{w \in S} \sum_{r \in R} s(w, r)p(r)$ for all substitutions S is minimised. A lower A implies a reduction in the number of high-scoring matches between a typical query sequence and union-sequences in the collection, thereby reducing the number of false-positives in which cluster members are fruitlessly recreated and aligned to the query.

In selecting the wildcard set W that minimises A we use the following greedy approach: first, we initialize W to contain only the default wildcard w_d. We then scan through W^* and select the wildcard that leads to the greatest overall reduction in A. This process is

Minimum alignment score	Physico-chemical classifications	
L,V,I,F,M	L,V,I	(aliphatic)
G,E,K,R,Q,H	F,Y,H,W	(aromatic)
A,V,T,I,X	E,K,D,R,H	(charged)
S,E,T,K,D,N	L,A,G,V,K,I,F,Y,M,H,C,W	(hydrophobic)
L,V,T,P,R,F,Y,M,H,C,W	S,E,T,K,D,R,N,Q,Y,H,C,W	(polar)
A,G,S,D,P,H	A,G,S,V,T,D,P,N,C	(small)
All residues	All residues	(default wildcard)

Table 7.2: Two different sets of wildcards to be used for clustering. Each list is sorted in order from lowest to highest average alignment score, A, and contains seven entries including the default wildcard. The left-hand list is selected to minimise the average alignment score, A, using a hill-climbing strategy, and the right-hand list is based on the amino acid classifications described in Taylor [1986]

repeated until the set W is filled, at each iteration considering the wildcards already in W in the calculation of A. Once W is full we employ a hill-climbing strategy where we consider replacing each wildcard with a set of residues from W^* with the aim of further reducing A.

A set of wildcards was chosen by applying this strategy to the GenBank NR database described in Section 7.2.5. The left-hand column of Table 7.2 lists the wildcards that were identified using this approach and used by default for reported experiments in this chapter.

We also consider defining wildcards based on groups of amino acids with similar physico-chemical properties. We used the amino acid classifications described in Taylor [1986] to define the set of seven wildcards shown in the right-hand column of Table 7.2. In addition to the default wildcard, six wildcards were defined to represent the aliphatic, aromatic, charged, hydrophobic, polar and small classes of amino acids. We present experimental results for this alternative set of wildcards in the following section.

7.2.5 Results

In this section we analyse the effect of our clustering strategy on collection size and search times. For our assessments, we used version 1.65 of the ASTRAL Compendium [Chandonia et al., 2004] that uses information from the SCOP database [Murzin et al., 1995; Andreeva et al., 2004] to classify sequences with fold, superfamily, and family information. The data-

base contains a total of 67,210 sequences classified into 1,538 superfamilies.

A set of 8,759 test queries were extracted from the ASTRAL database such that no two queries share more than 90% identity. To measure search accuracy, each query was searched against the ASTRAL database and the commonly-used Receiver Operating Characteristic (ROC) score was used [Gribskov and Robinson, 1996] following the approach described in Section 3.3.3 on page 97. A match between two sequences is considered positive if they are from the same superfamily, otherwise it is considered negative. The ROC_{50} score provides a measure between 0 and 1, where a higher score represents better sensitivity (detection of true positives) and selectivity (ranking true positives ahead of false positives).

The SCOP database is too small to provide an accurate measure of search time, so we use the GenBank non-redundant (NR) protein database to measure search speed. The GenBank collection was downloaded August 18, 2005 and contains 2,739,666 sequences in around 900 megabytes of sequence data. Performance was measured using 50 queries randomly selected from GenBank NR. Each query was searched against the entire collection three times with the best runtime recorded and the results averaged. Experiments were conducted on a Pentium 4 2.8GHz machine with two gigabytes of main memory.

We used FSA-BLAST—our own version of BLAST—with default parameters as a baseline. To assess the clustering scheme, the GenBank and ASTRAL databases were clustered and FSA-BLAST was configured to report all high-scoring alignments, rather than only the best alignment from each cluster. All reported collection sizes include sequence data and edit information but exclude sequence descriptions. CD-HIT version 2.0.4 beta was used for experiments with 90% clustering threshold and maximum memory set to 1.5 Gb. We also report results for NCBI-BLAST version 2.2.11 and our own implementation of Smith-Waterman that uses the exact same scoring functions and statistics as BLAST [Karlin and Altschul, 1990; Altschul and Gish, 1996]. The Smith-Waterman results represent the highest possible degree of sensitivity that could be achieved by BLAST and provides a meaningful reference point. No sequence filtering was performed for our experiments in this chapter.

The overall results for our clustering method are shown in Table 7.3. When used with default settings of $\lambda = 0.2$ and $T = 0.25$, and the set of wildcards selected to minimise alignment score in Table 7.2, our clustering approach reduces the overall size of the NR database by 27% and improves search times by 22%. Importantly, the ROC score indicates that there is no significant effect on search accuracy, with the highly redundant SCOP database reducing in size by 80% when clustered. If users are willing to accept a small loss in accuracy, then the parameters $\lambda = 0$ and $T = 0.3$ improve search times by 27% and reduce the size of

Scheme	GenBank NR		ASTRAL
	Time secs (% baseline)	Sequence data Mb (% baseline)	ROC_{50}
FSA-BLAST			
No clustering (baseline)	28.75 (100%)	900 (100%)	0.398
Cluster $\lambda = 0.2, T = 0.25$	22.54 (78%)	655 (73%)	0.398
Cluster $\lambda = 0, T = 0.3$	20.97 (73%)	650 (72%)	0.397
NCBI-BLAST	45.75 (159%)	898 (100%)	0.398
Smith-Waterman	—	—	0.415

Table 7.3: Average runtime for 50 queries searched against the GenBank NR database, and SCOP ROC_{50} scores for the ASTRAL collection.

the sequence collection by 28% with a decrease of 0.001 in ROC score when compared to our baseline. Since we are interested in improving performance with no loss in accuracy we do not consider these non-default settings further. Overall, our clustering approach with default parameters combined with improvements to the gapped alignment and hit detection stages of BLAST desribed in Chapters 4 and 5 more than double the speed of FSA-BLAST protein searches compared to NCBI-BLAST with no significant effect on accuracy. Both versions of BLAST produce ROC scores 0.017 below the optimal Smith-Waterman algorithm.

The results in Table 7.3 also show that our scheme is an effective means of compressing protein sequences, a task that has been deemed difficult by previous researchers [Nevill-Manning and Witten, 1999; Weiss et al., 2000]. Assuming a uniform, independent distribution of amino acids, protein sequence data can be represented with 4.322 bits per symbol [Nevill-Manning and Witten, 1999]. Our clustering scheme is able to reduce the space required to store protein sequence data in the GenBank non-redundant database to around 3.15 bits per symbol; to our knowledge, this is significantly less than the current best compression rate of 4.051 bits per symbol [Nevill-Manning and Witten, 1999].

The results presented in Table 7.3 also show a 37% reduction in search time for our FSA-BLAST baseline, without clustering, when compared to the NCBI-BLAST implementation. This compares with a 32% reduction in runtime observed for an Intel workstation in Section 5.3 for our FSA-BLAST implementation that employs the new gapped alignment algorithms and the optimised DFA. The small variation in speed gain observed between these two experiments

Wildcard set	GenBank NR		ASTRAL
	Time	Sequence data	
	secs (% baseline)	Mb (% baseline)	ROC_{50}
Minimum alignment score	22.54 (78%)	655 (73%)	0.398
Physico-chemical classifications	23.25 (81%)	656 (73%)	0.398
Default wildcard only	23.49 (82%)	663 (76%)	0.398

Table 7.4: Average runtime and SCOP ROC_{50} scores for varying sets of wildcards. The first two rows contain results for the wildcard sets defined in Table 7.2. The third row contains results for clustering with only the default wildcard $W = \{w_d\}$

is likely due to changes in the test collection; our earlier experiments were conducted on the smaller June 2004 release of GenBank, whilst the August 2005 release of GenBank was used for experiments in this chapter. This suggestions that our novel protein alignment methods scale at least linear with collection size, and will continue to provide a significant improvement to search times as collections continue to grow.

In Table 7.4 we compare search accuracy and performance for varying wildcard sets. The set of wildcards that were selected to minimise the average alignment score using the approach described in Section 7.2.4 provide the fastest search times and smallest collection size. The set of wildcards based on the physico-chemical classifications of Taylor [1986] do not perform as well, with 3% slower search times. Finally, search performance was worse still when the collections were clustered using only the default wildcard; this supports our approach of using multiple wildcards to construct clusters.

Figure 7.12 shows a comparison of clustering times between CD-HIT and our novel clustering approach that uses union-sequences and wildcards for four different releases of the GenBank NR database; details of the collections used are given in Table 7.5. The results show that the clustering time of our approach is roughly linear with the collection size and the CD-HIT approach is superlinear (Figure 7.12). On the recent GenBank non-redundant collection, CD-HIT is around 9 times slower than our approach and we expect this ratio to further increase with collection size. The performance of our clustering system is similar to the results presented in Figure 7.7 on page 200 for our RSDB system; this is to be expected given that the slotted SPEX algorithm described in Section 7.1.1 is used to identify pairs of similar sequences in both cases.

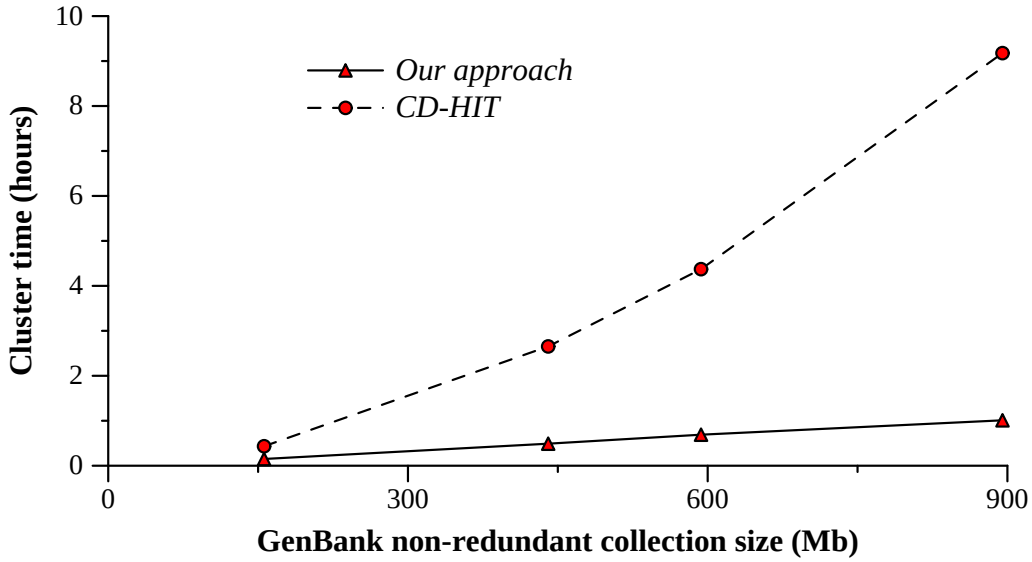

Figure 7.12: Clustering performance for GenBank NR databases of varying sizes.

Release date	Number of sequences	Collection Size (Mb)	Overall size reduction (Mb)	Percentage of collection
16 July 2000	521,662	157	45	28.9%
22 May 2003	1,436,591	443	124	28.1%
30 June 2004	1,873,745	597	165	27.4%
18 August 2005	2,739,666	900	245	27.3%

Table 7.5: Redundancy in GenBank NR database over time.

Table 7.5 shows the amount of redundancy in the GenBank NR database as it has grown over time, measured using our clustering approach. We observe that redundancy is increasing at a rate roughly proportional to collection size with the percentage reduction through clustering remaining almost constant at 27%–29% across versions of the collection tested. This suggests that redundancy will continue to plague genomic data banks as they grow further in size.

Figure 7.13 shows the effect on accuracy for varying values of λ and T. We have chosen $\lambda = 0.2$ as a default value because smaller values of λ result in a larger decrease in search accuracy, and larger values reduce search speed. We observe that for $\lambda = 0.2$ there is little variation in search accuracy for values of T between 0.05 and 0.3.

Figure 7.14 shows the effect on search times for varying values of T where $\lambda = 0.2$. As T

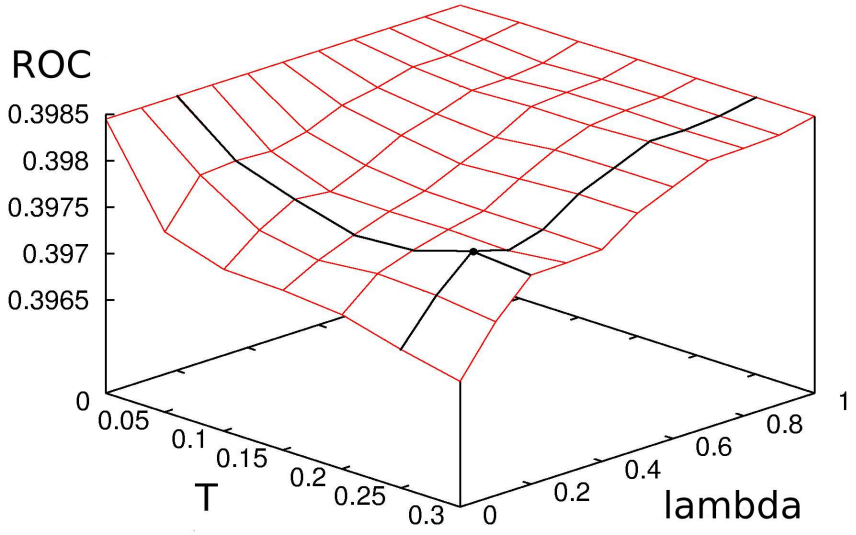

Figure 7.13: Search accuracy for collections clustered with varying values of λ and T. Default values of $\lambda = 0.2, T = 0.25$ are highlighted.

increases the clustered collection becomes smaller, leading to faster search times. However, if T is too large then union-sequences with a high percentage of wildcards are permitted, leading to an increase in the number of cluster members that are recreated and a corresponding reduction in search speed. We have chosen the value $T = 0.25$ that maximises search speed.

Figure 7.15 shows the distribution of cluster sizes for GenBank, which appears to closely follow a power-law distribution. Around 55% of clusters contain just two members, and the largest cluster contains 488 members. Of the ten largest clusters identified by our approach, five relate to human immunodeficiency virus proteins, three relate to cytochrome b, one relates to elongation factor 1α, and one relates to cytochrome oxidase subunit I. This supports our previous observation that cluster size is proportional to the interest in a research area.

7.3 Conclusion

Sequence databanks such as GenBank contain a large number of near-identical entries. Such internal redundancy has several negative effects including larger collection size, slower search times, and difficult-to-interpret results. Redundancy within a collection can lead to over-representation of alignments within particular protein domains, distracting the user from other potentially important hits. In this chapter we describe two improvements to existing

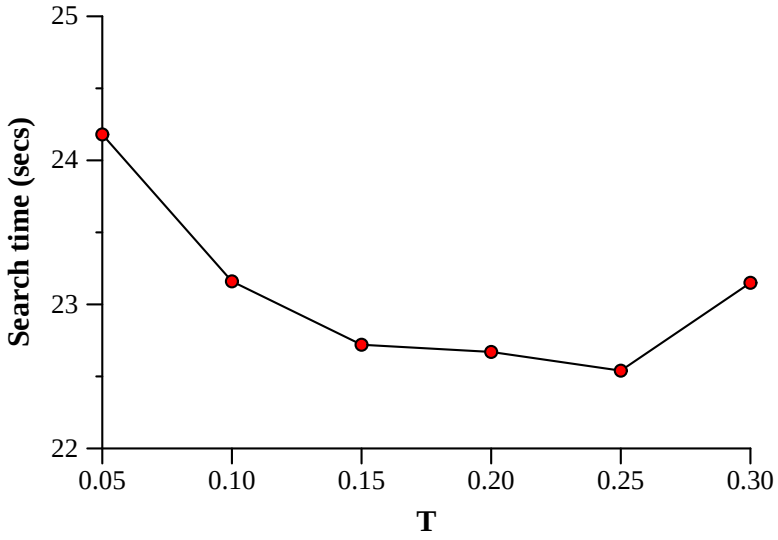

Figure 7.14: Average BLAST *search time using* $\lambda = 0.2$ *and varying values of* T.

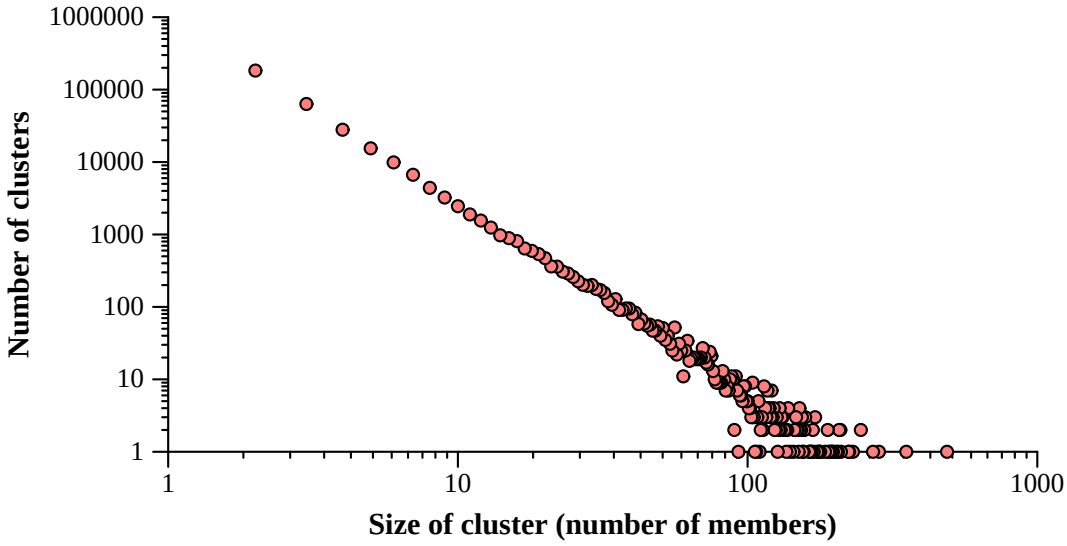

Figure 7.15: Distribution of sizes for clusters identified in the GenBank NR database.

methods for managing redundancy.

Our first contribution is a new system for identifying pairs of similar sequences in a large collection that uses fingerprinting [Manber, 1994; Brin et al., 1995; Heintze, 1996; Broder et al., 1997; Bernstein and Zobel, 2004]: a technique that has been used for grouping highly similar documents in extremely large collections. Fingerprinting operates by selecting fixed-length subsequences—known as *chunks*—from each document. This set of chunks is known as the document *fingerprint* and acts as a compact surrogate for the document. As highly similar documents are expected to share a large number of chunks, fingerprints are used to efficiently detect similar documents in a collection.

Fingerprinting has been successfully applied to text and web data for several applications including plagiarism detection, copyright protection, and search-engine optimisation. We have adapted existing fingerprinting methods to biological sequence data with our slotted SPEX algorithm. Our approach is suitable for processing genomic data that has different characteristics to English text; for example, genomic sequences do not contain natural word-delimiters such as punctuation and whitespace. Our approach can identify pairs of similar sequences in the entire GenBank NR protein database in around 1.5 hours on a standard workstation and appears to scale linearly in the size of the collection. This is a major improvement on the existing best approach, CD-HIT [Li et al., 2001b], that performs an all-against-all comparison, requires over 9 hours for the same task and scales superlinearly. Further, there is no notable change in accuracy.

Our second contribution is a novel approach for representing and managing redundancy in the collection. Instead of discarding near-duplicate sequences, our approach identifies clusters of highly-similar sequences and constructs a special union-sequence that represents all members of the cluster simultaneously through the use of special wildcard characters. We present a new approach for searching clusters that, when combined with a well-chosen set of wildcards and a system for scoring matches between wildcards and query residues, leads to faster search times without a loss in accuracy. Moreover, by recording the differences between the union-sequence and each cluster member using edit information our approach compresses the collection. Our scheme is general and can be adapted to most homology search tools.

We have integrated our clustering strategy into FSA-BLAST, our own implementation of BLAST that incorporates improvements to the algorithm described in previous chapters. Our experimental results for protein data show that our clustering scheme reduces BLAST search times against the GenBank non-redundant database by 22% and compresses sequence data by 27% with no significant reduction in overall search accuracy. We propose that pre-clustered

copies of the GenBank collection are made publicly available for download, resulting in faster search and collection download times. Overall, our new approach for managing redundancy, coupled with improvements to the gapped alignment and hit detection stages of BLAST that were described in previous chapters more than halve the average protein query evaluation time. Similarly, our improvements to nucleotide search described in Chapter 6 roughly halve average nucleotide query evaluation times.

In the next chapter, we describe possible extensions to our work and present our conclusions.

Chapter 8

Conclusion and Future Work

In this chapter, we discuss possible extensions to the work presented in this thesis and provide some concluding remarks. In Section 8.1 we proposed several new research directions. First, we comment on the application of our novel search strategies to iterative search tools such as PSI-BLAST [Altschul et al., 1997]. We also discuss methods for faster iterative search that reuse data between iterations. We consider applying the fast methods for sequence comparison that are embodied in BLAST to index-based approaches, and propose that our novel strategies for managing redundancy be applied to nucleotide data. We also consider employing our slotted SPEX algorithm to identify duplicates in text and web collections, and suggest that tighter coupling of the BLAST search stages be investigated. Finally, we present a summary of our research and our conclusions in Section 8.2.

8.1 Future Work

Iterative search

Iterative search algorithms such as PSI-BLAST [Altschul et al., 1997] and SAM-T98 [Karplus et al., 1998] use a recursive feedback approach that is similar to query expansion in information retrieval [Billerbeck and Zobel, 2004] and results in highly sensitive search of protein collections [Park et al., 1998; Chen, 2003]. The PSI-BLAST algorithm employs BLAST in an iterative fashion as follows. First, the collection is searched with a user-specified query sequence and alignments with an E-value below the default cutoff of $e = 10$ are displayed to the user[1]. High-scoring alignments with an E-value below the threshold t, where $t = 0.002$ by default,

[1]Default parameters for NCBI-BLAST version 2.2.12

are used to construct a profile or Position Specific Scoring Matrix (PSSM), which represents sequences in the collection that are highly similar to the query. In the second iteration, the collection is searched with the newly-constructed profile and the resulting high-scoring alignments are used to update the profile. This process is repeated until the maximum number of iterations is reached or the process converges, that is, no new high-scoring alignments are identified. Each iteration of PSI-BLAST takes roughly as long as a single BLAST search [Altschul et al., 1997], so that the overall iterative process is substantially slower than regular pairwise approaches such as BLAST.

Our innovations relating to BLAST could also be applied to the iterative PSI-BLAST approach. We expect that our improvements would yield similar speed gains when applied to PSI-BLAST, since the underlying BLAST search process, rather than the profile construction stages, appears to consume the vast majority of the overall search time [Altschul et al., 1997]. One interesting extension of our work would involve developing our own version of the PSI-BLAST tool based on our existing FSA-BLAST implementation that incorporates our innovations to BLAST. In addition to providing faster iterative search, this would permit a more detailed analysis of the PSI-BLAST approach and the performance characteristics of this tool. Further, we believe that several speed-related optimisations to PSI-BLAST are possible, as we describe next. Faster PSI-BLAST searches would be of enormous benefit to the biological community; the iteration approach is highly sensitive but often infeasible for large batches of queries because it is significantly slower than conventional pairwise methods.

We have considered two speed-related optimisations to PSI-BLAST. Our first proposed optimisation reduces the number of passes through the entire collection by introducing intermediate iterations where statistically significant collection sequences are realigned to the updated profile before another complete scan of the collection is performed. In our own preliminary study of PSI-BLAST[2] with the GenBank NR database and 50 randomly selected queries, we found that around 28% of collection sequences initially incorporated into the profile at iteration n also produced a statistically significant alignment at a prior iteration when $n > 1$. That is, collection sequences with an E-value below the default threshold for profile inclusion of $t = 0.002$ were often already reported in a previous iteration with an E-value between the default cutoff of $e = 10$ and the default threshold for inclusion of $t = 0.002$. This suggests that the iterative process could be hastened by inserting intermediate iterations, where all reported alignments are rescored and possibly added to the profile, before

[2]NCBI `blastpgp` tool version 2.2.10

another complete scan of the collection is performed. As a result, more sequences would be incorporated into the profile sooner, thereby reducing the number of computationally costly searches of the collection before the search converges.

Our second proposed optimisation to the PSI-BLAST approach involves reusing data from individual BLAST stages between iterations for faster search. We expect PSI-BLAST profiles to be highly similar from one iteration to the next, as only a handful of alignments are typically added to the profile at each iteration. In our preliminary study of PSI-BLAST described previously, we found that 86% of sequences incorporated into the profile at iteration n were also incorporated in the previous iteration when $n > 1$. Therefore, the results from each stage of search, such as the location of hits, high-scoring ungapped extensions and high-scoring gapped extensions, are unlikely to vary significantly between iterations. Faster PSI-BLAST searches may be achieved by storing and reusing the results from each stage of BLAST in subsequent iterations and considering only the effect of changes in the search profile on alignment scores. This method of information reuse would be most effective when changes to the profile are small, and may lead to a substantial reduction in overall PSI-BLAST search times.

The PSI-BLAST algorithm offers highly sensitive search and is able to detect roughly three times as many remote homologies as basic pairwise approaches such as BLAST [Park et al., 1998]. However, despite the success of profile-based methods, several problems continue to plague iterative search schemes such as PSI-BLAST. Two highly detrimental problems are profile contamination, where false positive sequences are incorporated into the profile, and profile saturation, where over-represented protein families dominate the profile so that it is no longer general enough to detect distant relationships. Park et al. [2000b] and Li et al. [2002] describe a solution to the latter problem, where a representative database that has fewer redundant entries is used to perform profile training in the earlier iterations of search, before a complete database such as the PDB or SCOP is searched in the final iteration. Previous research has shown that this approach drastically improves profile quality, leads to more accurate search, and reduces search times [Park et al., 2000b; Li et al., 2002]. Unfortunately, existing representative databases are unsuitable for the final iteration of search because they are not sufficiently comprehensive. Further, the two-database approach proposed by Park et al. [2000b] is not practical for searching large collections such as GenBank because the original and the representative collections must both reside in main-memory for reasonable search performance.

In Chapter 7, we described a new approach for managing redundancy that identifies

clusters of high-similar sequences and represents all members of a cluster simultaneously using a special union-sequence. We propose that our new clustering scheme for managing redundancy be applied to iterative search algorithms such as PSI-BLAST. When applied to regular pairwise methods such as BLAST, our clustering strategy supports two modes of operation; either the single highest-scoring alignment only from a cluster is reported, or alternatively all high-scoring alignments from a cluster are displayed. The former mode reduces redundancy in search results and could be applied to earlier iterations of PSI-BLAST to remove redundant entries from the PSSM and reduce profile saturation. In the final iteration of search, the latter mode could be employed to identify all statistically significant alignments and provide the user with a comprehensive set of results. We would expect this strategy to yield similar improvements in search accuracy to those reported by Li et al. [2002] without the need to maintain two versions of the target database in main-memory. Further, by removing redundant entries at search time our approach ensures that the closest matching sequence from each cluster is incorporated into the profile.

Despite the success of search tools such as PSI-BLAST and SAM-T98, iterative search algorithms remain poorly understood and we believe that further refinements to existing approaches are possible. Many of the design aspects of iterative tools appear ad-hoc and warrant scrutiny. For example, PSI-BLAST iterates until convergence and employs a fixed threshold for profile inclusion whereas SAM-T98, which uses Hidden Markov Models (HMMs) instead of PSSMs, performs four iterations and varies the inclusion threshold between iterations; it is unclear why different strategies have been employed for each tool. Further, in an initial study where we compared the sensitivity of iterative homology search tools we found that for a search with the same query and target database, PSI-BLAST and SAM-T98 frequently report greatly varying sets of alignments, despite both approaches being highly sensitive to homologous relationships [Park et al., 1998]. This poor agreement suggests that both tools fail to identify homologous relationships and that iterative search has considerable room for improvement. A search strategy that incorporates both PSSMs and HMMs certainly warrants investigation.

Index-based approaches

Several index-based methods for homology search have been proposed, where an inverted index of the collection is used to identify hits without exhaustively scanning the entire database. Schemes such as CAFE [Williams and Zobel, 1996; 2002], SCAN [Orcutt and Barker,

1984], FLASH [Califano and Rigoutsos, 1993], RAMDB [Fondrat and Dessen, 1995], and RAPID [Miller et al., 1999] employ an on-disk index and are suitable for searching large collections such as GenBank. CAFE is the most successful approach to employ an on-disk index with substantially faster search times and a small reduction in search accuracy when compared to BLAST [Chen, 2004]. In contrast, BLAT [Kent, 2002], PATTERNHUNTER [Ma et al., 2002; Li et al., 2004] and SSAHA [Ning et al., 2001] employ an index that resides in main-memory and offers staggeringly fast search times, but is unsuitable for searching large collections such as GenBank.

We propose combining the sensitive BLAST approach with fast on-disk index based schemes such as CAFE. One of the main drawbacks with the existing CAFE approach is that it employs inefficient methods for aligning collections sequences in the later stages of search. We expect that by incorporating the ungapped and gapped extension methods used by BLAST to align collection sequences, the search performance and accuracy of CAFE could be further improved. A variation of the BLAST algorithm that employs an in-memory index is also worth investigating. Although in-memory indices are unsuitable for processing large collections such as GenBank, such a scheme may serve well if used for coarse-grain distributed searches where the collection is divided amongst several processors and each node processes only a fraction of the entire collection [McGinnis and Madden, 2004].

Existing research and our own preliminary investigations suggest that index-based approaches are generally faster but less sensitive than exhaustive search schemes such as BLAST [Chen, 2004; Kent, 2002]. Further, index-based approaches are substantially faster for highly insensitive searches where a long word length is employed because search times are dependent on the number of hits rather than the size of the collection when an index is employed [Ning et al., 2001]. Therefore, we propose a two-stage approach to search that combines an index-based scheme such as CAFE with an exhaustive search method such as BLAST. Our proposed scheme works as follows. First, a fast but insensitive search is performed using an inverted index. If at least v statistically significant alignments are identified, where v is the maximum number of alignments reported to the user, then the search terminates. Otherwise, a more sensitive, slower exhaustive search is performed to identify more distantly-related homologs. Since not all queries must be processed using the slower exhaustive approach, average search times would be reduced. This two-stage approach is an automated variation of the manual procedure proposed by McGinnis and Madden [2004], who suggest that users "first find the very obvious similarities with a fast algorithm [and then] use more sensitive algorithms on the [query] sequences that did not have strong matches in the earlier step".

Managing redundancy in nucleotide collections

In Chapter 7 we described novel methods for managing redundancy in genomic collections. We confined our investigation to protein data, however the application of our methods to nucleotide sequences is certainly worth investigating. We expect to find a higher degree of duplication in nucleotide collections with large batches of expressed sequence tag sequences and parts of entire genomes prevalent in databases such as GenBank. Altschul et al. [1994] and Rapp and Wheeler [2005] both comment on the high degree of duplication in nucleotide collections due to the presence of thousands of overlapping sequence fragments from rapid sequencing techniques.

We do not believe our methods are trivially adapted to nucleotide data: collections such as the GenBank NR nucleotide database are larger than their protein counterparts, and DNA sequences exhibit different characteristics to protein sequences with a smaller alphabet size and longer average sequence length. Our fingerprinting strategy may require further tuning before it is successfully applied to nucleotide collections. Further, it is unclear how our clustering strategy could be applied to DNA sequences that are compressed using the byte packed scheme. Without decoding ambiguous codes (which is time consuming and currently only performed for a fraction of collection sequences in the final stage of BLAST) the compression scheme only supports an alphabet of size four that leaves no unassigned codes for the extra wildcard characters embedded in union-sequences. This problem would need to be addressed for our clustering strategy to be applied to nucleotide data.

Other possible extensions

We described a new fingerprinting algorithm in Chapter 7 that is based on the lossless SPEX approach by Bernstein and Zobel [2004]. Our slotted SPEX algorithm is suitable for genomic sequence data that must be processed with a higher granularity than English text, which does not contain natural delimiters such as whitespace. Slotted SPEX reduces the number of chunks inserted into the hashtable but ensures synchronisation between the chunks processed from similar sequences. As a results, fewer entries are made into the hashtable, less main-memory is required, each iteration is faster and fewer iterations are required. Although slotted SPEX was designed specifically to process sequence data, we propose that our novel algorithm also be applied to English text; we expect that similar benefits would also result when text documents and web data is processed. Further, slotted SPEX is likely to perform well at processing text in other languages such as Chinese that lack delimiters such as whitespace.

Finally, we propose that tighter coupling between the individual stages of BLAST be investigated. For example, information about the location of hits could be used for faster ungapped alignment by avoiding the need to realign these matching regions. Similarity, the location of high-scoring ungapped alignments could be used for faster and more sensitive gapped alignment, perhaps using a scheme that links the ungapped regions and only employs dynamic programming for joining these already aligned regions. The authors of PATTERNHUNTER [Ma et al., 2002] describe a scheme where short, matching regions between the query and a collection sequence are identified and then joined together to construct a local alignment; a more detailed evaluation of this approach in the context of exhaustive search strategies such as BLAST would certainly be worthwhile.

8.2 Conclusion

Our aim in this research project has been to investigate new methods for fast, accurate homology search. Continuing the evolution of search algorithms such as BLAST, by improving its algorithms and optimisations, is essential to reduce query evaluation times in the face of exponentially-increasing collection sizes. However, we wish to retain the high degree of sensitivity offered by the BLAST approach.

We have successfully met our aim with a wide range of innovations that afford faster search of genomic collections. We have described new methods for sequence alignment, new approaches for identifying hits, faster methods for comparing nucleotide sequences, and novel strategies for managing redundancy. The result of our improvements is a two-fold speed improvement to the BLAST algorithm for both protein and nucleotide searches. Importantly, none of our schemes have a detrimental effect on search accuracy.

To motivate the need for fast yet accurate homology search methods, we described genomic sequence data and existing methods for comparing sequences and searching data banks in Chapters 2 and 3. We began Chapter 2 with an overview of genomics and proteomics: we explained how sequence data is derived from proteins and DNA, and the roles that these molecules play in living organisms. We also described popular biological sequence repositories such as GenBank [Benson et al., 2005] and highlighted the exponential growth in sequence data over the past two decades. Next, we discussed techniques for comparing biological sequences through a sequence alignment. We presented methods for global [Sellers, 1974] and local [Smith and Waterman, 1981] alignment that support linear and affine gap costs [Gotoh, 1982]. We also described linear space methods for recording the optimal alignment through

traceback [Hirschberg, 1975] and methods for recording all high-scoring alignments, rather than only the best alignment, between a pair of sequences.

Pairwise sequence alignments are computationally expensive in time as well as space, and we presented two heuristics in Chapter 2 that reduce the time required to align sequences with minimal impact on accuracy. The first is the *banded alignment* strategy [Chao et al., 1992] that is employed in the FASTA homology search tool and the second is the *dropoff alignment* method [Zhang et al., 1998a] employed by BLAST. Finally, we explained how data mutation matrices such as BLOSUM [Henikoff and Henikoff, 1992] and PAM [Dayhoff et al., 1978] are constructed and used to score protein sequence alignments.

Genomic search methods have a long history with early approaches such as FASTP [Pearson and Lipman, 1985] dating back to more than two decades ago. In Chapter 3 we surveyed the most successful approaches to homology search. We began by describing the popular FASTA [Pearson and Lipman, 1988] and BLAST [Altschul et al., 1990; 1997] exhaustive pairwise methods. We presented a detailed description of the BLAST algorithm based on original research papers describing the approach [Altschul et al., 1990; 1997] and our own analysis of the NCBI-BLAST source code. Our description of BLAST incorporated a study of the performance characteristics of each stage that provided us with a valuable analysis of the filtering mechanisms that it employs. We also provided an overview of BLAST usage based on our analysis of query data that were provided to us by researchers at the NCBI.

In Chapter 3 we also described a range of alternative approaches to homology search. We surveyed approaches for identifying hits efficiently that employ an index of the collection, similar to those commonly used for text retrieval in search engines such as Google[3]. Index-based search tools such as CAFE [Williams and Zobel, 1996; 2002], BLAT [Kent, 2002] and PATTERNHUNTER [Ma et al., 2002; Li et al., 2004] avoid scanning the entire collection and are considerably faster than exhaustive approaches such as FASTA and BLAST. Unfortunately, these methods are less sensitive, or are unsuitable for processing large collections. We also considered distributed approaches to search, whereby the task is divided amongst a cluster of processors to reduce query evaluation times, and iterative search methods such as PSI-BLAST [Altschul et al., 1997] that employ profiles for highly sensitive but time consuming search.

We considered a range of issues relating to genomic search in Chapter 3. We discussed methods for assessing the statistical significance of alignments, based on the observation that optimal pairwise alignment scores for random or unrelated sequences follow an extreme

[3]See: `http://www.google.com/`

value distribution. We described in detail the methods by which BLAST calculates an *E*-value for each alignment that represents the likelihood the alignment score was due to a chance similarity. We also discussed the effect of low-complexity sequence regions on the quality of search results, and surveyed existing approaches for identifying and managing such compositional bias. Next, we presented methods for assessing the accuracy of homology search tools that employ protein classification databases such as SCOP [Murzin et al., 1995; Andreeva et al., 2004]. Finally, we discussed the detrimental effects of internal redundancy in genomic collections on search and surveyed existing approaches for identifying and managing near-duplicate sequences.

The gapped alignment stages of BLAST consume on average around 32% of the total query evaluation time for protein searches, and improvements to this stage warrant investigation. In Chapter 4 we described three methods that reduce the time taken to align sequences with negligible effect on accuracy. First, we considered a rearrangement of the dynamic programming recurrence relations that was described by Zhang et al. [1997] however is not well-known. We demonstrated that the new arrangement affords a reduction in computation for each cell in the alignment matrix and leads to a 20% reduction in the time taken to perform the gapped alignment stages of BLAST.

Our most significant innovation in Chapter 4 was a novel sequence comparison method called *semi-gapped* alignment that, when carefully parameterised, leads to 40% faster alignment. Our approach only considers insertions and deletions at fixed intervals in each sequence by dividing cells in the alignment matrix into one of four classes. A different set of recurrence relations is employed to solve cells in each class, depending on whether insertion or deletion events are permitted. We found that semi-gapped alignment closely approximates gapped alignment, and employ our novel technique as an additional search stage between the ungapped and gapped alignment stages of BLAST.

Finally, we described an alignment heuristic called restricted insertion that does not consider adjacent gaps — an insertion follow immediately by a deletion — and leads to a reduction in computation for some cells in the alignment matrix. Our novel restricted insertion approach reduces the time taken to align sequence by roughly 8%. Importantly, all three of our approaches can be used in conjunction with the dropoff heuristic already employed by BLAST to minimise search times. We have integrated the three new methods into our own implementation of the gapped alignment stages of BLAST and show that, when combined, they halve the time taken to align sequences with no significant effect on search accuracy.

The first stage of a BLAST search involves identifying hits; short, fixed-length subsequences or words from the query sequence and the current collection sequence that are identical or highly similar. For protein searches, the task of identifying hits consumes on average around 37% of the total search time, however very little analysis of the BLAST hit detection process has been conducted. In Chapter 5 we considered the effect of the word length and neighbourhood threshold parameters on this first stage. We found that larger word lengths permit a high degree of search accuracy with far less computation, but also increase the size of the lookup structure employed to identify hits. As a result, word lengths of four or greater result in slow search times because the lookup structure does not fit into cache.

Our most significant contribution in Chapter 5 was a novel data structure for identifying hits. Our approach employs a cache conscious deterministic finite automaton that has been carefully optimised to minimise its size and improve search performance. The automaton is significantly smaller than the lookup table employed by NCBI-BLAST; it is between 1% and 15% of the size of the original data structure for the parameters settings tested. Furthermore, our new approach reduces the average time taken to identify hits by 41% when default parameters are employed and is substantially faster when a word length of four is used. We also compared the two-hit and one-hit modes of operation supported by BLAST in Chapter 5 and confirm earlier conjecture that the two-hit mode affords faster search with comparable accuracy [Altschul et al., 1997].

Our analysis of BLAST usage data in Chapter 3 revealed that around 57% of queries related to BLASTN nucleotide searches, and that evaluating nucleotide queries represents a significant investment of computing resources. However, little attention has been paid to the process through which BLAST processes nucleotide collections. In Chapter 6 we described a series of innovations that permit faster nucleotide query evaluation. Each of our schemes is based on the special *byte packed* representation already employed by BLAST, whereby a single byte is used to store four nucleotide bases. Whereas NCBI-BLAST decompresses collection sequences for each stage of search, we proposed novel approaches to each stage that employ lookup tables and numeric comparisons to process sequences in their compressed form and compare four nucleotide bases at a time. Our improvements to the hit detection and ungapped alignment stages are relatively straight forward; we described simple algorithms that provide a substantial reduction in search time with no affect on the result. However, byte-wise comparisons in the gapped alignment stages, where insertions and deletions complicate the alignment of a single byte from the collection, requires more sophisticated methods. We described two alternative techniques, called *bytepacked alignment* and *table-driven alignment*,

for aligning compressed collection sequences. The former allows gaps to start and end only on the collection sequence byte boundary, permitting four bases to be processed at a time without consideration for gaps. The latter uses a lookup table, which contains pre-computed answers, to align four bases from the collection to a single base from the query at a time. When combined with our improvements to the first and second stages of BLAST, our schemes more than halve the average time taken to search nucleotide collections.

Genomic collections such as GenBank contain a large number of near-duplicate or redundant entries. Such internal redundancy leads to larger collection sizes and longer search times; near duplicate sequences consume extra disk space but rarely contribute any insightful information during search, and homology search tools compare the query to multiple near-duplicate entries that are likely to produce a very similar result. In Chapter 7, we described new methods for managing redundancy in genomic collections. Our first contribution was a new approach for identifying pairs of highly similar sequences in a large collection that employs *fingerprinting*, a technique that has been successfully applied to near-duplicate detection for web and text data [Manber, 1994; Heintze, 1996; Broder et al., 1997; Shivakumar and Garcia-Molina, 1999]. We introduced the slotted SPEX algorithm that efficiently identifies redundancy in large genomic collections. Our approach is designed to be suitable for processing genomic data, which exhibits different characteristics to English text. When applied to GenBank, our method is around six times faster than the most successful existing approaches that rely upon an all-against-all comparison. Further, we show that our method scales linearly with collection size and is equally sensitive to existing methods.

Our second contribution in Chapter 7 was a new method for representing and searching redundant entries in protein collections. We group highly similar sequences into clusters, and represent each cluster using a special union-sequence. The union-sequence contains wildcard characters to denote positions in the sequence that vary between the cluster members. We carefully selected a set of seven wildcards for our clustering strategy, whereby each wildcard represents a distinct set of residues and may only be substituted in place of residues that it represents. We also described our method for scoring matches between amino acids in the query sequence and wildcard characters in the union-sequences during search that ensures our clustering approach maintains a high degree of sensitivity. When applied to the GenBank NR protein database, our method reduces collection size by 27% and leads to 22% faster search times, with no noticeable loss in accuracy. Table 8.1 provides an overview of the reductions in BLAST search times that were presented in this thesis.

In this chapter, we discussed several possible extensions to our work. First, our methods

Improvement	Percent original runtime		Cumulative percentage	
	Protein	Nucleotide	Protein	Nucleotide
Improved gapped alignment	83 %	–	85 %	–
New DFA for hit detection	85 %	–	68 %	–
Clustering with wildcards	78 %	–	49 %	–
Bytepacked alignment methods	–	49 %	–	49 %

Table 8.1: Summary of speed improvements to the BLAST *algorithm presented in this thesis for an Intel workstation. Cumulative speed gains are based on reported experimental results rather than interpolated.*

for fast sequence comparison could also be applied to the iterative PSI-BLAST tool. Further, we proposed inserting intermediate iterations into the PSI-BLAST process where similar sequences are rescored, thereby reducing the number of scans of the entire collection. The results from each stage of search could also be reused between the iterations of PSI-BLAST when changes to the profile are small. Next, our scheme for managing redundancy could be applied to PSI-BLAST to minimise the likelihood of profile saturation. The efficient filtering steps of the BLAST algorithm could also be applied to index-based schemes such as CAFE [Williams and Zobel, 1996; 2002] and PATTERNHUNTER [Ma et al., 2002; Li et al., 2004]. Our innovations for managing redundancy could also be applied to nucleotide data. The application of our slotted SPEX approach to English text is also worth investigating. Finally, we propose tighter integration of the individual stages of BLAST to reuse comparison information.

In this thesis, we have proposed new methods for efficient, sensitive search of genomic sequence databases. Our innovations have enabled us to increase the speed of the popular BLAST homology search tool without sacrificing search accuracy. When combined, our new methods for faster sequence comparison more than halve average query evaluation times and are general enough to be applied to other tools. Our improvements to BLAST are embodied in a new, open-source distribution of the tool that is available for download at `http://www.fsa-blast.org/`. With a significant advancement in search performance achieved, we expect that our contributions will form an integral part of state-of-the-art homology search tools in the future.

Bibliography

B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Garland Publishing, New York, USA, third edition, 1994.

B. Alpern, L. Carter, and K. Gatlin. Microparallelism and high-performance protein matching. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 24, New York, USA, 1995. ACM Press.

S. Altschul. A protein alignment scoring system sensitive at all evolutionary distances. *Journal of molecular evolution*, 36(3):290–300, 1993.

S. Altschul. Generalized affine gap costs for protein sequence alignment. *PROTEINS: Structure, Function, and Genetics*, 32(1):88–96, 1998.

S. Altschul, M. Boguski, W. Gish, and J. Wootton. Issues in searching molecular sequence databases. *Nature Genetics*, 6:119–129, 1994.

S. Altschul, R. Bundschuh, R. Olsen, and T. Hwa. The estimation of statistical parameters for local alignment score distributions. *Nucleic Acids Research*, 29(2):351–361, 2001.

S. Altschul and B. Erickson. Locally optimal subalignments using nonlinear similarity functions. *Bulletin of Mathematical Biology*, 48(5–6):633–660, 1986a.

S. Altschul and B. Erickson. A nonlinear measure of subalignment similarity and its significance levels. *Bulletin of Mathematical Biology*, 48(5–6):617–632, 1986b.

S. Altschul and W. Gish. Local alignment statistics. *Methods in Enzymology*, 266:460–480, 1996.

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

S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI–BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

I. Anderson and A. Brass. Searching DNA databases for similarities to DNA sequences: when is a match significant? *Bioinformatics*, 14(4):349–356, 1998.

A. Andreeva, D. Howorth, S. Brenner, T. Hubbard, C. Chothia, and A. Murzin. SCOP database in 2004: refinements integrate structure and sequence family data. *Nucleic Acids Research*, 32:D226–D229, 2004.

A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:316–336, 1987.

L. Aravind and E. Koonin. Gleaning non-trivial structural, functional and evolutionary information about proteins by iterative database searches. *Journal of Molecular Biology*, 287(5):1023–1040, 1999.

T. Attwood and P. Higgs. *Bioinformatics and Molecular Evolution*. Blackwell Publishing, Oxford, UK, 2004.

G. Barton. An efficient algorithm to locate all locally optimal alignments between two sequences allowing for gaps. *Computer Applications in the Biosciences*, 9(6):729–734, 1993.

D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, and D. Wheeler. Genbank. *Nucleic Acids Research*, 33:D34–D38, 2005.

D. Benson, D. Lipman, and J. Ostell. GenBank. *Nucleic Acids Research*, 21(13):2963–2965, 1993.

H. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. Shindyalov, and P. Bourne. The protein data bank. *Nucleic Acids Research*, 28(1):235–242, 2000.

A. Bernal, U. Ear, and N. Kyrpides. Genomes OnLine Database (GOLD): a monitor of genome projects world-wide. *Nucleic Acids Research*, 29(1):126–127, 2001.

Y. Bernstein and M. Cameron. Fast discovery of similar sequences in large genomic collections. In M. Lalmas, A. MacFarlane, S. M. Rüger, A. Tombros, T. Tsikrika, and A. Yavlin-

sky, editors, *28th European Conference on IR Research, ECIR 2006, London, UK, April 10-12, 2006, Proceedings*, pages 432–443, London, UK, 2006. Springer.

Y. Bernstein and J. Zobel. A scalable system for identifying co-derivative documents. In A. Apostolico and M. Melucci, editors, *Proceedings of the String Processing and Information Retrieval Symposium (SPIRE)*, pages 55–67, Padova, Italy, 2004. Springer.

Y. Bernstein and J. Zobel. Redundant documents and search effectiveness. In A. Chowdhury, N. Fuhr, M. Ronthaler, H. Schek, and W. Teiken, editors, *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 736–743, Bremen, Germany, 2005. ACM Press.

B. Billerbeck and J. Zobel. Techniques for efficient query expansion. In A. Apostolico and M. Melucci, editors, *Proceedings of the String Processing and Information Retrieval Symposium (SPIRE)*, pages 30–42, Padova, Italy, 2004. Springer.

A. Bleasby and J. Wootton. Construction of validated, non-redundant composite protein sequence databases. *Protein Engineering*, 3(3):153–159, 1990.

B. Boeckmann, A. Bairoch, R. Apweiler, M. Blatter, A. Estreicher, E. Gasteiger, M. Martin, K. Michoud, C. O'Donovan, I. Phan, S. Pilbout, and M. Schneider. The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003. *Nucleic Acids Research*, 31 (1):365–370, 2003.

N. Bray, I. Dubchak, and L. Pachter. AVID: A global alignment program. *Genome Research*, 13(1):97–102, 2003.

B. Brejova, D. Brown, and T. Vinar. Optimal spaced seeds for homologous coding regions. *Journal of Bioinformatics and Computational Biology*, 1(4):595–610, 2004.

B. Brejova, D. Brown, and T. Vinar. Vector seeds: an extension to spaced seeds. *Journal of Computer and System Sciences*, 70(3):364–380, 2005.

S. Brenner, C. Chothia, and T. Hubbard. Assessing sequence comparison methods with reliable structurally identified distant evolutionary relationships. *Proceedings of the National Academy of Sciences USA*, 95(11):6073–6078, 1998.

S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 398–409, New York, USA, 1995. ACM Press.

A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997.

D. Brown. Multiple vector seeds for protein alignment. In I. Jonassen and J. Kim, editors, *Algorithms in Bioinformatics, 4th International Workshop, WABI 2004, Bergen, Norway, September 17-21, 2004, Proceedings*, pages 170–181. Springer, 2004.

D. Brown. Optimizing multiple seeds for protein homology search. *IEEE Transactions on Computational Biology and Bioinformatics*, 2(1):29–38, 2005.

D. Brown, M. Li, and B. Ma. A tutorial of recent developments in the seeding of local alignment. *Journal of Bioinformatics and Computational Biology*, 2(4):819–842, 2004.

D. Brutlag, J.-P. Dautricourt, R. Diaz, J. Fier, B. Moxon, and R. Stamm. BLAZE: An implementation of the smith-waterman sequence comparison algorithm on a massively parallel computer. *Computers & Chemistry*, 17(2):203–207, 1993.

C. Buckley and E. Voorhees. Evaluating evaluation measure stability. In N. Belkin, P. Ingwersen, and M.-K. Leong, editors, *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 33–40, New York, USA, 2000. ACM Press.

J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.

J. Buhler, U. Keich, and Y. Sun. Designing seeds for similarity search in genomic DNA. In *RECOMB '03: Proceedings of the seventh annual international conference on Research in computational molecular biology*, pages 67–75, New York, USA, 2003. ACM Press.

J. Burke, D. Davison, and W. Hide. d2_cluster: A validated method for clustering EST and full-length DNA sequences. *Genome Research*, 9(11):1135–1142, 1999.

A. Califano and I. Rigoutsos. FLASH: a fast look-up algorithm for string homology. In *Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology*, pages 56–64, 1993.

M. Cameron, Y. Bernstein, and H. E. Williams. Clustered sequence representation for fast homology search. *Journal of Computational Biology*, 2006a. To appear.

M. Cameron, Y. Bernstein, and H. E. Williams. Clustering near-identical sequences for fast homology search. In A. Apostolico, C. Guerra, S. Istrail, P. A. Pevzner, and M. S. Waterman, editors, *Proceedings of the International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 175–189, Venice, Italy, 2006b. Springer.

M. Cameron and H. E. Williams. Comparing compressed sequences for faster nucleotide blast searches. *IEEE Transactions on Computational Biology and Bioinformatics*, 2006. To appear.

M. Cameron, H. E. Williams, and A. Cannane. Improved gapped alignment in BLAST. *IEEE Transactions on Computational Biology and Bioinformatics*, 1(3):116–129, 2004.

M. Cameron, H. E. Williams, and A. Cannane. A deterministic finite automaton for faster protein hit detection in BLAST. *Journal of Computational Biology*, 13(4):965–978, 2006c.

J. Chandonia, G. Hon, N. Walker, L. Conte, P. Koehl, M. Levitt, and S. Brenner. The ASTRAL compendium in 2004. *Nucleic Acids Research*, 32:D189–D192, 2004.

K. Chao, R. Hardison, and W. Miller. Recent developments in linear-space alignment methods: a survey. *Journal of Computational Biology*, 1(4):271–91, 1994.

K. Chao, W. Pearson, and W. Miller. Aligning two sequences within a specified diagonal band. *Computer Applications in the Biosciences*, 8(5):481–487, 1992.

K.-M. Chao. On computing all suboptimal alignments. *Information Sciences*, 105(1-4): 189–207, 1998.

K.-M. Chao, J. Zhang, J. Ostell, and W. Miller. A local alignment tool for very long DNA sequences. *Computer Applications in the Biosciences*, 11(2):147–153, 1995.

X. Chen. A framework for comparing homology search techniques. Master's thesis, School of Computer Science and Information Technology, RMIT University, 2004.

Z. Chen. Assessing sequence comparison methods with the average precision criterion. *Bioinformatics*, 19(18):2456–2460, 2003.

C.-F. Cheung, J. Yu, and H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):90–105, 2005.

F. Chiaromonte, V. Yap, and W. Miller. Scoring pairwise genomic sequence alignments. In *Pacific Symposium on Biocomputing*, pages 115–126, 2002.

K. Choi, F. Zeng, and L. Zhang. Good spaced seeds for homology search. *Bioinformatics*, 20(7):1053–1059, 2004.

M. K.-M. Chow. *Mechanisms of folding and misfolding by ataxin-3, a polyglutamine protein.* PhD thesis, School of Biochemistry and Molecular Biology, Monash University, 2004.

J. Collins, A. Coulson, and A. Lyall. The significance of protein sequence similarities. *Computer Applications in the Biosciences*, 4(1):67–71, 1988.

O. Couronne, A. Poliakov, N. Bray, T. Ishkhanov, D. Ryaboy, E. Rubin, L. Pachter, and I. Dubchak. Strategies and tools for whole-genome alignments. *Genome Research*, 13(1): 73–80, 2003.

M. Crochemore, G. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 679–688, Philadelphia, USA, 2002. Society for Industrial and Applied Mathematics.

G. Crooks, R. Green, and S. Brenner. Pairwise alignment incorporating dipeptide covariation. *Bioinformatics*, 21(19):3704–3710, 2005.

M. Dayhoff, R. Schwartz, and B. Orcutt. A model of evolutionary change in proteins. *Atlas of protein sequence and structure*, 5:345–358, 1978.

A. Delcher, S. Kasif, R. Fleischmann, J. Peterson, O. White, and S. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

A. Deshpande, D. Richards, and W. Pearson. A platform for biological sequence comparison on parallel computers. *Computer Applications in the Biosciences*, 7(2):237–247, 1991.

R. Durbin. *Biological sequence analysis: probabilistic models of proteins and nucleic acids.* Cambridge University Press, 1998.

D. Fetterly, M. Manasse, and M. Najork. On the evolution of clusters of near-duplicate web pages. In R. Baeza-Yates, editor, *Proceedings of the 1st Latin American Web Congress*, pages 37–45, Santiago, Chile, 2003. IEEE.

W. Fitch and T. Smith. Optimal sequence alignments. *Proceedings of the National Academy of Sciences USA*, 80(5):1382–1386, 1983.

C. Fondrat and P. Dessen. A rapid access motif database (RAMdb) with a search algorithm for the retrieval patterns in nucleic acids or protein databanks. *Computer Applications in the Biosciences*, 11(3):273–279, 1995.

M. Galperin. The molecular biology database collection: 2004 update. *Nucleic Acids Research*, 32(Database issue):D3–D22, 2004.

M. Gerstein. Measurement of the effectiveness of transitive sequence comparison, through a third 'intermediate' sequence. *Bioinformatics*, 14(8):707–714, 1998.

V. Gotea, V. Veeramachaneni, and W. Makalowski. Mastering seeds for genomic size nucleotide BLAST searches. *Nucleic Acids Research*, 31(23):6935–6941, 2003.

O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

J. Gracy and P. Argos. Automated protein sequence database classification. I. Integration of compositional similarity search, local similarity search, and multiple sequence alignment. *Bioinformatics*, 14(2):164–173, 1998.

M. Gribskov and N. Robinson. Use of receiver operating characteristic (ROC) analysis to evaluate sequence matching. *Computers & Chemistry*, 20:25–33, 1996.

G. Grillo, M. Attimonelli, S. Liuni, and G. Pesole. CLEANUP: a fast computer program for removing redundancies from nucleotide sequence databases. *Computer Applications in the Biosciences*, 12(1):1–8, 1996.

R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 397–406, New York, USA, 2000. ACM Press.

D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

P. Hall and G. Dowling. Approximate string matching. *ACM Computing Surveys*, 12(4): 381–402, 1980.

J. Hancock and J. Armstrong. SIMPLE34: an improved and enhanced implementation for VAX and Sun computers of the SIMPLE algorithm for analysis of clustered repetitive motifs in nucleotide sequences. *Computer Applications in the Biosciences*, 10(1):67–70, 1994.

N. Heintze. Scalable document fingerprinting. In *1996 USENIX Workshop on Electronic Commerce*, 1996.

S. Henikoff and J. Henikoff. Automated assembly of protein blocks for database searching. *Nucleic Acids Research*, 19:6565–6572, 1991.

S. Henikoff and J. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences USA*, 89(22):10915–10919, 1992.

S. Henikoff and J. Henikoff. Performance evaluation of amino acid substitution matrices. *PROTEINS: Structure, Function, and Genetics*, 17(1):49–61, 1993.

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

L. Holm and C. Sander. Removing near-neighbour redundancy from large protein sequence collections. *Bioinformatics*, 14(5):423–429, 1998.

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

R. Hughey. Parallel hardware for sequence comparison and alignment. *Computer Applications in the Biosciences*, 12(6):473–479, 1996.

N. Hulo, C. J. Sigrist, V. Saux, P. Langendijk-Genevaux, L. Bordoli, A. Gattiker, E. Castro, P. Bucher, and A. Bairoch. Recent improvements to the prosite database. *Nucleic Acids Research*, 32(Database issue):D134–D137, 2004.

M. Itoh, T. Akutsu, and M. Kanehisa. Clustering of database sequences for fast homology search using upper bounds on alignment score. *Genome Informatics*, 15(1):93–104, 2004.

M. Johnson and J. Overington. A structural basis for sequence comparisons. an evaluation of scoring methodologies. *Journal of Molecular Biology*, 233(4):716–738, 1993.

S. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967.

N. Jones and P. Pevzner. *An Introduction to Bioinformatics Algorithms*. MIT Press, 2004.

Y. Kallberg and B. Persson. KIND — a non-redundant protein database. *Bioinformatics*, 15(3):260–261, 1999.

M. Kann, P. Thiessen, A. Panchenko, A. Schäffer, S. Altschul, and S. Bryant. A structure-based method for protein sequence alignment. *Bioinformatics*, 21(8):1451–1456, 2005.

C. Kanz, P. Aldebert, N. Althorpe, W. Baker, A. Baldwin, K. Bates, P. Browne, A. van den Broek, M. Castro, G. Cochrane, K. Duggan, R. Eberhardt, N. Faruque, J. Gamble, F. G. Diez, N. Harte, T. Kulikova, Q. Lin, V. Lombard, R. Lopez, R. Mancuso, M. McHale, F. Nardone, V. Silventoinen, S. Sobhany, P. Stoehr, M. A. Tuli, K. Tzouvara, R. Vaughan, D. Wu, W. Zhu, and R. Apweiler. The EMBL nucleotide sequence database. *Nucleic Acids Research*, 33:D29–D33, 2005.

S. Karlin and S. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Sciences USA*, 87(6):2264–2268, 1990.

K. Karplus, C. Barrett, and R. Hughey. Hidden Markov models for detecting remote protein homologies. *Bioinformatics*, 14(10):846–856, 1998.

K. Karplus, R. Karchin, J. Draper, J. Casper, Y. Mandel-Gutfreund, M. Diekhans, and R. Hughey. Combining local-structure, fold-recognition, and new-fold methods for protein structure prediction. *PROTEINS: Structure, Function, and Genetics*, 53(Supplement 6): 491–496, 2003.

K. Karplus, K. Sjolander, C. Barrett, M. Cline, D. Haussler, R. Hughey, L. Holm, and C. Sander. Predicting protein structure using hidden markov models. *PROTEINS: Structure, Function, and Genetics*, (Supplement 1):134–139, 1997.

W. Kent. BLAT–the BLAST-like alignment tool. *Genome Research*, 12(4):656–664, 2002.

P. Ko, M. Narayanan, A. Kalyanaraman, and S. Aluru. Space-conserving optimal DNA-protein alignment. In *3rd International IEEE Computer Society Computational Systems Bioinformatics Conference (CSB 2004), 16-19 August 2004, Stanford, CA, USA*, pages 80–88. IEEE Computer Society, 2004.

G. Korodi and I. Tabus. An efficient normalized maximum likelihood algorithm for DNA sequence compression. *ACM Transactions on Information Systems*, 23(1):3–34, 2005.

D. Kreil and C. Ouzounis. Comparison of sequence masking algorithms and the detection of biased protein sequence regions. *Bioinformatics*, 19(13):1672–1681, 2003.

S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2), 2004.

G. Lavorgna, R. Triunfo, F. Santoni, U. Orfanelli, S. Noci, A. Bulfone, G. Zanetti, and G. Casari. Antihunter 2.0: increased speed and sensitivity in searching BLAST output for EST antisense transcripts. *Nucleic Acids Research*, 33(Web Server issue):W665–W668, 2005.

A. Lehninger, D. Nelson, and M. Cox. *Principles of Biochemistry*. Worth Publishers, New York, USA, second edition, 1993.

V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Cybernetics and Control Theory*, 10(8):707–710, 1966.

A. Levitin. *Introduction to the Design and Analysis of Algorithms*, chapter 8. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2002.

M. Li, B. Ma, D. Kisman, and J. Tromp. PatternHunter II: Highly sensitive and fast homology search. *Journal of Bioinformatics and Computational Biology*, 2(3):417–439, 2004.

W. Li, L. Jaroszewski, and A. Godzik. Clustering of highly homologous sequences to reduce the size of large protein databases. *Bioinformatics*, 17(3):282–283, 2001a.

W. Li, L. Jaroszewski, and A. Godzik. Tolerating some redundancy significantly speeds up clustering of large protein databases. *Bioinformatics*, 18(1):77–82, 2001b.

W. Li, L. Jaroszewski, and A. Godzik. Sequence clustering strategies improve remote homology recognitions while reducing search times. *Protein Engineering*, 15(8):643–649, 2002.

C. Liébecq, editor. *Biochemical Nomenclature and Related Documents*, pages 122–126. Portland Press, London, UK, 2nd edition, 1992.

H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel BLAST. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, page 72b, 2005.

B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

T. Madden. Personal communication. US National Centre for Biotechnology Information, 2005.

V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusöz, editors, *Combinatorial Pattern Matching, 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5-7, 2004, Proceedings*, pages 420–433. Springer, 2004.

K. Malde, E. Coward, and I. Jonassen. Fast sequence clustering using a suffix array algorithm. *Bioinformatics*, 19(10):1221–1226, 2003.

U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, USA, 1994.

U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

A. Marchler-Bauer and S. Bryant. Cd-search: protein domain annotations on the fly. *Nucleic Acids Research*, 32(Web Server issue):W327–W331, 2004.

S. McGinnis and T. Madden. BLAST: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Research*, 32:W20–W25, 2004.

C. Miller, J. Gurd, and A. Brass. A RAPID algorithm for sequence database comparisons: application to the identification of vector contamination in the EMBL databases. *Bioinformatics*, 15(2):111–121, 1999.

W. Miller. Comparison of genomic DNA sequences: solved and unsolved problems. *Bioinformatics*, 17(5):391–397, 2001.

S. Miyazaki, H. Sugawara, K. Ikeo, T. Gojobori, and Y. Tateno. DDBJ in the stream of various biological data. *Nucleic Acids Research*, 32:D31–D34, 2004.

A. Müller, R. MacCallum, and M. Sternberg. Benchmarking PSI-BLAST in genome annotation. *Journal of Molecular Biology*, 293(5):1257–1271, 1999.

A. Murzin, S. Brenner, T. Hubbard, and C. Chothia. SCOP: a structural classification of proteins database for the investigation of sequences and structures. *Journal of Molecular Biology*, 247(4):536–540, 1995.

E. Myers. An overview of sequence comparison algorithms in molecular biology. Technical Report 91–29, University of Arizona, Department of Computer Science, 1991.

E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5): 345–374, 1994.

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

G. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(2):432–448, 1992.

G. Myers. Whole-genome DNA sequencing. *Computing in Science and Engineering*, 1(3): 33–43, 1999.

G. Myers and R. Durbin. A table-driven, full-sensitivity similarity search algorithm. *Journal of Computational Biology*, 10(2):103–117, 2003.

D. Naor and D. Brutlag. On suboptimal alignments of biological sequences. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *CPM '93: Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, pages 179–196, London, UK, 1993. Springer.

G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33 (1):31–88, 2001.

S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:444–453, 1970.

C. Nevill-Manning and I. Witten. Protein is incompressible. In J. Storer and M. Cohn, editors, *DCC '99: Proceedings of the IEEE Data Compression Conference*, pages 257–266, Washington DC, USA, 1999. IEEE Computer Society.

H. Nicholas, D. Deerfield, and A. Ropelewski. Strategies for searching sequence databases. *BioTechniques*, 28(6):1174–1178, 2000.

Z. Ning, A. Cox, and J. Mullikin. SSAHA: a fast search method for large DNA databases. *Genome Research*, 11(10):1725–1729, 2001.

B. Orcutt and W. Barker. Searching the protein sequence database. *Bulletin of Mathematical Biology*, 46:545–552, 1984.

C. Orengo, A. Michie, S. Jones, D. Jones, M. Swindells, and J. Thornton. CATH–a hierarchic classification of protein domain structures. *Structure*, 5(8):1093–1108, 1997.

A. Panchenko. Finding weak similarities between proteins by sequence profile comparison. *Nucleic Acids Research*, 31(2):683–689, 2003.

J. Park, L. Holm, and C. Chothia. Sequence search algorithm assessment and testing toolkit (SAT). *Bioinformatics*, 16(2):104–110, 2000a.

J. Park, L. Holm, A. Heger, and C. Chothia. RSDB: representative sequence databases have high information content. *Bioinformatics*, 16(5):458–464, 2000b.

J. Park, K. Karplus, C. Barrett, R. Hughey, D. Haussler, T. Hubbard, and C. Chothia. Sequence comparisons using multiple sequences detect three times as many remote homologues as pairwise methods. *Journal of Molecular Biology*, 284(4):1201–1210, 1998.

J. Park, S. Teichmann, T. Hubbard, and C. Chothia. Intermediate sequences increase the detection of homology between sequences. *Journal of Molecular Biology*, 273(1):349–354, 1997.

J. Parsons. Improved tools for DNA comparison and clustering. *Computer Applications in the Biosciences*, 11(6):603–613, 1995.

W. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in Enzymology*, 183:63–98, 1990.

W. Pearson. Comparison of methods for searching protein sequence databases. *Protein Science*, 4:1145–1160, 1995.

W. Pearson. Effective protein sequence comparison. *Methods in Enzymology*, 266:227–258, 1996.

W. Pearson. Empirical statistical estimates for sequence similarity searches. *Journal of Molecular Biology*, 276(1):71–84, 1998.

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

W. Pearson and D. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences USA*, 85(8):2444–2448, 1988.

W. Pearson and W. Miller. Dynamic programming algorithms for biological sequence comparison. *Methods in Enzymology*, 210:575–601, 1992.

S. Pietrokovski, J. Henikoff, and S. Henikoff. The blocks database-a system for protein classification. *Nucleic Acids Research*, 24(1):197–201, 1996.

F. Plewniak, J. Thompson, and O. Poch. Ballast: Blast post-processing based on locally conserved segments. *Bioinformatics*, 16(9):750–759, 2000.

M. Pop, S. Salzberg, and M. Shumway. Genome sequence assembly: Algorithms and issues. *Computer*, 35(7):47–54, 2002.

B. Rapp and D. Wheeler. Bioinformatics resources from the national center for biotechnology information: An integrated foundation for discovery: Research articles. *Journal of the American Society for Information Science and Technology*, 56(5):538–550, 2005.

G. Reeck, C. de Haen, D. Teller, R. Doolittle, W. Fitch, R. Dickerson, P. Chambon, A. McLachlan, E. Margoliash, T. Jukes, and E. Zuckerkandl. "Homology" in proteins and nucleic acids: a terminology muddle and a way out of it. *Cell*, 50(5):667, 1987.

J. Reese and W. Pearson. Empirical determination of effective gap penalties for sequence comparison. *Bioinformatics*, 18(11):1500–1507, 2002.

A. Robinson and L. Robinson. Distribution of glutamine and asparagine residues and their near neighbors in peptides and proteins. *Proceedings of the National Academy of Sciences USA*, 88(20):8880–8884, 1991.

T. Rognes and E. Seeberg. SALSA: improved protein database searching by a new algorithm for assembly of sequence fragments into gapped alignments. *Bioinformatics*, 14(10):839–845, 1998.

T. Rognes and E. Seeberg. Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000.

D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, USA, 1983.

A. Schaffer, L. Aravind, T. Madden, S. Shavirin, J. Spouge, Y. Wolf, E. Koonin, and S. Altschul. Improving the accuracy of PSI–BLAST protein database searches with composition-based statistics and other refinements. *Nucleic Acids Research*, 29(14):2994–3005, 2001.

A. Schaffer, Y. Wolf, C. Ponting, E. Koonin, L. Aravind, and S. Altschul. IMPALA: matching a protein sequence against a collection of PSI–BLAST-constructed position-specific score matrices. *Bioinformatics*, 15(12):1000–1011, 1999.

S. Schwartz, W. Kent, A. Smit, Z. Zhang, R. Baertsch, R. Hardison, D. Haussler, and W. Miller. Human-mouse alignments with BLASTZ. *Genome Research*, 13(1):103–107, 2002.

R. Sedgewick. *Algorithms in C.* Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1990.

P. Sellers. On the theory and computation of evolutionary distances. *Journal of Applied Mathematics*, 26:787–793, 1974.

P. Sellers. Pattern recognition in genetic sequences by mismatch density. *Bulletin of Mathematical Biology*, 46(4):501–514, 1984.

N. Shivakumar and H. Garcia-Molina. Finding near-replicas of documents on the web. In *WEBDB: International Workshop on the World Wide Web and Databases, WebDB.* LNCS, 1999.

E. Shpaer, M. Robinson, D. Yee, J. Candlin, R. Mines, and T. Hunkapiller. Sensitivity and selectivity in protein similarity searches: A comparison of Smith-Waterman in hardware to BLAST and FASTA. *Genomics*, 38:179–191, 1996.

M. Sierk and W. Pearson. Sensitivity and selectivity in protein structure comparison. *Protein Science*, 13(3):773–785, 2004.

T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

J. Spouge. Fast optimal alignment. *Computer Applications in the Biosciences*, 7(1):1–7, 1991.

D. States and P. Agarwal. Compact encoding strategies for DNA sequence similarity search. In D. J. States, P. Agarwal, T. Gaasterland, L. Hunter, and R. Smith, editors, *Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology, St. Louis, MO, USA, June 12-15 1996*, pages 211–217. AAAI Press, 1996.

D. States, W. Gish, and S. Altschul. Improved sensitivity of nucleic acid database searches using application-specific scoring matrices. *Methods: A companion to Methods in Enzymology*, 3(1):66–70, 1991.

T. A. Sudkamp. *Languages and machines: an introduction to the theory of computer science*. Addison-Wesley Longman Publishing Co., Inc., 1997.

Y. Sun and J. Buhler. Designing multiple simultaneous seeds for DNA similarity search. In *RECOMB '04: Proceedings of the eighth annual international conference on Research in computational molecular biology*, pages 76–84, New York, USA, 2004. ACM Press.

S. Tata, R. Hankins, and J. Patel. Practical suffix tree construction. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 36–47. Morgan Kaufmann, 2004.

W. Taylor. The classification of amino-acid conservation. *Journal of Theoretical Biology*, 119:205–218, 1986.

M. Waterman. General methods for sequence comparison. *Bulletin of Mathematical Biology*, 46(4):473–500, 1984.

M. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. *Journal of Molecular Biology*, 197(4):723–728, 1987.

M. Waterman and M. Vingron. Rapid and accurate estimates of statistical significance for sequence data base searches. *Proceedings of the National Academy of Sciences USA*, 91 (11):4625–4628, 1994.

O. Weiss, M. Jimenez-Montano, and H. Herzel. Information content of protein sequences. *Journal of Theoretical Biology*, 206(3):379–386, 2000.

D. Wheeler, D. Church, R. Edgar, S. Federhen, W. Helmberg, T. Madden, J. Pontius, G. Schuler, L. Schriml, E. Sequeira, T. Suzek, T. Tatusova, and L. Wagner. Database resources of the national center for biotechnology information: update. *Nucleic Acids Research*, 32(Database issue):D35–40, 2002.

C. White, R. Singh, P. Reintjes, J. Lampe, B. Erickson, W. Dettloff, V. Chi, and S. Altschul. Bioscan: A vlsi-based system for biosequence analysis. In *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 504–509, Washington, USA, 1991. IEEE Computer Society.

W. Wilbur and D. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences USA*, 80(3):726–730, 1983.

H. E. Williams. Genomic information retrieval. In K.-D. Scheme and X. Zhou, editors, *Proceedings of the Australasian Database Conference*, pages 27–35, Adelaide, Australia, 2003.

H. E. Williams and J. Zobel. Indexing nucleotide databases for fast query evaluation. In P. Apers, M. Bouzeghoub, and G. Gardarin, editors, *EDBT '96: Proceedings of the 5th International Conference on Extending Database Technology*, pages 275–288, London, UK, 1996. Springer-Verlag.

H. E. Williams and J. Zobel. Compression of nucleotide databases for fast searching. *Computer Applications in the Biosciences*, 13(5):549–554, 1997.

H. E. Williams and J. Zobel. Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering*, 14(1):63–78, 2002.

I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, USA, 1999.

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

J. Wootton and S. Federhen. Analysis of compositionally biased regions in sequence databases. *Methods in Enzymology*, 266:554–571, 1996.

C. Wu, L.-S. Yeh, H. Huang, L. Arminski, J. Castro-Alvear, Y. Chen, Z.-Z. Hu, R. Ledley, P. Kourtesis, B. E. Suzek, C. R. Vinayaka, J. Zhang, and W. C. Barker. The protein information resource. *Nucleic Acids Research*, 31(1):345–347, 2003.

S. Wu, U. Manber, and E. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

T. Yap, O. Frieder, and R. Martino. Parallel computation in biological sequence analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):283–294, 1998.

Y. Yu, J. Wootton, and S. Altschul. The compositional adjustment of amino acid substitution matrices. *Proceedings of the National Academy of Sciences USA*, 100(26):15688–15693, 2003.

J. Zhang and T. Madden. PowerBLAST: A new network BLAST application for interactive or automated sequence analysis and annotation. *Genome Research*, 7(6):649–656, 1997.

Z. Zhang, P. Berman, and W. Miller. Alignments without low-scoring regions. *Journal of Computational Biology*, 5(2):197–210, 1998a.

Z. Zhang, W. Pearson, and W. Miller. Aligning a DNA sequence with a protein sequence. *Journal of Computational Biology*, 4(3):339–349, 1997.

Z. Zhang, A. Schaffer, W. Miller, T. Madden, D. Lipman, E. Koonin, and S. Altschul. Protein sequence similarity searches using patterns as seeds. *Nucleic Acids Research*, 26 (17):3986–3990, 1998b.

Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7(1–2):203–214, 2000.

M. Zuker. Suboptimal sequence alignment in molecular biology. alignment with error analysis. *Journal of Molecular Biology*, 221(2):403–420, 1991.