Explorations In Searching Compressed Nucleic Acid And Protein Sequence Databases And Their Cooperatively-Compressed Indices

Paul Gardner-Stephen

Bachelor of Science,

Flinders University of South Australia

A Thesis Submitted for the Degree of Doctor of Philosophy

Flinders University

School of Informatics and Engineering

Adelaide, South Australia

2008

(Submitted 20 December 2007)

Acknowledgements

Before I acknowledge those who have helped me make it through my candidature, I wish to thank the unbroken chain of people who have believed in me, encouraged me, nurtured and supported me, and without whom I would not have made it to the starting line: Mrs Jan Stephen, my mother, who first kindled my interest in science, and Mr. Keith Gardner, my father; Mr. Peter Brune, a primary school teacher, who believed that I would be someone great one day (perhaps I will get there yet); Dr. Allen Hodson, a high school teacher who taught me more lessons about life and statistics than I suspect he realises; Dr. Todd Rockoff, a university lecturer who kindled my interest in hardware design; Mr. Michael Renz of Germany, who taught me my first lessons about the commercial world; Mr. Murray Rogers, who took a scruffy young university student and made a respectable (but still scruffy) Systems Administrator; and Prof. Greg Knowles, who had enough faith in me to take me on as his student.

For their support throughout my candidature, I wish to thank again: Mr. Murray Rogers for graciously releasing my time so that I could study; and Prof. Greg Knowles for his support and investment as my supervisor. In addition to these people I wish to thank: Prof. Janet Verbyla as head of the School of Informatics & Engineering, and my colleagues (academic staff, general staff and fellow candidates) for their support and friendship during my candidature, and also my friends who have not abandoned me, even though “as busy as a Gardner-Stephen” has become a proverb among them.

A special mention must go to Ms. Fran Banytis of the Staff Development & Training Unit, for the way she has invested much of her self into my candidature. Her encouragement and support throughout my candidature has been as valuable as it has been unexpected.

Finally I would like to thank Dr Dione Gardner-Stephen, my beautiful wife who not only loved me enough to trade in her perfectly good maiden name for one that no-one can spell, but believed in me and supported me through the thick and thin of candidature, not only with the insight and understanding of another doctoral candidate, but in the way that only a loving wife can. I love you much more than words can express.

Through the actions of all these people I see the guiding hand of a God of love, and who has been gracious enough, not merely to bring me to this place, but to invite me to know Him personally. That God is Jesus Christ of Nazareth, and to Him I say “thank you” for the contributions that each of you have made in my life.

Abstract

Nucleic acid and protein databases such as GenBank are growing at a rate that perhaps eclipses even Moore's Law of increase in computational power[1]. This poses a problem for the biological sciences, which have become increasingly dependant on searching and manipulating these databases. It was once reasonably practical to perform exhaustive searches of these databases, for example using the algorithm described by Smith and Waterman, however it has been many years since this was the case. This has led to the development of a series of search algorithms, such as FASTA, BLAST and BLAT, that are each successively faster, but at similarly successive costs in terms of thoroughness.

Attempts have been made to remedy this problem by devising search algorithms that are both fast and thorough. An example is CAFE, which seeks to construct a search system with a sub-linear relationship between search time and database size, and argues that this property must be present for any search system to be successful in the long term.

This dissertation explores this notion by seeking to construct a search system that takes advantage of the growing redundancy in databases such as GenBank in order to reduce both the search time and the space required to store the databases and their indices, while preserving or increasing the thoroughness of the search.

The result is the creation and implementation of new genomic sequence search and alignment, database compression, and index compression algorithms and systems that make

[1]More accurately, Moore's Law predicts that the capacity for transistors on an integrated circuit will double approximately every two years. In practice, due to the efforts of computer architects this has translated into a roughly corresponding increase in computation throughput.

progress toward resolving the problem of reducing search speed and space requirements while improving sensitivity. However, success is tempered by the need for databases with adequate local redundancy, and the computational cost of these algorithms when servicing un-batched queries.

"I Paul Gardner-Stephen, certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text."

Candidate:

Paul Gardner-Stephen

Contents

List of Figures

List of Tables

Part I

Introductory

Chapter 1

Introduction

This introductory chapter provides the context and the definition of the thesis of this dissertation. The following sections present: (1) the motivation for this dissertation; (2) the aims and hypotheses of this dissertation; (2) the specification of the scope and re-statement of thesis; and, finally, (3) the structure of this dissertation. The chapter closes by identifying the specific thesis that will be tested by this dissertation, as well as outlining how that thesis is assessed in the remainder of this dissertation.

1.1 Motivation

Biologists and bioinformaticists are reliant on nucleic acid and protein sequence databases as they push the boundaries of their science in the twenty first century. Collectively, these databases are growing at an exponential rate[1] that equals or surpasses Moore's Law (Moore 1965) of increase in computational power[2]. Moreover, these databases are being searched more often as biology becomes more information oriented. As a result, the total time expended on searching these databases is increasing over time. It is many years since

[1] `http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html` [On line; accessed 4 September 2006].

[2] Moore's Law predicts that the capacity for transistors (and, by implication, computational performance) on an integrated circuit will double approximately every two years — not every 18 months as is often quoted (Moore 2005). The current doubling interval (2006) is slightly shorter than two years, but still longer than the doubling interval for the GenBank nucleotide sequence database (18 months) (Benson et al. 2006).

it was computationally feasible to routinely search a major database, such as GenBank (Benson et al. 2006), using an exhaustive, i.e., completely sensitive, alignment algorithm such as the one described by Smith and Waterman (1981).

Increased search times have created a significant motivation to make progressively faster sequence search and alignment algorithms. This pressure has prompted the emergence of a series of increasingly rapid algorithms, which have progressively sacrificed sensitivity in order to reduce search times. The first major instance of this phenomenon was the rise of FASTA (Pearson 1990) at the expense of the Smith-Waterman algorithm. Use of FASTA, in turn, has largely been replaced by the faster, but less sensitive, NCBI-BLAST (Altschul et al. 1997). This process is continuing, with yet faster algorithms gaining popularity, such as BLAT (Kent 2002).

This constant sacrificing of sensitivity in order to maintain acceptable search times has not gone completely unnoticed. Dwan (2002) has raised the issue of the cost of increased sensitivity sacrifices, particularly as end users are rarely aware of the precise trade-offs that have been made. Williams and Zobel have also noticed this problem, and taken a proactive approach by constructing a fast search system, CAFE (Williams and Zobel 2002a, Williams 1999, Williams and Zobel 1997b), which endeavours to make no further compromises against sensitivity. They argue that as computational power continues to lag behind the growth of nucleic acid and protein databases, that for any algorithm to be successful in the long term, it must exhibit a sub-linear relationship between the size of a database being searched, and the time and space required to service the search — an opportunity that the CAFE system partially explores. This goal is further explored in this dissertation by considering the increasing redundancy that is present in nucleic acid and protein databases as they increase in size.

In summary, nucleic acid and protein sequence search and alignment is facing a looming crisis as the annualised increase in computational capacity lags behind the combined ef-

fects of: (a) increasing database sizes, and; (b) increasing search demand by users. As long as this remains true, the pursuit must continue for increasingly efficient, yet sensitive, generations of sequence search and alignment algorithms. Therefore, nucleic acid and protein sequence searching and alignment is still an open topic (Dwan 2002).

1.2 Statement And Scope Of Thesis

1.2.1 Introduction

By crafting a compression algorithm such that recurrences are explicitly coded, it is possible to enhance the indexing and searching processes of an index driven biological sequence search and alignment in the following ways:

1. By indexing only one occurrence of a repetition, the index will contain fewer postings, and should, therefore, require less *space*.

2. The reduction in the number of indexed items also translates into a natural reduction in search *time*.

3. Because the discovery of an alignment that intersects a recurrent region can have the intersecting segment cloned onto each other recurrence, it can be used to seed an alignment that may not be discovered otherwise, thus increasing search *sensitivity*.

In this way it is possible to simultaneously attack the three way trade-off of space, time and sensitivity in biological sequence search and alignment.

To illustrate this concept, consider the Biblical text of Nehemiah 3:14a (while, for clarity, only English text is used here in this example, the arguments are equally applicable to nucleotide, protein and similar classes of character sequences). Table 1.1 presents the text as it appears in a variety of translations (International Bible Society 1973 - 1984, Darby 1890,

Table 1.1: The Multiple Text Alignment Of Eight Translations Of Nehemiah 3:14a.

NIV:	The Dung Gate was		repaired by	Malkijah		son of Recab,	
D-T:	And the dung-gate		repaired	Malchijah	the	son of Rechab,	
D-R:	And the gate of the dunghill			Melchias	the	son of Rechab	
WEB:	The dung gate		repaired	Malchijah	the	son of Rechab,	
Web:	But the dung-gate		repaired	Malchiah	the	son of Rechab,	
YLT:	And the dung-gate		hath	Malchijah		son of Rechab,	
NAS:				Malchijah	the	son of Rechab,	
T-M:	The Dung Gate itself	was	rebuilt by	Malkijah		son of Recab,	

NIV:	ruler	of the	district of	Beth Hakkerem.	
D-T:	the chief	of the	district of	Beth-haccerem;	
D-R:	built, lord	of the	street of	Bethacharam :	he built it,
WEB:	the ruler	of the	district of	Beth Haccherem;	
Web:	the ruler	of part		of Beth-haccerem;	
YLT:	head	of the	district of	Beth-Haccerem,	strengthened;
NAS:	the official	of the	district of	Beth-haccherem	repaired the Refuse Gate
T-M:	the mayor	of the	district of	Beth Hakkerem;	

Challoner 1752, Johnson 2003, Webster 1833, Young 1898, Barker et al. 1960 - 1995, Peterson 2002). The text has been aligned to assist the reader. The translations differ in form by varying degrees, yet share similar function, somewhat analogously to homologous nucleotide or protein sequences.

1.2.2 Revealing Recurrences Through Data Compression

That the set of translations can be gainfully compressed is apparent because of the many shared strings and words among them. In the present context, we are primarily interested in encoding each successive translation as a combination of new text and *recurrences* from the preceding translations. In this context, recurrence refers to the repeated appearance of a string of characters or words. Table 1.2 shows the coverage of the texts by recurrences, if word level redundancies were the basis, and where the minimum length or length of *consensus* is two words. Of the 143 words, 88 (62%) are identified as recurrent in this way. If the criterion were reduced to recurrences of a single word, 105 (73%) words are identified as recurrent.

Table 1.2: Eight translations of Nehemiah 3:14a, marked with inter-translation repetitions of two or more words (ignoring case and punctuation). Only recurrences are marked. A marking of "NIV3" indicates the original instance of the text occurred in the NIV translation, beginning at the third word.

Source	Text of Nehemiah 3:14a
NIV	The Dung Gate was repaired by Malkijah son of Recab, ruler of the district of Beth Hakkerem.
D-T	And [the dung-gate]^{NIV1} repaired Malchijah the [son of]^{NIV8} Rechab, the chief [of the district of Beth]^{NIV12}-haccerem
D-R	[And the]$^{D-T1}$ gate [of the]^{NIV12} dunghill Melchias [the son of Rechab]$^{D-T6}$built, lord [of the]^{NIV12} street of Bethacharam : he built it.
WEB	[The dung gate repaired Malchijah the son of Rechab, the]$^{D-T1}$ [ruler of the district of Beth]^{NIV11} Haccerem;
Web	But [the dung-gate repaired]$^{D-T2}$ Malchiah [the son of Rechab, the ruler of]^{WEB6} part [of Beth-haccerem;]$^{D-T16}$
YLT	[And the dung-gate]$^{D-T1}$ hath Malchijah [son of Rechab]$^{D-T8}$, head [of the district of Beth-Haccerem]$^{D-T13}$, strengthened;
NAS	[Malchijah the son of Rechab]$^{D-T6}$, the official [of the district of Beth-haccerem]$^{D-T13}$ repaired the Refuse Gate.
T-M	[The Dung Gate]$^{D-T2}$ itself was rebuilt [by Malkijah son of Recab]^{NIV7}, the mayor [of the district of Beth Hakkerem]^{NIV12};

While this is a relatively trivial example, it highlights the kind of redundancy that may be present in a group of related texts — whether English prose, or nucleotide sequences. This concept is not new, and is utilised in varying forms by most general purpose compression algorithms, especially dictionary based algorithms such as those based on the Ziv-Lempel family of algorithms.

The Ziv-Lempel dictionary based approaches have the property that the recurrence information is clearly expressed in the encoded message. This contrasts with a statistical compressor that represents this information implicitly in its statistical model. It is true that a statistical compressor, such as Dynamic Markov Compression, may yield better compression. However, in that case the recurrence information must be represented separately,

probably negating any compression gains for an index of the compressed data. Further, statistical compressors generally require a certain volume of data in order to achieve their superior performance. This works well in general purpose compression, where the normal use is serial compression or decompression of an entire corpus. However, for data compression to be useful in biological sequence search and alignment, it must be possible to rapidly retrieve single (often short) sequences from a large database.

Finally, statistical compressors generally use Arithmetic Coding, which severely limits decompression speed. While this all but rules out the use of adaptive statistical compressors, it again plays to the strengths of dictionary based algorithms. This is because it is possible to allow a dictionary based compressor to reference recurrences only in neighbouring records (sequences) in a database, without unduly sacrificing the performance of either compression or the speed when retrieving random sequences. In this way it is possible to create a compression algorithm that retains credible performance while making *recurrence records*, that is the location and size of recurrences, readily available to other processes.

1.2.3 Using Recurrences To Compress Indices

An example of a potential user of recurrence records in a compressed database, is a process that seeks to create an index of that data. It is possible to make a more compact index by making use of the recurrence information, and knowing that any reader of the index can do like wise. If this information were not available, it would be necessary to index every occurrence of every word.

A simple application is to index only those instances of words that cannot be located in a recurrence. The effect of this is illustrated in Table 1.3 where such an index is presented for the text of Nehemiah 3:14a. The individual words not included in the index can be found by taking the list of occurrences that are in the index, and then identifying the recurrences that reference those particular words.

Taking the example of “ruler”, of its three actual occurrences, only one is recorded in the index, the one corresponding to the eleventh word of the NIV text. However, by referring back to Table 1.2 we find that the WEB text makes reference to this. The list of occurrences is now NIV{11}, WEB{11}. The process is now repeated for the newly added instance. This leads to the discovery that the word ruler also occurs at Web{12}. As no subsequent texts refer to this instance, the list of occurrences is now final at NIV{11}, WEB{11} and Web{12}. Thus the index can efficiently look up all instances of any word, even though it contains direct references to only 38% of the text.

That excluding postings from the index should translate into a smaller compressed inverted list is not certain. This is because this process may destroy the clusters of term occurrences that the best inverted list coding schemes make use of. The clusters that aid inverted list compression are clusters of the *same* word or term occurring, with strong spacial locality. The cooperative compression of the index and database excludes adjacent words in common phrases from the index, requiring only one instance of each word in the repeating phrase to be indexed. Therefore a cluster of any single term may be reduced to a single instance. Therefore the compression benefit of the cluster may be inhibited. However, with fewer pointers requiring encoding, this should still result in smaller size overall. (An exception to this would be if a term occurs with probability $p > 0.5$. In that case removing pointers will actually result in the entropy of the list increasing as it is shortened. When this occurs, it would be better not to thin the list out.)

1.2.4 Using Recurrences To Improve Search Sensitivity

The second useful property of a compression scheme that makes recurrence information explicit is in the context of index driven sequence search and alignment. The reduction in index size comes from following the chain of recurrences back to their origins. By following the chain of recurrences in the reverse order each time an alignment is identified,

Table 1.3: Index Of Instances Of Words In Nehemiah 3:14a, Excluding Those That Occur In Identified Recurrences. Only 55 (38%) of the 143 occurrences are indexed in order to cover the entire text.

Word	Indexed Instances	Word	Indexed Instances
And	D-T{1}	Malchiah	Web{6}
Beth	NIV{16}	Malchijah	D-T{5}, YLT{6}
Bethacharam	D-R{18}	Malkijah	NIV{7}
built	D-R{12,20}	mayor	T-M{13}
but	Web{1}	Melchias	D-T{7}
by	NIV{6}	of	NIV{9,12,15}, D-R{17}
chief	D-T{12}	official	NAS{7}
district	NIV{14}	part	Web{14}
dung	NIV{2}	Recab	NIV{10}
dunghill	D-R{6}	Rechab	D-T{10}
gate	NIV{3}, D-R{3}, NAS{17}	refuse	NAS{16}
Haccerem	D-T{18},WEB{17}	rebuilt	T-M{6}
Hakkerem	NIV{17}	repaired	NIV{5}, D-T{5}, NAS{14}
hath	YLT{5}	ruler	NIV{11}
he	D-R{19}	son	NIV{8}
head	YLT{11}	street	D-R{16}
it	D-R{21}	strengthened	YLT{18}
itself	T-M{4}	the	NIV{1,13}, D-T{7,11}, NAS{6,15}, T-M{12}
lord	D-R{13}	was	NIV{4}, T-M{5}

it is possible to increase the sensitivity of the search to less conserved sequences. This is because index based search algorithms ordinarily require there to be a consensus, with some minimum length, between the query and a sequence. Once a consensus is found, it can be extended to its maximum length. During this extension phase, the alignment is normally allowed to be approximate. When such an alignment segment spans a recurrence, the recurrence information can be used to immediately identify all instances of the recurrence. Hence candidate alignments can be identified without requiring them to contain the minimum consensus length mandated by the index.

By way of illustration of how sensitivity can be gained in this way, consider searching for the NIV version of the text of Nehemiah 3:14a in an index based search system with a minimum consensus length of five words. The only phrase of the text that would be found in any other translation would be “of the district of Beth”, as illustrated in Table 1.4. This would be sufficient for a competent index based search algorithm to discover the complete alignment of the NIV translation against the D-T, WEB, YLT, NAS and T-M translations.

However, no alignments would be discovered against the D-R or Web texts, because there is no consensus of five or more words between them and the NIV text — despite their functional (semantic) similarity with the NIV query text. This is a sensitivity blind spot of traditional index based search algorithms.

By making use of the recurrence information from Table 1.2, it is possible to efficiently detect the missing alignments: The D-R text makes reference to the recurrence of “the son of Rechab”, which first occurs in D-T. The consensus here does not need to be five words, nor does it need to exactly match the query text, as the index is not involved in its discovery: The recurrence records in the compressed database provides the required information. The alignment against the Web translation can be discovered in a similar way, through either “the dung-gate repaired” or “the son of Rechab”, both of which first occur in the D-T translation. Hence additional alignments are discovered, while substantially preserving the

Table 1.4: Consensus Region Between NIV And Other Translations Of Nehemiah 3:14a, Minimum Length Of Five Words. No consensus is identified with the D-R or Web translations.

Source	Text of Nehemiah 3:14a
NIV	The Dung Gate was repaired by Malkijah son of Recab, ruler *of the district of Beth* Hakkerem.
D-T	And the dung-gate repaired Malchijah the son of Rechab, the chief *of the district of Beth*-haccerem
D-R	And the gate of the dunghill Melchias the son of Rechab built, lord of the street of Bethacharam : he built it.
WEB	The dung gate repaired Malchijah the son of Rechab, the ruler *of the district of Beth* Haccerem;
Web	But the dung-gate repaired Malchiah the son of Rechab, the ruler of part of Beth-haccerem;
YLT	And the dung-gate hath Malchijah son of Rechab, head *of the district of Beth*-Haccerem, strengthened;
NAS	Malchijah the son of Rechab, the official *of the district of Beth*-haccerem repaired the Refuse Gate.
T-M	The Dung Gate itself was rebuilt by Malkijah son of Recab, the mayor *of the district of Beth* Hakkerem;

high speed of index driven searching: This is where a gain in sensitivity can be realised. The computational cost of this additional sensitivity is to traverse through the recurrence records in the compressed database, and to decompress each sequence that is identified as containing an alignment candidate. A potential pitfall of this method is that because chains of unknown length must be traversed to reconstruct the index, there is an unpredictable contribution to search time. This is a significant issue, particularly if the data are to be disk-resident, as each step in the chain will trigger a costly disk seek.

1.2.5 Opportunity And Research Questions

The potential compactness and sensitivity gains discussed above are interesting because they are precisely the short comings that have prevented index driven searching from becoming mainstream: BLAST retains its dominance, for protein searching in particular,

because it has modest space requirements, and none of the well known faster algorithms can match its sensitivity, as is shown later in this dissertation.

The challenge comes in that this opportunity has been shown in the context of a trivial example only, where recurrence information is abundant. It may not be reasonable to expect biological sequence databases to exhibit such positive characteristics. However, fortunately there exists highly-redundant pre-sorted biological databases, such as the UniGene transcriptome databases that can be used to test this approach. But even assuming that postings can be thinned out from the inverted list, it is not clear whether the resultant lists will actually compress more compactly, as the natural clustering of terms may be destroyed by this process.

From a practical perspective, additional challenges exist related to time and space efficiency. The indexing methods described here require traversal of the compressed data stream in order to make effective use of the recurrence information. This may prove computationally prohibitive, or result in too many costly random disk accesses. This may imply that the database and index must be memory resident in order to obtain acceptable performance, and there is the risk that the compressed database and index may simply be too large, and preclude the method from being competitive.

1.2.6 Summary And Statement Of Thesis

In the preceding text, it has been argued that it is possible, in theory, to compress data (and nucleic acid and protein sequence data specifically) in such a way that the recurrences that occur within the data are made visible. This information can then be employed to produce more compact index structures, and to increase the sensitivity of searches that use them. The result being an indexed search and alignment algorithm with better size and sensitivity parameters than the current state of the art, while substantially preserving the desirable speed characteristics present in existing index based methods.

However, this is not without challenges. It is not clear whether recurrences exist in real databases in sufficient quality and quantity to be profitable, nor is it clear whether the final product will exhibit sufficient speed, sensitivity and overall compactness in order to be an attractive alternative to existing methodologies. This dissertation examines these issues by considering the thesis that:

By crafting a compression algorithm such that recurrences are explicitly coded, it is possible to enhance the indexing and searching processes of an index driven nucleic acid and protein sequence search and alignment in the following ways: (1) By indexing only one occurrence of a repetition, the index will contain fewer postings, and should, therefore, require less space*; (2) The reduction in the number of indexed items also translates into a natural reduction in search* time*, and; (3) Because the discovery of an alignment that intersects a recurrent region can have the intersecting segment cloned onto each other recurrence, it can be used to seed a full alignment that may not be discovered otherwise, thus increasing search* sensitivity*. In this way it is possible to simultaneously attack the three way trade-off of space, time and sensitivity in biological sequence search and alignment.*

1.3 Assumptions

The assumption is made in this dissertation that the algorithms and systems proposed will be run on either a capable desktop type computer (with several GB of RAM), or a cluster of such systems, and that the database and index size will be such that it can fit entirely in the RAM of the computer or computers concerned. This assumption is made because it is recognised that the reconstruction of omitted index postings requires random-access retrieval of additional database records, which would most likely result in poor throughput. It is accepted that this is atypical for this kind of algorithms, and that therefore the speed comparison with the peer group of algorithms is potentially slanted in favour of the algo-

rithms described in this dissertation. The application of the algorithms presented in this dissertation to disk resident searching is discussed in principle in Section 7.9.

It is further assumed that the data volume is of a scale where sorting and clustering is possible, which at the time the work was carried out meant databases of no more than several giga-bases. This means that the approaches described in this dissertation would require some adaption if they were intended to be used with a dynamic data set, as might be the case in assembling shot-gun sequencing fragments. In that case the subsequentially developed approach of Bernstein and Cameron (2006) would be appropriate. Indeed, this application of assembly of shot-gun sequences, while now a popular use of sequence search and alignment systems, has not been a consideration of this dissertation. This is also reflected in the selection of queries in Chapter 3, where longer queries hundreds to thousands of bases are used, rather than the dozens to low hundreds of bases in length that is more typical of shot-gun sequencing. This narrowing of focus is reasonable given that specialised algorithms used to perform shot-gun sequence assembly, e.g., (Batzoglou et al. 2002, Jaffe et al. 2003, Chaisson and Pevzner 2008), differ substantially from general purpose sequence alignment algorithms, in part because general purpose sequence alignment algorithms are too slow for that application.

1.4 Contributions

This dissertation attempts to contribute to the state of the art in sequence search and alignment by devising and testing several new algorithms related to nucleic acid and protein sequence search and alignment, with a particular focus on reducing search time and space requirements without sacrificing search sensitivity.

The first algorithm, DASH, seeks to create a sequence search and alignment system that can efficiently search both uncompressed databases and indices and those where redundant

records and phrases within records share their storage to save space. This endeavour was successful.

The second algorithm, FOLDDB, seeks to create a compact database and index structure for use with DASH that when faced with redundant records shares their storage to save space. This endeavour, described in Chapter 5 was successful when provided with a database with sufficient redundant records.

The third and fourth algorithms, NP3 and NIX attempt to refine FOLDDB by devising a database and index structure that shares the storage of redundant phrases or sub-records, while also being suitable for high-speed sequence search and alignment. This endeavour faced mixed results. A more compact database and index representation was obtained. However, while that representation supported fast random record retrieval, it was not entirely successful when applied to sequence search and alignment. This was due to the time complexity involved in the exhaustive reconstruction and searching of index records, which while improving sensitivity, resulted in poor search speed for processing individual queries. However, the partitioned characteristics of the NP3/NIX data formats means that this computational cost can be amortised over batches of queries such that DASH+NP3/NIX can search a disk-resident database several times faster than NCBI-BLAST can search a memory-resident database.

1.5 Structure Of Thesis

Immediately following this chapter, Chapter 2 provides background information for this dissertation, including an overview of the popular BLAST sequence alignment program. Chapter 3 follows this by describing the materials and methods used in this dissertation, covering: (a) the databases to be searched; (b) the test queries; (c) the assessment of test results, and; (d) the generation of benchmark results for comparing the algorithms introduced

in this dissertation. Together with this chapter, these chapters constitute the introductory matter of this dissertation.

Following the introductory matter, Parts II through III define the various components that form the test of the thesis:

Part II applies cooperative database and index compression to the relatively easy domain of a highly redundant database.

Chapter 4 commences this part by defining the index driven Diagonal Assembling Search Heuristic (DASH) sequence search and alignment algorithm. Two different index structures are considered in: (1) Chapter 5, and; (2) the chapters of Part III.

First, in Chapter 5 an initial database format for DASH is devised, and a coarse grained approach to cooperative compression of biological sequence databases, record folding, is presented. This is used to show that the thesis of this dissertation is possible, provided that sufficient redundancy exists in the database being processed.

Part III follows the initial success of Part II, and describes a fine grained cooperative compression scheme, that is intended to be more generally applicable than the record folding approach introduced in Chapter 5. Part III, consists of Chapters 6 and 7, which, respectively, document the NP3 recurrence revealing nucleic acid database compression algorithm, and the NIX algorithm for producing compact companion indices from NP3 encoded databases. Chapter 7 also compares the performance of the NP3 compression algorithm with GeNML in the context of cooperative compression. These chapters show that substantial reductions in index size are possible, along with a modest increase in sensitivity, supporting the thesis of this dissertation. Although it is shown that by batching queries, DASH+NP3/NIX can search a disk-resident database several times faster than NCBI-BLAST can search a memory-resident database, the fine grained approach turns out to be rather computationally intensive, resulting in poor search speed when processing single queries.

The final part of this dissertation, Part IV, consists of a single chapter, Chapter 8, which briefly summarises the results of the preceding parts, draws conclusions, and closes the dissertation with suggested avenues for future research.

Chapter 2

Background

This chapter provides background information on the underlying technologies and methodologies that are relevant to this dissertation. This consists of: (1) an introduction into the similar problems of the pair-wise alignment of nucleic acid sequences and the pair-wise alignment of protein sequences; (2) a survey of existing methods used to accelerate the pair-wise alignment of these sequences; (3) a survey of text compression techniques relevant to database and DNA sequence compression, and; (4) a brief introduction to index construction, compression and maintenance.

2.1 Sequence Alignment

This section presents the context of the research problem by giving an introduction to nucleic acid and protein sequence search and alignment.

2.1.1 Sequence Search And Alignment

2.1.1.1 Bioinformatics And Sequence Similarity Searching

DNA sequences consist of the alphabet `A`, `C`, `G` and `T`, representing the nucleotides that constitute a DNA strand. Such databases may also include additional symbols that repre-

sent ambiguity regarding the identity of a base: These symbols are the *wild-cards*. The most common wild-card is `N`, signifying that the identity of the nucleotide is completely unconstrained. See Table 2.1 for the complete list of wild-cards.

Searching biological sequence databases consists of finding similarities, often in the hope of identifying *homology*, i.e., familial relationship, between a query and each sequence in a database. Similarity is assessed by performing pair-wise alignments of the query against the sequences in the database. This pair-wise alignment problem is closely related to the general string similarity search methods of Levenshtein (1966) and Knuth et al. (1977), such as *weighted string edit distance*.

The edit distance of two strings is the minimal number of operations require to transform one string into another. For example "time" and "money" have an edit distance of 4, because "time" can be transformed into "money" in 4 steps: `time` $\rightarrow$ `timey` $\rightarrow$ `mimey` $\rightarrow$ `momey` $\rightarrow$ `money`. In this case substitutions and insertions were considered as events of equal cost. This can be generalised such that different costs are applied depending on whether each event is a substitution, insertion or deletion, resulting in a weighted edit distance. If substitutions have a cost of 2 while insertions and deletions have a cost of 1, then converting time into money as previously described would have a weighted cost of $1+2+2+2=7$, however by using the steps: `time` $\rightarrow$ `timey` $\rightarrow$ `imey` $\rightarrow$ `mey` $\rightarrow$ `mney` $\rightarrow$ `money` would have a weighted cost of only $1+1+1+1+1=5$. Provided that the costs are formulated such that the comparison of two random strings results in an actual cost, i.e., $cost > 0$, then this method can be used to determine the relative similarity of two strings. It is this property that is used in genomic pair-wise sequence alignment.

Three increasing strengths of sequence comparison used in pair-wise alignment are described below: (a) exact sub-string alignment; (b) non-exact sub-string (non-gapped) alignment, and; (c) non-exact gapped alignment.

Table 2.1: IUPAC-IUB Codes (Joint Commission on Biochemical Nomenclature 1983) And Their 4 bit Representations As Used In This Dissertation.

IUPAC Code	Description	Base(s)	Four Bit Coding
X	No base	-	0000
G	Guanine	G	0001
A	Adenine	A	0010
T	Thymine	T	0100
C	Cytosine	C	1000
R	Purine	A or G	0011
Y	Pyrimidine	C or T	1100
M	Amino	A or C	1010
K	Ketone	G or T	0101
S	Strong interaction	C or G	1001
W	Weak Interaction	A or T	0110
H	Not-G	A, C or T	1110
B	Not-A	C, G or T	1101
V	Not-T	A, C or G	1011
D	Not-C	A, G or T	0111
N	Any	A, C, G or T	1111

2.1.1.2 Exact Sub-String Alignment

Exact sub-string alignment consists of locating contiguous strings of corresponding symbols in both a query and subject sequence. It is *exact*, in that no substitutions are allowed. It is of *sub-strings*, in that an alignment may include only part of each sequence: The entire sequence is not required to be aligned. Consider the following example:

query sequence: CGA CTG ATC TAG

subject sequence: CGT GTA GCT AGC AGT GTA GTC TAG CGT ACG TGC

The sub-string CG (the 1^{st}–2^{nd} letters of the query sequence) can be aligned against the 1^{st}–2^{nd}, 25^{th}–26^{th}, and 29^{th}–30^{th} letters of the subject sequence, as shown below:

CGt gta gct agc agt gta gtc tag CGt aCG tgc

Similarly, TCTAG (the 8^{th}– 12^{th} letters of the query sequence) can be aligned against the 20^{th}–24^{th} letters of the subject sequence:

cgt gta gct agc agt gta gTC TAG cgt acg tgc

2.1.1.3 Non-Exact Sub-String (Non-Gapped) Alignment

This is a generalisation of exact sub-string alignment where substitutions, or "mistakes" are allowed in the alignments. It is *non-exact*, in that these substitutions are allowed, i.e., any one symbol may be substituted for any other symbol. It is *non-gapped*, in that any one symbol must be replaced by precisely one symbol, neither more nor less, since that would result in the introduction of one or more *gaps*. Consider again the example from exact sub-string alignment, but now allow one or two substitutions in each alignment:

query sequence: CGA CTG ATC TAG
subject sequence: CGT GTA GCT AGC AGT GTA GTC TAG CGT ACG TGC

The string CGACT (the 1^{st}–5^{th} letters of the query sequence) can now be aligned against the 1^{st}–5^{th} letters of the subject sequence (substitutions are shown in italics):

CG*t* *g*Ta gct agc agt gta gtc tag cgt acg tgc

Similarly, TCTAG (the 8^{th}– 12^{th} letters of the query sequence) can be aligned against the 3^{rd}–7^{th}, 15^{th}–19^{th} and 20^{th}–24^{th} letters of the subject sequence (substitutions are shown in italics):

cgT *g*TA Gct agc agT *g*TA GTC TAG cgt acg tgc

In practice, various rules are applied to determine what level of similarity is required before an alignment is considered significant. These typically involve a *reward* score for corresponding symbols in the sequences, and a *penalty* score for differing symbols. These are

combined with statistical theory to determine whether a given alignment is statistically significant, or, conversely, if the alignment could be reasonably expected to occur by chance. For instance, comparing `CGACT` with `CGTGT` using reward and penalty scores of 1 and -3 respectively would result in a score of -3 (Table 2.2).

This score can be translated into an expected value, and optionally a *p*-value. Although the particular formula varies among sequence alignment tools, the translation is almost always based on an Extreme Value Distribution (EVD), with a two stage process that first converts a raw score, S, into a normalised score, S', that takes into account the scoring system. Using the equations indicated by Karlin and Altschul (1990), S is translated according to:

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

Where λ is derived from the scoring system, such that $\sum_{i=1}^{n}\sum_{j=1}^{i} p_i p_j e^{\lambda s_{ij}} = 1$, where p_i and p_j are the probabilities of the $i^{th}$and j^{th} symbols of the alphabet, and s_{ij} is the log-odds score of substituting the same symbols. K is a less significant factor that is also derived from the scoring system and is used to correct for the non-random correlation of matching residues in the alignment of similar sequences.

For the scoring system used in our example these compute to $\lambda = 1.37$ and $K = 0.711$. The second stage takes into account the size of the search space to convert the normalised S' into an expected value that indicates the number of alignments with raw score $\geq S$ that would be expected to occur by chance. E is computed according to $E = \frac{mn}{2^{S'}}$, where m and n are the lengths of the query and sequence being searched. Naturally the larger the search space, the larger E will be for a given S', as the increased search space allows more opportunities for matches to occur. Finally, if required, an additional step can be applied to translate the expected value into a *p*-value by computing $p = 1 - e^{-E}$. However this is not usually necessary, if only because $E \approx p$ when $E < 1$. In the case of our example alignment, the full match of `TCTAG` in the 33 base sequence gives a raw score of $S = 5$ (five matches $\times$

Table 2.2: Non-Gapped Alignment Of CGACT And CGTGT.

C	G	*A*	*C*	T
C	G	*T*	*G*	T
+1	+1	-3	-3	+1

reward score of +1), which normalises to $S' = 12.4$, and yielding $E = p = 0.029$. (Note that for these statistics to be appropriate longer sequences are required than are used in this example).

Non-exact sub-string alignment is the type of searching performed by BLAST version 1 (Altschul et al. 1990).

2.1.1.4 Non-Exact Gapped Alignment

Gapped matching introduces the concept of *gaps* caused by insertions or deletions, i.e., substituting a single symbol with either: (a) more than one, or; (b) zero symbols. For example, substituting the A in CGACT with AAG results in the string CGAAGCT. The strings are now of differing lengths, and in order to align the entire strings, gaps must be inserted into the shorter sequence so that the strings are again of equal length. Gaps are usually indicated by a hyphen ("−").

For gapped alignment, *gap creation* and *gap extension* penalties are introduced to supplement the reward and penalty scores associated with non-gapped alignment. This system of scoring gaps is referred to as affine, or sub-linear. Other gap scoring approaches are also used, such as linear, piece-wise linear and logarithmic. Affine scoring is popular as it provides a trade-off between the speed of computation afforded by linear, and the accuracy of the logarithmic model preferred by many biologists.

As a simple example of affine gapping, consider the alignment of CGACT and CGAAGCT, accomplished by adding gaps into the query sequence (Table 2.3). Assuming a gap creation

Table 2.3: Gapped Alignment Of CGACT And CGAAGCT.

C	G	A	-	-	C	T
C	G	A	A	G	C	T
+1	+1	+1	-5	-2	+1	+1

score of -5, a gap extension score of -2, a reward score of +1, and a penalty score of -3, the example above would have a final score of -2.

Non-exact gapped alignment is the type of search performed by PatternHunter (Li et al. 2004), BLAST version 2 (Altschul et al. 1997), FASTA (Pearson 1990) and Smith-Waterman (Smith and Waterman 1981), and most other sequence alignment algorithms.

2.1.1.5 Dynamic Programming

Dynamic programming is generally applied to the gapped sequence alignment problem as it is considered the fastest complete algorithm available (Dwan 2002).

This approach consists of generating a dynamic programming space that has the query and subject sequences as axes. Each discrete cell in the space is then evaluated, relying on the values of cells nearer the origin than itself. If the alignment is global then it is required to include the entirety of both strings, where as for local alignment it is not. Thus CAT=CAT would be a valid local alignment of SCATTER and BOBCAT, where as it would not be a valid global alignment of those strings, because it does not include the entirety of both strings. Whether global or local alignment is being performed, the score of each cell is calculated using a recurrence, such as:

$$L(i,j) = max \begin{pmatrix} max_{k\geq 1}(L(i,j-k) - g(k)) \\ L(i-1,j-1) + w(q_i, s_j) \\ max_{k\geq 1}(L(i-k,j) - g(k)) \end{pmatrix}$$

Where $L(i, j)$ is the score of the cell in the i^{th} row and j^{th} column. The gap penalty function is $g()$, and hence $g(n)$ is the penalty for n consecutive gaps. The substitution matrix, $w()$ is used to compute the appropriate reward or penalty score for the pair of letters q_i and s_i which are the i^{th} letter of the query and subject sequence, respectively.

This recurrence then, selects the maximally scoring option among, respectively: (a) the cell in the same row that maximises the sum of the score and the gap penalty from the current cell; (b) the score of the cell corresponding to the previous position in both the query and subject sequence plus the reward or penalty score from the letter in the current position of the query and subject sequences, which will either be identical (yielding a reward), or different (possibly yielding a penalty); or (c) the cell in the same column that maximises the sum of the score and the gap penalty from the current cell. If the alignment is required rather than just the optimal score, then the option taken must be recorded for each cell. This allows the discovery of the path that led to the optimal score, and thus the corresponding alignment.

Note that options (a) and (c) require iterating through the corresponding row or column of cells in order to find the maximum score. However, if the gap penalty function is strictly increasing, i.e., $g(k+1) > g(k) \forall k > 0$, then this can be resolved to a single look up operation in exchange for remembering one value per row and column. Thus the evaluation of each cell requires O(1) time, and thus discovery of the optimal alignment requires O($m \times n$) time, where m and n are the lengths of the query and subject sequences.

The dynamic programming process is illustrated in Figure 2.1, where the cell at (i, j), marked in grey, is to be evaluated. Only the cells with solid borders need be evaluated, i.e, the current row and column, and the cell corresponding to the immediately preceding letters of the query and subject sequences, i.e., the cell at $(i-1, j-1)$. If we assume that $g(n) = 3+2n$, and that $w(q_{i,}s_j) = 1$, then we can determine the optimal score for the cell (i, j), and record the back trace information for that cell.

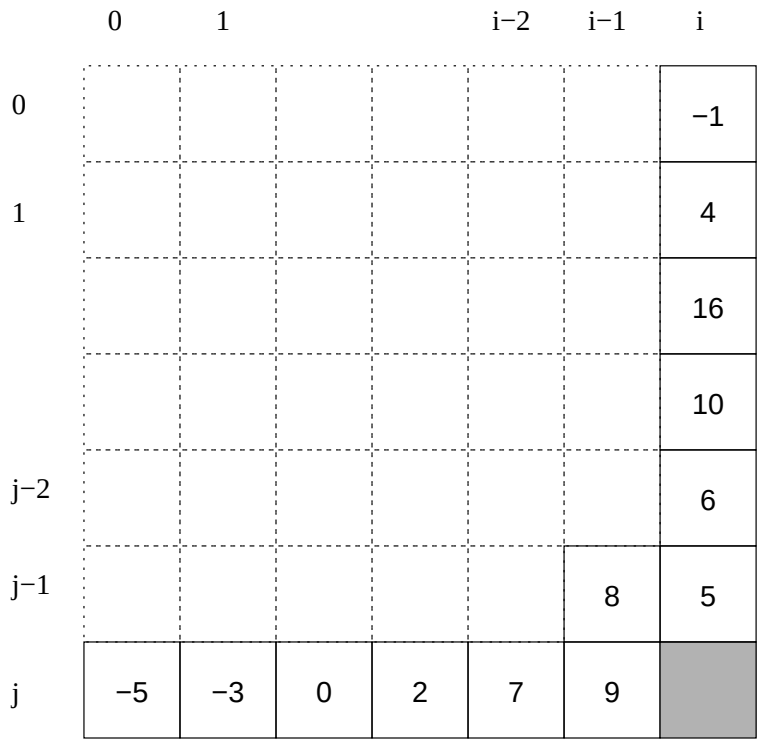

Figure 2.1: Example Of Dynamic Programming Evaluation.

The back trace information for each cell is just the coordinates of the cell that led to the current cell, i.e., one of:

$$\begin{aligned} &(i-k, j) && \text{where} \quad i > k \geq 1; \\ &(i, j-k) && \text{where} \quad j > k \geq 1; \text{ or} \\ &(i-1, j-1). \end{aligned}$$

Taken as a whole, the back trace information for a dynamic programming space is sufficient to determine an optimal path, i.e., least cost path, from the origin to any cell in the space by inspecting the back trace for the end cell, and then recursively consulting the back trace of that cell, and so on until the start of the path is found. These processes are now explained using the example of Figure 2.1.

Considering first the j^{th} row, the possibles scores are computed according to $L(i,j) = max_{k\geq 1}(L(i-k,\, j) - g(k))$. The possible values for k are the integers $1 \ldots i-1$, which results in possible scores of:

$$\begin{aligned}
L(i-1,\, j) - g(1) &= 9-5 &&= 4 \\
L(i-2,\, j) - g(2) &= 7-7 &&= 0 \\
L(i-3,\, j) - g(3) &= 2-9 &&= -7 \\
L(i-4,\, j) - g(4) &= 0-11 &&= -11 \\
L(i-5,\, j) - g(5) &= -3-13 &&= -15 \\
L(i-6,\, j) - g(6) &= -5-15 &&= -20
\end{aligned}$$

Thus, the optimal selection from the i^{th} column is $L(i-1,\, j) - g(1) = 4$.

The score from using the second equation of the recurrence can be calculated as:

$$L(i,j) = L(i-1,\, j-1) + w(q_i, s_j) = 8+1 = 9$$

Finally, the optimal selection from the i^{th} column by using $L(i,j) = max_{k\geq 1}(L(i,\, j-k) - g(k))$:

$$\begin{aligned}
L(i,\, j-1) - g(1) &= 5-5 &&= 0 \\
L(i,\, j-2) - g(2) &= 6-7 &&= -1 \\
L(i,\, j-3) - g(3) &= 10-9 &&= 1 \\
L(i,\, j-4) - g(4) &= 16-11 &&= 5 \\
L(i,\, j-5) - g(5) &= 4-13 &&= -9 \\
L(i,\, j-6) - g(6) &= -1-15 &&= -16
\end{aligned}$$

This determines that the optimal selection from the i^{th} column is $L(i,j) = L(i, j-4) - g(4) = 5$. The recurrence can now be evaluated as:

$$L(i,j) = max \begin{pmatrix} 4 \\ 9 \\ 5 \end{pmatrix}$$

As 9 is greater than either 4 or 5, the second line of the recurrence is used, and the cell at (i,j) will be given the score 9, and the back trace information will indicate that the alignment came from $(i-1, j-1)$. This process is repeated for every cell in the dynamic programming space. In this way, the path taken to reach any cell can be determined by tracing backwards to reveal the optimal alignment of the two sequences. Such alignments can be either *global* or *local*. A *global alignment* is an alignment that must include the entirety of each sequence. In contrast, a *local alignment* involves only a portion (substring) of one or both sequences.

Consider also the example of Table 2.4 where the dynamic programming grid has been evaluated for the sequences `CGACT` and `CGAAGCT`. The fifth cell in the second row (value -7) is evaluated by finding the maximum score that is possible by considering the cells immediately left, above and above-left. Assuming the scoring system of the previous example, this would be:

$$\begin{aligned} score &= max \begin{pmatrix} -5 + gapextend & = & -5-2 & = & -7 \\ -10 + gapcreate & = & -10-5 & = & -15 \\ -8 + reward & = & -8+1 & = & -7 \end{pmatrix} \\ &= -7 \end{aligned}$$

If a global alignment is being performed, then the end of the alignment would be the bottom right cell in the dynamic programming space, i.e., the cell corresponding to the last letter

Table 2.4: Evaluated Dynamic Programming Space For CGACT And CGAAGCT. The origin of the space is at the top left of the table. The back trace information is indicated by the arrows in each cell that point to the next (previous) cell that is in the least cost path to the cell. Super scripts indicate the distance to that cell when it is not 1. In some cases the minimum cost can be achieved by multiple paths. In those cases arrows to each path are shown.

	C	G	A	A	G	C	T
C	1	$\leftarrow$-4	$\leftarrow^{2}$-6	$\leftarrow^{3}$-8	$\leftarrow^{4}$-10	$\leftarrow^{5}$-12	$\leftarrow^{6}$-14
G	$\uparrow$-4	$\nwarrow$2	$\leftarrow$-3	$\leftarrow^{2}$-5	$\nwarrow\leftarrow^{3}$-7	$\leftarrow^{4}$-9	$\leftarrow^{5}$-11
A	$\uparrow^{2}$-6	$\uparrow$-3	$\nwarrow$3	$\nwarrow\leftarrow$-2	$\leftarrow^{2}$-4	$\leftarrow^{3}$-6	$\leftarrow^{4}$-8
C	$\uparrow^{3}$-8	$\uparrow^{2}$-5	$\uparrow$-2	$\nwarrow$0	$\nwarrow$-5	$\nwarrow$-3	$\leftarrow$-8
T	$\uparrow^{4}$-10	$\uparrow^{3}$-7	$\uparrow^{2}$-4	$\nwarrow$-5	$\nwarrow$-3	$\nwarrow$-8	$\nwarrow$-2

Table 2.5: Example Of Local Sequence Alignment.

C	G	A
C	G	A
+1	+1	+1

in the query and subject sequences. However, if local alignment was being performed, then the highest scoring cell in the dynamic programming space would be used as the starting point. Also for local alignment, the calculation of cell scores would be adjusted so that no cell may score below zero, thus ensuring the highest scoring alignment is not concealed because it is preceded by a negatively scoring alignment.

For Table 2.4, a locally optimal path is obtained by finding the highest scoring cell, and tracing the path from that cell to the lowest scoring cell on that path. This results in the path from the cell scoring 3 of (A,A) $\rightarrow$ (G,G) $\rightarrow$ (C,C). This path is then reversed, and the query and subject sequence fragments are collated, to obtain the alignment depicted in Table 2.5. The globally optimal path can be obtained similarly by starting at the bottom right cell and following the path to the origin. This yields the back trace path of (T,T) $\rightarrow$ (C,C) $\rightarrow$ (--,G) $\rightarrow$ (A,A) $\rightarrow$ (G,G) $\rightarrow$ (C,C), yielding the global alignment of Table 2.6.

Table 2.6: Example Of Global Sequence Alignment.

C	G	A	-	-	C	T
C	G	A	A	G	C	T
+1	+1	+1	-5	-2	+1	+1

Table 2.7: Example Of Proteomic Sequence Alignment Using A Substitution Matrix.

I	F	G	M	M	R	C
I	S	S	M	M	Q	C
+4	-2	0	+5	+5	+1	+9

2.1.1.6 Proteomic Similarity

The searching techniques above are also applied to proteomic databases. The four letter nucleotide alphabet, A, C, G and T, is replaced by an alphabet of 22 amino acids[1]. The relative properties and frequencies of the amino acids are more complex than for nucleotides. To accommodate this, the fixed reward and penalty scores are discarded in favour of a substitution matrix that assigns differing scores for each pairing of amino acids. Such matrices take into account the biological relatedness of the different amino acids, e.g., hydrophilic versus hydrophobic. Two popular families of substitution matrices are BLOSUM (Henikoff and Henikoff 1992) and PAM (Dayhoff et al. 1978, Schwartz and Dayhoff 1978). Table 2.7 shows a simple alignment of two peptides using the BLOSUM62 substitution matrix. It can be seen that while identical residues score most highly, conservative substitutions, e.g., R versus Q, can also produce positive scores. There necessarily remains a substantial number of combinations that result in a negative score, e.g., F versus S, which scores -2.

Since each amino acid is encoded by a *codon* each consisting of three nucleotides[2] it is possible to search for proteins in nucleotide sequences by *translating* to and from the nucleotide to amino acid domain. Because some amino acids are encoded by any of up to

[1]This includes the relatively recently discovered natural encoding of selenocysteine (Bock et al. 1991) and pyrrolysine (Srinivasan et al. 2002) .

[2]Excluding the longer encoding factors introduced by selenocysteine.

six different codons, such searching introduces new complexities as multiple distinct nucleotide sequences can encode the same protein. Translated searching is not given further consideration in this dissertation.

2.1.1.7 Biological And Statistical Significance Of Alignments

Of equal importance as finding the alignment between a pair of sequences, is giving the alignment a score that correctly indicates its relative biological significance. The difficulty is what is "biologically significant"? For any given query, NCBI & WU BLAST differ in the alignment and scores they return, and these are different again compared with the alignments and scores returned by the Smith-Waterman algorithm (Dwan 2002), or by other alignment scoring metrics, such as POZ scores (Booth et al. 2004). These algorithms uses statistical significance to estimate the biological significance. However, the precise interpretation of biological significance depends heavily on the context (Galisson 2000). That statistical significance is a valid approximation of biological significance is argued by Smith, Waterman & Burks (Altschul and Gish 1996b). Therefore, statistical significance remains the dominant mechanical approximation for biological significance.

Work by Altschul et al. (1997) and Galisson (2000) provides some commentary on the progress made in the statistical issues of biological sequence searching. Smith et al. (1985) modelled the distribution of nucleic sequence similarities by applying the work by Erdos and Renyi (1970), and Erdos and Revesz (1975) on finding the longest run of heads in a series of coin tosses by considering nucleotide matches as "heads" and substitutions as "tails". Their theory is ideally applicable to non-gapped alignments with some applicability to gapped alignments. A similar work that applied only to exact alignments was also undertaken by Karlin et al. (1983). Altschul (1997) and Karlin and Altschul (1993) have also shown that it is possible to use these results to estimate the combined significance of multiple nearby alignments. More recently, Booth et al. (2004) introduced POZ-scores as

another scoring method that is more resistant to false positives, but may also increase the false negative rate.

These theories provide an accurate statistical model for non-gapped alignments. Their common purpose is to provide a mechanism to convert between dynamic programming scores to normalised scores (Altschul and Gish 1996a). Such normalised scores are often measured in "bits" of information by rescaling to base 2 logarithms (Altschul 1993, 1991, Altschul et al. 1997). Normalised scores allow the calculation of the probability or expected value for alignments of any given score, and hence objectively evaluate the statistical significance of any given alignment.

Unfortunately, to date no corresponding theory for gapped alignments has been proved (Galisson 2000, Altschul et al. 1997). However, the non-gapped theories are generally accepted as valid for gapped alignments, subject to certain caveats. These caveats usually relate to the selection of appropriate gap penalty regimes. The practical validity of applying the non-gapped theories to gapped alignments is assumed by both the FASTA and BLAST 2 algorithms (Altschul et al. 1997, Galisson 2000). Arguments in support have also been made by various authors at various times, e.g., Altschul and Gish (1996a,b), Collins et al. (1988), Mott (1992), Waterman and Vingron (1994), Pearson (1998), Altschul et al. (1997) and Booth et al. (2004).

However, due to the lack of a formal theoretical model, the values of various parameters must be estimated by simulation with either random data or unrelated sequences (Altschul et al. 1997, Galisson 2000, Pearson 1998). A particular difficulty with these statistical estimates for gapped alignments is if the composition of the sequences differs substantially from that of the data used to calculate the parameters. In that case the accuracy of the estimations suffers (Galisson 2000).

As previously mentioned, the relatedness of these statistical models of significance to biology are considered by Altschul et al. (1994) and Pearson (1998), who, while conceding

that the two are not identical, argue that they are necessarily similar and related. The continued use of models of statistical significance as an estimation of biological significance is testimony to their general suitability to their application in the absence of a better approach.

2.1.2 Comparing The Performance Of Sequence Search And Alignment Algorithms

Difficulties, however, arise when comparing different sequence search and alignment algorithms. Since there is no objective computational method to assess biological significance, approximations must be made. Two approximations are: (a) to use human judged results, and; (b) to use mechanically judged results, i.e., refer to the results of a trusted algorithm — a so called "benchmark" (which probably ultimately depends on statistical significance).

2.1.2.1 Human Judgement

The first approach is to use human judgements. This approach is popular in the information retrieval community, e.g., the Text REtrieval Conference (TREC[3]) community. It has also been applied to biological sequence searching by making use of classified sequences, such as in some protein databases, for example by Williams and Zobel (2002a). However, there are inherent difficulties with using human judgements.

The first difficulty is the presence of misjudgements that penalise correct judgements made by computer programs. There is evidence to suggest that misclassifications in human supplied judgements may occur in a worryingly high percentage of cases, perhaps 10 percent or more (Bernstein and Zobel 2005). Having said this, the misclassification rate is not usually sufficient to invalidate comparisons of search and alignment systems, as the difference in effectiveness of the systems will be much greater, as is demonstrated in this dissertation.

[3] `http://trec.nist.gov` [On line; accessed 17 February 2007]

The second and greater difficulty is that the number of human supplied judgements is usually very small compared to the collection sizes. Moreover, there are no well distributed human judged test collections for nucleotide sequence alignment.

2.1.2.2 Benchmarks

An alternative to using human judged results, is to compare each competitor to some benchmark. A good candidate for a benchmark in nucleic acid and protein sequence alignment is the algorithm of Smith and Waterman (1981). This is because its search process is both exhaustive, and well trusted. Nonetheless, there are two potential pitfalls that must be addressed.

Firstly, the Smith-Waterman algorithm has no mechanism for filtering results that are statistically significant, but biologically insignificant. This will penalise any algorithm that does filter such uninteresting results. This bias can be reduced by disabling any low-complexity filters in other algorithms (most supply the option to do this).

Secondly, because the Smith-Waterman algorithm is designed to optimise recall, it will implicitly penalise algorithms that are designed to optimise precision. This would cause precision-oriented systems to score lower than they would if assessed by a human judged framework.

Nonetheless, it seems that unless the residual electronic misjudgements and biases occur more often than in the human judged case, the electronic judgement approach is superior, because it can be applied to arbitrary collections without requiring the collection of human judgements: Slow as the Smith-Waterman algorithm is, it is still much faster (and easier) to obtain results for a large collection than to solicit a similar number of results from human experts. Further, since most heuristic sequence alignment algorithms are derived from the Smith-Waterman algorithm, there is a certain fairness and common sense in comparing the heuristic algorithms to their ancestor.

Nonetheless, the fundamental limitations and blind spots will apply that arise whenever evaluation is at an engineering level, processing level and output levels only, such as not extending to evaluate use or social issues such as fitness for use or differences in productivity (Saracevic 1995). Such social, user and use evaluation is beyond the scope of this dissertation.

2.1.2.3 Sensitivity Metrics

Irrespective of the origin of the benchmark against which competitors may be compared, a metric must be selected in order to perform the comparison. A traditional measure in information retrieval is *Recall and Precision*[4]. This double measure compares both the sensitivity and selectivity of a query. *Recall* measures the extent to which the results of a query contain the correct results. Conversely, *precision* measures the proportion of returned results that are correct. Therefore returning entirely relevant results will result in high recall and precision. Omitting some relevant results will cause the recall to drop, but will not affect the precision, where as including some irrelevant results will cause the precision to fall, but leave the recall measure unchanged.

In sequence alignment applications, it is trivial to limit the number of results returned. In that case, a reduction in precision does result in a corresponding drop in recall, as relevant results are displaced by irrelevant ones. Therefore if the number of results each competitor returns is fixed, then recall and precision are necessarily connected by a positive correlation. Consequentially, a metric based on recall will capture much of the value of measuring both recall and precision if the number of results is limited.

Favouring recall over precision (and using the measures of recall and precision at all) also has broader implications in terms of usability, partly because human assessed relevance is, like many aspects of humanity, practically impossible to define or compute, partly because

[4]Recall and Precision are isomorphic with the alternative approach of considering false positive and false negative rates that is perhaps more common in the field of biology.

its assessment is highly variable among subjects (Saracevic 1995). Thus, the variability in opinion that leads to many of the misclassification by human judges also introduces a similar type of error in that recall is but one opinion of relevance.

In nucleic acid and protein sequence alignment there is an added complication in that a result can be *partially* returned, e.g., if only part of an alignment between two sequences is discovered (possibly as multiple fragments). One solution is to sum the fractional alignments. However, this assumes that the majority of an alignment is of less value than its entirety, and therefore will underestimate the value of the results. Alternatively, each alignment could be scored as though the entire alignment were reported — thus overestimating the value of the results.

Li et al. (2004) employ the metric of counting all alignments that score $> \frac{x}{n}$, where x is the score of the optimal alignment, and n typically being 2. This method of measuring recall can be called the *PatternHunter* metric, after the title of the algorithm presented by Li et al. (2004). It is appealing because it is not only simple, but also intuitive, in that it compromises between the under- and over- estimations previously described. A variation on this metric is introduced later as the basis for measuring search sensitivity in this dissertation.

2.2 Accelerating Sequence Searching

This section provides a survey of selected work in several major areas of accelerating the nucleic acid and protein sequence search processes. The survey includes: (a) heuristic algorithms; (b) clustered (parallel) computing, and; (c) indexing techniques.

2.2.1 Heuristic Algorithms

The current heuristic algorithms typically claim two to three orders of magnitude improvement in execution time compared to Smith-Waterman (Altschul et al. 1997, Williams and Zobel 2002a, Kent 2002). However, as observed by Pearson (1990) and Galisson (2000), this improvement is at the expense of accuracy. That is, heuristics offer speed improvements by excluding much of the search space. Therefore, if any interesting results lie in the excluded regions of the search space, they will not be identified.

2.2.1.1 A Brief Comparison Of Selected Heuristic Algorithms

The algorithm by Smith and Waterman (1981) is exhaustive, i.e., does not exclude any regions of the search space, and this makes it helpful as a base line by which to measure both the performance and quality of heuristic algorithms (Dwan 2002). This is reinforced by the fact that the majority of the heuristic algorithms are directly or indirectly derived from the Smith-Waterman algorithm. Table 2.8 compares the performance of Smith-Waterman and the algorithms discussed below.

2.2.1.2 FASTA

The FASTA (Pearson 1990) group of algorithms, developed during the 1980s, are generally considered to be among the best quality heuristic algorithms. Despite being the among the most sensitive heuristic algorithms, significant sensitivity compromises are still made (Galisson 2000). The primary interest of FASTA in this dissertation is that it is the direct ancestor of the widely used BLAST family of algorithms.

These algorithms function by finding all *k*-mers, that is strings of length *k*, that correspond between the query and subject sequences. This list of *k*-mers is then used to identify and re-score the ten best scoring regions. A joining procedure is applied to these regions and

finally band limited dynamic programming is applied to optimise the final score. This algorithm finds both gapped and non-gapped similarities, and offers 10 to 100 times speed improvement compared to the Smith-Waterman algorithm.

2.2.1.3 BLAST

The first version of the BLAST algorithm (Altschul et al. 1990) searches for non-gapped alignments only. This makes it somewhat faster and somewhat less sensitive than the FASTA algorithm. The loss of sensitivity is partly offset when searching for protein sequences, because BLAST takes into account the similarity of amino acids in the initial phase of the search (Galisson 2000): rather than using only the *k*-mers of the query sequence, when searching for protein sequences BLAST also includes all neighbouring *k*-mers, i.e., *k*-mers that are similar to each query *k*-mer. This expanded list of *k*-mers is then used to look up each exact occurrence in the subject sequence to perform an un-gapped alignment. The highest score for each extension is retained.

The BLAST algorithm was later refined to incorporate more advanced statistical methods, e.g., Altschul (1993, 1991), Karlin and Altschul (1990), Dembo et al. (1994), including using Poisson approximations and taking into account multiple nearby alignments when calculating the expected value or probability of a match occurring in a random sequence.

The ability to perform gapped searches with BLAST family algorithms was later included by both Altschul et al. (1997) and Gish resulting in NCBI-BLAST 2 and WU-BLAST 2 respectively.

NCBI-BLAST 2 claims to be $>$ 100 times faster than Smith-Waterman (Altschul et al. 1997). This performance improvement is arguably at a notable sensitivity cost compared to BLAST 1 (Williams and Zobel 2002a), although the authors of NCBI-BLAST 2 claim that sensitivity is actually improved (Altschul et al. 1997).

The NCBI BLAST 2 nucleotide search algorithm differs in several substantial ways from the NCBI BLAST 2 protein search algorithm. First, neighbouring *k*-mers are used during protein searching to increase sensitivity, where as for nucleotide searching they are not. The use of similar as well as identical *k*-mers as baits contributes significantly to the sensitivity of BLAST when searching protein databases. Second, the default *k*-mer size is three for protein while it is at least seven and defaults to eleven for nucleotide searches. Third, whereas in order to contain the time required to perform a search, two hits must occur in close proximity to trigger the extension of a hit during protein searching, a single hit is sufficient to trigger extension of a hit during a nucleotide search.

NCBI-BLAST 2 operates similarly to the original BLAST algorithm, first discovering all corresponding *k*-mers (again using neighbouring *k*-mers for protein searches) between the query and subject sequence(s), with each hit being extended using non-gapped alignment, and the highest scoring alignments being extended using a variation of the Smith-Waterman algorithm that allows gapped alignment.

NCBI-BLAST 2 takes time proportional to the product of the query and subject sequences to scan the database, plus an additional factor proportional to the total length of hits due to the dynamic programming extension performed for each hit. This can cause NCBI-BLAST 2 to search slowly if the database contains many sequences similar to the query, or if the query contains low complexity regions that are well known to occur with excessive frequency. This problem is addressed in part by masking the query against low complexity regions using an algorithm such as XNU (Claverie and States 1993) or DUST (Hancock and Armstrong 1994). However, the performance issue still remains for long high-quality alignments, resulting in overall performance which is approximately proportional to the total length of alignments returned, rather than the size of the query. Consequentially, it is difficult to predict in advance the time required to complete a BLAST search.

Moreover, because for BLAST builds an index of the query sequence, but not of the database being searched[5], searching is intrinsically slower than if the database were indexed instead. The trade-off is that the size of a database prepared for searching by BLAST will be very small compared with the algorithms that index the database.

The other variation of the BLAST algorithm, WU-BLAST 2, was released prior to NCBI-BLAST and makes even greater claims of speed and sensitivity improvement than does NCBI-BLAST 2. WU-BLAST also offers NCBI-BLAST 1 & 2 compatibility modes. The actual algorithm has, unfortunately, not been released. Refer to Galisson (2000) for a more thorough explanation and comparison of the FASTA and the NCBI- and WU- variants of the BLAST family of algorithms.

2.2.1.4 BLAT

BLAT (Kent 2002) is a more recent heuristic algorithm that uses a memory resident index of the database to increase speed rather than sensitivity. This combination of RAM-resident index and disk-resident database enables BLAT to be run at great speed on inexpensive computers. It is approximately five times faster than NCBI-BLAST 2. Its sensitivity for nucleic acid queries is somewhat less than that of NCBI-BLAST 2, while protein sensitivity is substantially lower, being hampered by the use of a rigid index, as will be shown later in this dissertation. However, for its original application of genome assembly and annotation the sensitivity is more than acceptable, particularly given the speed increases obtained.

BLAT operates by creating an index of all non-overlapping *k*-mers in the database that for many databases is small enough to fit in the RAM of a desktop computer. It is because the index is constructed for only non-overlapping *k*-mers that sensitivity is sacrificed compared with BLAST. On the positive size, BLAT stitches together un-gapped alignments to form the gapped alignments, including exons that are separated by gaps. This effectively avoids

[5]According to `http://blast.wustl.edu/blast/dbfmts.html`, the NCBI BLAST database format indexes the names of each sequence, but not their content [On line; Accessed 25 October 2008].

much of BLAST's problem of expending up to O(n^2) effort to discover an alignment of length n, resulting in a search time that is more readily predicted based on the product of the length of the query and database.

2.2.1.5 FLASH

FLASH (Califano and Rigoutsos 1993) is an earlier example of indexed sequence searching, where the intention was to maintain or improve sensitivity as well as speed. FLASH creates a probabilistic index that consists of a hash-table that contains not only the *k*-mers present in the indexed databases, but also all similarly ordered permutations of each *k*-mer. This allows the index to look up all sequences that contain similar strings to the query, instead of only looking up identical strings.

This greatly aids sensitivity, and results in a system that is more sensitive than BLAST. In addition, FLASH is an order of magnitude faster than BLAST. Its success is, however, hampered by the size of the indices it uses. The storing of multiple permutations of each *k*-mer results in indices that are two orders of magnitude larger than the input sequence collections (Williams and Zobel 2002a).

2.2.1.6 CAFE

CAFE (Williams and Zobel 2002b, Williams 1999, Williams and Zobel 1997b,a, 1996) is an example of a later indexed algorithm that utilises much smaller indices than FLASH, and offers similar sensitivity to BLAST 1 or FASTA.

CAFE is intended to operate on disk-resident databases and indices, and utilises many of the techniques used to search disk-resident collections of text. This includes a strong emphasis on compressing index and data records to minimise their retrieval time.

The CAFE search algorithm consists of two stages, a first coarse search that involves only the index, and a second fine search stage that involves the records, or relevant parts of records where they are large, thus further reducing retrieval time.

The CAFE index stores not only which records contain a given *k*-mer, but also the offset of the occurrence(s) of that *k*-mer in the record. It is this feature that allows CAFE to perform the coarse search without the expense of retrieving compressed records from the database. This is because the *k*-mer is known from the query sequence.

A number of scoring and ranking schemes can be applied to the collection of *k*-mers revealed by the index to occur in any given record.

A naive scheme that works well is to simply count the number of matching *k*-mers. The authors call this FRAMECOUNT. Another scheme that works well is to consider the coverage of the record by the *k*-mers. In that scheme, two adjacent *k*-mers would score less than two *k*-mers that did not overlap. The authors call this COVERAGE. However, better results are obtained by combining these two measures into a COMBINED score, according to:

$$\text{COMBINED} = \text{COVERAGE} - k \times (\text{LENGTH} - \text{COVERAGE})$$

Where k is an empirically determined constant, typically selected such that $k \ll 1$.

Initially, only hits are combined that occur in the same frame, i.e., a fixed difference between the query and subject sequence offsets. For example, (5,7) and (6,8) would be in the same frame, because 5-7=-2 and 6-8=-2. However, (8,11) would not be in that frame because $8-11=-3\neq-2$.

Hits in plausible combinations of frames may correspond to alignments containing gaps. The scores of such frames are combined according to $\sqrt[d]{S}$, where S is the score of the

frame, and *d* is the distance of the frame that is being added to the frame that is receiving the addition.

In the fine search stage, CAFE retrieves the top ranking records from the database, or the relevant sections of them, if they are long. They are then subjected to a FASTA like assembly and extension procedure.

An attractive feature of the CAFE algorithm is the sub-linear increase in search time as the database size increases. This occurs, in part because: (a) the coarse search does not need to retrieve records from the database, thus decoupling search time from database size, and; (b) unlike in NCBI-BLAST, the fine search occurs only once for each hit, even if it contains gaps.

The authors note that given the continuing upward trend of database sizes that this effect will result in a shift over time towards indexed algorithms as the only practicable solution. It is also notable that CAFE search times are more reliably predicted by search space size when compared to BLAST 2 (Williams and Zobel 2002a). This predictability of computation expense is an attractive feature for operators of sequence search and alignment services.

2.2.1.7 Acceptance Of Heuristic Algorithms

Perhaps the most famous and widely used of the heuristic algorithms is NCBI-BLAST 2 (Altschul et al. 1997). Galisson (2000) observes that the prevalent use of this algorithm is evidence that the accuracy trade-off made by heuristic algorithms, and by BLAST specifically, is a tolerable one. Suggested reasons for the acceptability of the trade-off are both pragmatic, i.e., the NCBI BLAST web site is fast, reliable and easy for biologists to use, and the habit is now firmly established; and intelligent, in that BLAST must in general find results that are helpful — otherwise it would not be used. Thus, it seems reasonable to suggest the heuristic algorithms are popular: (a) because they are cheap, i.e., they require

Table 2.8: Inferred Relative Speed Of Various Heuristic Sequence Similarity Search Algorithms.

Algorithm	Gaps	Claimed Speedup vs S-W
Smith and Waterman (1981)	Y	1.00
FASTA (Pearson 1990)	Y	10 to 100 (Pearson 1990)
BLAST 1 (Altschul et al. 1990)	N	~30-300 (Altschul et al. 1997)
NCBI-BLAST 2 (Altschul et al. 1997)	Y	~100-1000 (Williams and Zobel 2002a)
WU-BLAST 2*	Y	~100-3000** Galisson (2000)
BLAT (Kent 2002)	Y	~1000-10000 (Kent 2002)
FLASH(Califano and Rigoutsos 1993)	Y	~1000-10000 (Williams and Zobel 2002a)
CAFE(Williams and Zobel 2002a)	Y	~800-8000 (Williams and Zobel 2002a)
* The unpublished algorithm for WU-BLAST 2 is discussed in Galisson (2000). ** Figures inferred from what information could be found regarding WU-BLAST		

no special hardware; (b) because they are fast; (c) because they are reasonably reliable; and, (d) because they are accessible.

2.2.1.8 Ignorance Of Users To Specific Heuristic Trade-Offs

However, a significant risk with heuristic algorithms lies in the fact that the users of them are frequently not familiar with the precise concessions to accuracy made by a particular heuristic. Therefore many people apply heuristic algorithms to particular problems, without understanding whether it is reasonable or appropriate. This issue of applicability has been explored, e.g., by Galisson (2000) and Dwan (2002), however it is unlikely that most regular users of BLAST or other heuristics are familiar with these issues. This problem is likely to escalate, as more aggressive heuristics are devised to combat the continuing difference between Moore's Law and biological sequence database sizes: Slower and more accurate heuristics such as FASTA are already being abandoned by users (Williams and Zobel 2002a).

Perhaps relief from this problem will emerge from index based algorithms such as CAFE that provide sub-linear search time with respect to database size, without significant additional sacrifice of accuracy. The inefficiencies introduced by the size of the indices will become less significant as databases continue to grow to the point where they will not fit into the main memory of even a large computer or cluster (Williams and Zobel 2002a).

2.2.2 Clustering (Parallel Computing)

Clustering, also known as Grid Computing, is a method of parallel computing where a work load is divided into a number of parts that are processed in parallel on a number of computers. Clustering is a popular method of improving the performance of search systems, as evidenced by their proliferation, particularly in the case of internet search engines. Perhaps the most prominent cluster in popular use for sequence alignment is the NCBI BLAST online search facility. Their attraction is in increasing search throughput by typically 1 – 3 orders of magnitude, without sacrificing flexibility in which algorithms can be utilised.

Within clusters there are two main divisions: *homogeneous* and *heterogeneous*, referring to the level of similarity of the component nodes. Homogeneous clusters are still by far the most common, due to ease of implementation and use, particularly for parallel applications requiring low latency inter-communication. However, as desktop computer capabilities have increased by several orders of magnitude over recent years there is a vast latent resource available in most organisations and departments. The challenge in harnessing this resource is the often heterogeneous nature the computing resources. Thus specialised applications are required to use such computers as an efficient cluster. Also, the maintenance of this type of cluster is typically more difficult as the nodes have more than one (potentially conflicting) function. Despite these difficulties the heterogeneous cluster approach has been successfully employed in such projects as SETI@Home (Anderson et al. 2002, Werthimer et al. 2001).

While the implementation and running costs of clusters may be relatively high (many computers must be configured, powered, and maintained), the design cost and associated risks are relatively small. Homogeneous clusters, in particular, are well understood and can be constructed relatively quickly and reliably. These characteristics of high running costs and low design risk contrast with hardware acceleration techniques, which may be cheaper to run, but involve more risk in the design phase.

2.2.3 Indexing

There are two principle and related issues to consider when designing a search system that exhibits sub-linear search time with respect to database size: space and time. It is often straight forward to reduce the effect of one, or the other: The difficulty comes in reducing both simultaneously, a problem perhaps epitomised by the FLASH algorithm (Califano and Rigoutsos 1993) which decreased search times by two orders of magnitude, but increased disk space requirements by two orders of magnitude. Generally, time is controlled by the effective structuring of the data to be searched, typically by indexing, while space is controlled via compression. However, structured data takes space, and decompression takes time, and so the use of both must be judicious. While these concerns apply to a much broader range of problems, in the following discussion the focus is on nucleic acid and protein search systems.

The effect of indexing on search systems is intuitively easy to understand. Indices occur in many day to day situations. For example, telephone directories are a list of telephone numbers and addresses, indexed by the subscribers names. The value of such a indexing of the data is profound. To illustrate this, consider a telephone directory that was not sorted at all. The average consultation of the telephone directory would require reading half of the entire directory. While this might be practical for small collections of telephone numbers, such as a short list of the telephone numbers of friends and family, it is a less suitable

approach to apply to the telephone directory of even a moderate sized town, and certainly useless for a large city.

Indexing, therefore, is a technique that makes it possible to avoid the cost of an exhaustive linear search, and can result in tremendous speed increases (Witten et al. 1999). However, to make indexing cost effective for large collections, the index structures must be compressed (Witten et al. 1999), an approach that has been proved for biological sequence search and alignment by Williams and Zobel (2002a) in their CAFE system. Construction of compact indices is given further attention in Section 2.4.

2.2.4 Summary

The current state of the art can be described as heuristic software algorithms dominating the market. NCBI-BLAST holds the predominant position in that sector. Algorithms have been proposed, such as BLAT (Kent 2002) and FLASH (Califano and Rigoutsos 1993), which provide significant speed up versus BLAST, but with non-trivial sensitivity or space costs. However, with the relentless increase in data volumes continuing to outstrip Moore's Law, the only long term solution remains the construction of search systems that exhibit sub-linear time and space cost versus data volume. The best candidates to accomplish this are those using compressed indexed databases, such as CAFE (Williams and Zobel 2002a, Williams 1999, Williams and Zobel 1997b).

2.3 Compression

2.3.1 Introduction

Where as indexing is a technique that, when correctly applied, has the effect of substantially reducing the time required to perform a search, data compression is the reduction of the space required to store a given set of data.

Compression can speed up the searching of large databases by reducing the number of accesses that are required to high latency storage devices, such as disks. Moreover, compression may allow the entire database to be stored in a lower latency storage medium, e.g., in RAM rather than on disk. Moving a database to a lower latency storage medium can decrease search times by orders of magnitudes. This is why compression is of interest to sequence search and alignment, and especially so for indexed sequence search and alignment where the raw uncompressed structures may be too large to fit into RAM.

The remainder of this section provides an overview of text compression, giving consideration to the issues that arise when compressing a random access database, and with a final focus on DNA compression and index compression.

2.3.2 Entropy Coding Methods

The major ground work in information theory was laid by Shannon (1948), Weaver and Shannon (1949) and Huffman (1952). A significant contribution of Shannon was to establish the notion of *entropy* in the field of data compression. The entropy of a message, H, is calculated by summing the probability of each symbol in a given alphabet a of size n, weighted by its probability:

$$H = -\sum_{i=1}^{n} \left[p(a_i) \log p(a_i) \right].$$

Hence a symbol, a_k, with probability $p(a_k) = 1$, contributes $-p(a_k)\log p(a_k) = -1\log 1 = 0$ to the entropy. This makes intuitive sense, because the occurrence of the symbol is certain, it contributes zero information: There is no need to encode an event that must occur. For an alphabet where all symbols are equally frequent, i.e., $p(a_0) = p(a_1) = \ldots = p(a_{n-1})$, each symbol contributes equally to the entropy. In fact, in this case the entropy of the message is $H = 1$. This is the case for the overwhelming majority of possible messages. This is significant because a message with $H = 1$ cannot be compressed using any *entropy coding* method.

An entropy coding method is one that seeks to allocate codewords to symbols based on their observed frequency; more frequent symbols are allocated shorter codewords, while less frequent symbols are allocated longer ones. Since the more frequent symbols are the common case, a net saving in space results from their using shorter codes. The relatively rarer symbols require longer codewords. The length of a given codeword is inversely proportional to the probability of its occurrence in a message.

In practice, the symbol probabilities are estimated rather than known exactly. This leads to a problem known as the *zero frequency problem* (Witten and Bell. 1991). In an optimal encoding, a symbol with an expected frequency of zero will be assigned a codeword of infinite length. If such a symbol does actually occur in a message, the compressed message will be infinitely long. This is obviously not a desirable solution, since the objective is to reduce the message size. The practical solution is to insist on a minimum probability for each possible symbol. Although the methods and encodings used vary (e.g., Cleary and Witten (1984), Bunton (1997), Cleary and Teahan (1997), Teahan and Harper (2001), Begleiter et al. (2004)), they are all similarly effective (Witten et al. 1999).

There are two major optimal entropy coding schemes used in text compression practice: Huffman Codes (Huffman 1952) and Arithmetic Codes (Rissanen and Langdon 1979, Witten et al. 1987, Howard and Vitter 1992, 1994). Both are optimal, in that assuming a

source that is memory-less, and for which the symbol probabilities are known, no other code operating with the same constraints will produce a smaller message. Thus, while Huffman Coding is not optimal in terms of entropy, it is optimal for all whole-bit binary codes. In the case of Arithmetic Coding, the fraction of the size of the encoded message when compared to the original, will be almost exactly the entropy, H, of the message. This near optimal coding efficiency is, however, achieved at some cost. Primarily, Arithmetic Coding is slower when compared with Huffman Coding, and also requires additional care when marking the end of a message.

In comparison to Arithmetic Coding, Huffman Coding is much faster and simpler to implement. The disadvantage of Huffman Coding is that it will produce larger output than Arithmetic Coding, particularly if there are one or more symbols with high probabilities. This is common when encoding alphabets of small size, such as the binary alphabet, $A = \{0, 1\}$. The redundancy of the Huffman Code has been shown to be bounded by $p_1 + 0.086$, where p_1 is the probability of the most frequent symbol. This is the *Gallager Limit* (Gallager 1978). This inefficiency can often be ameliorated, for example by the extension of the alphabet to include digrams and higher order structures to reduce the maximum frequency. This makes Huffman Codes more generally applicable than it would first appear. However, there are situations when the coding inefficiency of Huffman Codes remains problematic.

A problem that is often coincident with the inefficiency of Huffman Codes, is the cost of generating the Huffman Code tree. For static and semi-static compression, the Huffman Code tree need be calculated only once. However, in fully adaptive compression schemes, the tree may need to be regenerated frequently. Efficient algorithms exist for generating Huffman Code trees, e.g., Moffat and Turpin (1998). However, cost still increases with alphabet size. As a result, the coding efficiency gains achieved by extending the alphabet to are in direct opposition to the computational efficiency of the tree construction process. At some point there is a cross over, where Arithmetic Coding begins to offer both better compression and computational efficiency. Typically this is in situations where

adaptive probabilistic models are employed (Bookstein et al. 1993). The border territories, however, remain under dispute with ground shifting as advances are made on either side, e.g., Fenwick (1996), Moffat et al. (1998), Moffat (1999) and Moffat and Turpin (1998), Turpin and Moffat. (2000).

2.3.3 Dictionary Methods

The second major category of loss-less text compression algorithms are those based on dictionary methods. The canonical algorithms in this class are LZ77 and LZ78 both of Ziv and Lempel (1977, 1978). Both algorithms encode successive portions of a message by referencing previous parts of the message. The differentiation lies in how references are encoded.

In the LZ77 algorithm, the longest matching string, plus the next symbol is encoded. The encoding takes the form of a triplet that specifies the distance from the current point, the length of the common prefix, and the single character to be appended in order to produce the new string. The current point is then advanced to just beyond the last symbol that has been encoded. To cater for the situation when no match can be found against the preceding text, the prefix string is set to the empty string, and only a single symbol is encoded. Consider the example of coding the word “banana”. This would result in the triplets as indicated in Table 2.9. For the first three letters, there is no match earlier in the string. However, upon reaching the fourth position, “an” has already been seen. In fact, because during decompression the “an” can be extracted before the final “a” will be read, the last three letters can be recursively encoded in a single triplet.

LZ78 is a refinement of LZ77 where, instead of referring to the distance and length of a string, each candidate string for matching is referenced by a unique index number. The table of strings that can be matched against are formed by concatenating a string that is

Table 2.9: Example Of LZ77 Coding For “banana”.

Distance	Length	Single Character	Region Encoded
0	0	b	Banana
0	0	a	bAnana
0	0	n	baNana
2	3	NULL	banANA

Table 2.10: Example Of LZ78 Coding For “banana”.

String #	Single Character	Region Encoded	New String	New String #
0	b	Banana	b	1
0	a	bAnana	a	2
0	n	baNana	n	3
2	n	banANa	an	4
2	-	bananA	a-	5

already in the table, and the next symbol from the input stream. The candidate string table begins with a single entry, zero, which is the empty string.

Table 2.10 shows how the LZ78 algorithm works for the example text “banana”. For this example, LZ78 encoding requires one more code than LZ77. However, the encoded message can be substantially smaller, because each code can be much more compact. LZ78 codes require $\lceil log_2 n \rceil + \lceil log_2 a \rceil$, where n is the number of strings in the dictionary a given point, and a is the size of the alphabet being encoded. This compares to $\lceil log_2 w \rceil + \lceil log_2 l \rceil + \lceil log_2 a \rceil$ for LZ77, where w is the window length where strings may be referenced, and l is the maximum length of a reference. Efficient implementations of either algorithm may use variable length encodings to improve compression performance. However, this does little to modify the relative ratio of their code sizes.

The attraction of dictionary based algorithms over entropy encoding techniques is two fold, compression performance and speed.

First, dictionaries of strings can provide a better zero-order model than does a method based on dictionary of characters. Such high-order models can indeed achieve better compression

than a zero-order model, but a dictionary-based algorithm can still be a zero-order model and cannot achieve better than zero-order entropy on a given dictionary.

The second attraction of these algorithms is their high speed, especially for decoding. The speed of the LZ algorithms comes from their simple block copy operations. These operations are very fast, and when dictionary methods are combined with byte-alignment, such as in LZRW (Williams 1991) or LZO (Oberhumer 1997), they yield even greater speed, in return for some sacrifice of compression. Indeed, the fastest variant of the LZ algorithms, LZO, decompresses only four times slower than a simple memory-memory copy, while still encoding below the zero-order entropy in many cases (Oberhumer 1997). Because of this combination of speed and compression performance, derivatives of the LZ methods are used in many popular compression algorithms, such as compress (Welch 1984) and gzip (Gailly 1993).

However, in the context of this dissertation the principle value of the LZ methods is that they explicitly encode redundant strings, thus making the redundancy information visible to a sequence search and alignment system that was equipped to use it. The specification and initial evaluation of such sequence search and alignment systems is a major focus of this dissertation.

One challenge that must be overcome in this pursuit is that the LZ methods are adaptive in that they build the dictionary of recurrent strings as they process the data. This means that it is not possible to obtain efficient random access to individual records of an LZ encoded data stream, thus rendering them unsuitable for use in information retrieval systems, such as sequence search and alignment systems. Fortunately algorithms have been developed that substantially address this issue, by combining explicit description of redundancies with an effective means of accessing random records. Two such algorithms are SEQUITUR(Nevill-Manning and Witten 1997) and XRAY(Cannane and Williams 2002).

SEQUITUR operates by creating – in a single pass with linear time and space requirements – a context free grammar from the input text that can be used to reproduce the input text. It does this by searching for repeating digrams. Whenever a digram is detected as occurring more than once, both occurrences are replaced by a non-terminal symbol that triggers a rule that produces the digram.

The non-terminal symbols are treated exactly as symbols from the input text, in that if a digram containing a non-terminal symbol occurs twice, then a compound rule will be created that eventually triggers the first rule. In some situations creating a new rule may cause an existing rule to now be used only once. In that situation the singly used rule is deleted, and its occurrence replaced by its production. This process is illustrated in figure 2.2 with the compression of "wooloomooloo"[6].

No digrams are detected until the 6th letter has been processed. Until that point, the grammar is a single rule, *S*, that produces the output directly. Once the repetition of "oo" is detected, it is replaced with another rule, *A*. Rule *S* is altered to call rule *A* instead of using the repeated string. The process continues uneventfully until the next occurrence of "oo". This time a rule (*A*) already exists for encoding the digram, so that is used. When the second "l" is processed the contraction "*A*l" now occurs twice, so it is replaced by a new rule, *B*. And so the process continues until the last letter is processed. Now the construction *BA* occurs twice, so a new rule, *C*, is produced to remove the redundancy. But this means that the only place where rule *B* occurs is in rule *C*. In order to minimise the size of the grammar (and thus maximise compression) rule *B* is discarded, and its production inserted directly into rule *C*.

Redundancy is explicitly encoded using this method. However, synchronisation points, and thus random access to records, is still problematic because the grammar hierarchy is unlikely to coincide with all record boundaries without some special manipulation.

[6]Woolloomooloo has been purposely mis-spelt to amplify the example.

symbol number	the string so far	resulting grammar	remarks
1	w	S→w	
2	wo	S→wo	
3	woo	S→woo	
4	wool	S→wool	
5	woolo	S→woolo	
6	wooloo	S→wooloo	oo appears twice
		S→wAlA	enforce digram uniqueness
		A→oo	
7	wooloom	S→wAlAm	
		A→oo	
8	wooloomo	S→wAlAmo	
		A→oo	
9	wooloomoo	S→wAlAmoo	oo appears twice
		A→oo	
		S→wAlAmA	enforce digram uniqueness
		A→oo	
10	wooloomool	S→wAlAmAl	Al appears twice
		A→oo	
		S→wBAmB	enforce digram uniqueness
		A→oo	
		B→Al	
11	wooloomoolo	S→wBAmBo	
		A→oo	
		B→Al	
12	wooloomoolo	S→wBAmBoo	enforce digram uniqueness
		A→oo	
		B→Al	
		S→wBAmBA	BA appears twice
		A→oo	
		B→Al	
		S→wCmC	enforce digram uniqueness
		A→oo	
		B→Al	
		C→BA	B is only used once
		S→wCmC	enforce rule utility
		A→oo	
		C→ABA	

Figure 2.2: Compression of “wooloomooloo” using SEQUITUR.

Whereas SEQUITUR lacks a mechanism for providing synchronisation points between records, the XRAY algorithm is designed with random access specifically in mind.

XRAY compresses a document collection in three steps: (1) a small sub-set of the collection is analysed to identify the recurrent phrases and so construct a hierarchical phrase dictionary, that is in many ways similar to that of SEQUITUR; (2) the same sub-set of the collection is compressed, record by record, using the model developed in step (1). This is done to tune the model to take into account the difference between algorithms used to discover the redundant phrases in step (1), and the left-to-right encoding performed in step (3), and; (3) encode the entire collection, record by record, using the refined model that was obtained in step (2).

Because XRAY encodes record by record using a fixed model, synchronisation points naturally result. Further, because the model remains static for the collection, records can be added, modified and deleted from the collection without altering the model and requiring an expensive rebuild. Finally, because the model is small enough to be loaded entirely into RAM, there is no impediment to using XRAY as the method for storing and retrieving records in an information retrieval system.

It is only more recently as computational power and memory capacities have dramatically increased, that a class of algorithms with potentially superior compression to dictionary methods capability has begun to gain popularity, that class is the statistical modelling algorithms.

2.3.4 Statistical Modelling Methods

Data compression algorithms implicitly use the property that symbol probabilities often depend on the preceding symbols in the text. It is possible to use this property, to create compression algorithms that model the statistical behaviour of the message, distinct from its encoding (Rissanen and Langdon Jr. 1981). If the statistical model has a defined initial

state, and is updated using only the previously encoded symbols, it is possible for the decoder to do the same. Thus encoder and decoder remain synchronised, without the need to transmit the model explicitly, allowing the use of arbitrarily complex models that are, hopefully, able to make good quality predictions for each successive symbol.

When such models are combined with an optimal entropy coder, such as Arithmetic Coding, it is possible to create powerful new algorithms with better performance than the LZ family. As the predictions approach $p = 1$, the number of bits required to encode each symbol approaches zero. A statistical model that is able to consistently make high confidence predictions can, therefore, encode each symbol in only a fraction of a bit. Various approaches to modelling exist and are represented in a variety of algorithms, e.g., Prediction by Partial Match (PPM) and variants (Cleary and Witten 1984, Bunton 1997, Cleary and Teahan 1997, Teahan and Harper 2001 etc), Context-Tree Weighting (CTW) (Willems et al. 1995, Tjalkens and Willems 1997) and Dynamic Markov Compression (DMC) (Cormack and Horspool 1987). Of these, DMC is conceptually the simplest, and is described below as an example of a Statistically Modelling text compression algorithm.

The DMC model begins with some simple structure, as in Figure 2.3. Note that any initial structure is possible, and that judicious selection of an initial model can have a significant impact on the compression performance. The model is a simple finite state machine with probabilities assigned to each transition.

The model begins in some state, s. As each bit of the message arrives, the predicted probabilities of the bit being a one or a zero are consulted. These values are passed to an Arithmetic Coder that encodes the event. The frequency counter for the actual value is then updated, thus altering the model. The state transition indicated for the actual value is then taken. Record is maintained of the number of times a state, s_d, is reached from any given state, $s_k, 0 \leq k \leq n$, where n is the number of states in the model.

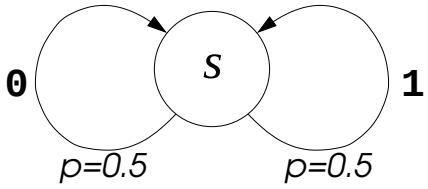

Figure 2.3: Example DMC Initial Model.

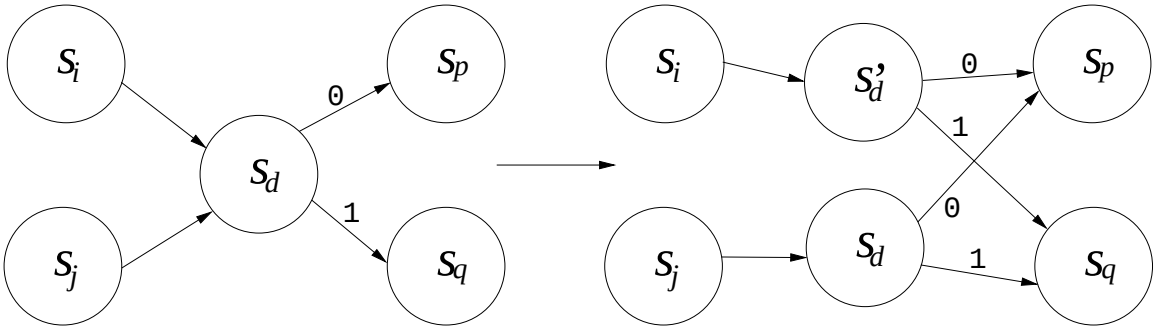

Figure 2.4: Example Cloning Of States In A DMC Model.

When s_d is repeatedly selected by more than one state, e.g., s_i and s_j, it is cloned into two copies each of which are arrived at from only one of s_i or s_j. The frequency counts of the old state are shared between its daughters. In this way, the order of the model increases, and correlations between successive bits, i.e., contexts, can be discovered and modelled. Figure 2.4 shows how this cloning could occur with s_d being cloned to produce s'_d. This process can occur indefinitely to build up an arbitrarily complex model of the source.

2.3.5 Performance Of Compression Algorithms

A common approach for comparing the performance of different algorithms is to benchmark them using a common corpus, such as the Canterbury Corpus (Arnold and Bell 1997). Such a corpus contains a variety of files intended to reflect the types of files that are compressed in the wild. Tables 2.11 and 2.12 contain the Canterbury Corpus compression ratio and decompression speed rankings as of 18 May 2006.

Table 2.11: Table of compression ratio results (sorted by increasing ratio) from the Canterbury Corpus (Arnold and Bell 1997) web site, 18 May 2006. Results are listed in bits per byte, where 8.00 indicates no compression.

Method	Weighted Bits/Byte	Average Bits/Byte	Standard Deviation
ppmD5	1.52	2.11	0.64
ppmD7	1.56	2.15	0.65
bzip-6	1.49	2.15	0.72
bzip-9	1.50	2.15	0.71
szip-b	1.46	2.16	0.74
szip	1.48	2.18	0.74
ppmC-896	1.61	2.19	0.64
ppmD3	1.65	2.20	0.62
bzip2-6	1.54	2.23	0.75
bzip-1	1.59	2.23	0.74
bzip2-9	1.54	2.23	0.75
bzip2-1	1.64	2.31	0.77
ppmCnx-896	1.75	2.32	0.64
bred-r3	1.80	2.38	0.73
dmc-50M	1.74	2.39	0.75
dmc-16M	1.76	2.40	0.74
dmc-5M	1.81	2.43	0.74
gzip-b	2.08	2.53	0.74
gzip-d	2.09	2.54	0.73
ppmC-56	2.16	2.70	0.90
gzip-f	2.46	2.91	0.82
ppmCnx-56	2.41	2.93	0.95
yabba-d	2.59	3.14	0.81
srank-d	2.65	3.31	0.92
compress	2.55	3.31	0.95
lzrw1	3.58	4.18	1.07
huffword2	3.64	4.20	1.44
char	3.64	4.49	1.20
pack	3.74	4.53	1.07
yabba512	3.94	5.19	1.54
cat	8.00	8.00	0.00

Table 2.12: Relative decompression times (sorted by increasing time) from the Canterbury Corpus (Arnold and Bell 1997) web site, 18 May 2006.

Method	Average Time (seconds)	Standard Deviation
cat	0.58	0.86
gzip-b	0.72	0.96
gzip-f	0.79	1.03
gzip-d	0.81	1.06
lzrw1	0.82	1.06
compress	0.92	1.13
pack	0.97	1.13
yabba512	0.98	1.18
huffword2	1.00	1.11
bzip2-6	1.31	1.01
bzip2-1	1.32	1.04
bzip2-9	1.37	0.97
yabba-d	1.42	1.25
bred-r3	1.93	1.13
srank-d	2.03	1.57
char	2.67	1.09
szip	2.86	1.47
ppmCnx-896	3.27	1.29
szip-b	3.29	1.67
ppmCnx-56	3.38	0.89
bzip-6	3.82	1.45
bzip-9	4.28	1.64
bzip-1	4.48	1.88
ppmC-896	4.71	2.09
ppmD3	4.86	2.31
ppmC-56	4.92	1.28
ppmD5	6.45	3.14
ppmD7	7.86	3.39
dmc-5M	11.09	3.40
dmc-16M	11.11	3.11
dmc-50M	11.48	3.21

The PPM derived algorithms dominate the top of the compression ratio chart, but are very slow to decompress. This is to be expected, since a general rule of thumb is that the better the compression: (a) the slower it will be; (b) the more memory it will require; or, (c) both (Witten et al. 1999). Typically, there is a large penalty in execution time for algorithms that use Arithmetic Coding, and the Prediction by Partial Match (PPM) based algorithms are good examples of this.

There are, however, some programs, bzip2[7] (Seward et al. 2001) and szip (Schindler 1996), that compress almost as compactly as the PPM algorithms, but decompress much faster. These are the algorithms representing block sorting algorithms, based on the Burrows-Wheeler transform (Burrows and Wheeler 1994, Manzini 1999). The ability of the Burrows-Wheeler transform to compress rapidly, and almost as compactly as the much more complex statistical modelling compressors has caused a lot of interest, e.g., Balkenhol et al. (1999), Effros (1999), Deorowicz (2000), Wirth (2001), Seward et al. (2001), Deorowicz (2002), Fenwick et al. (2003), Manzini (1999), and is briefly discussed in the following text.

2.3.6 Burrows-Wheeler Transform

The Burrows-Wheeler transform (Burrows and Wheeler 1994, Manzini 1999) is not actually a compression algorithm. In fact, it slightly increases the size of the data to be compressed. However, in the process it sorts the data in a given block making them more readily compressible. For the Burrows-Wheeler transform to be of practical interest, it must also be reversible. This is best illustrated with an example.

Consider compressing the string “CASABA” with the Burrows-Wheeler transform. The first step is to construct a square matrix where each row consists of the successive rotation

[7] Note also bzip, which a previous version of bzip2 that differs only in that it uses Arithmetic Coding instead of Huffman Coding. The difference in compression ratio and decompression speed offered by the two is a practical example of the trade-off between time and compressive power represented by the Huffman Codes (bzip2) and Arithmetic Codes (bzip).

Table 2.13: Burrows-Wheeler Transform Step 1: Construct a square matrix consisting of the successive rotation of the input text, “CASABA”.

C	A	S	A	B	A
A	C	A	S	A	B
B	A	C	A	S	A
A	B	A	C	A	S
S	A	B	A	C	A
A	S	A	B	A	C

Table 2.14: Burrows-Wheeler Transform Step 2: Sort the rows of the matrix.

A	B	A	C	A	S
A	C	A	S	A	B
A	S	A	B	A	C
B	A	C	A	S	A
C	A	S	A	B	A
S	A	B	A	C	A

of the input text, as depicted in Table2.13 for the string “CASABA”. Next, sort the matrix by row, which in our example gives the result shown in Table 2.14. The output of the Burrows-Wheeler transform is nothing more than the right-most column of the matrix and the row at which the original text occurs, which for this example would be (SBCAAA,5). Observe that the transform has produced a more compressible string, by causing the list to be partially sorted.

The transform is reversed by reconstructing the matrix from the output. This transformation can be reversed surprisingly efficiently (Witten et al. 1999). The reconstruction begins by creating a partial re-construction of the matrix using the available information. Recall that the first column of the matrix has been sorted alphabetically, and thus while we know only the last column of the matrix, we can compute the first column by sorting the values in the last column, which using the running example results in Table 2.15. We now have a list of all two letter pairs that exist in the plain text, i.e., (SA, BA, CA, AB, AC, AS). Since we know that the matrix is sorted alphabetically, we can sort that list of pairs to obtain (AB, AC, AS, BA, CA, SA), which must be the first and second columns of the matrix. Filling in

Table 2.15: Burrows-Wheeler Transform Step 3: Reconstruct first and last columns.

A					S
A					B
A					C
B					A
C					A
S					A

Table 2.16: Burrows-Wheeler Transform Step 4: Reconstruct second column from first and last columns by sorting all letter pairs. The sorted letter pairs correspond to the first two columns of the matrix. This process then continues inductively by taking the letter triples (SAB, BAC, CAS, ABA, ACA, ASA), and sorting them to obtain the first three columns of the matrix. This process is repeated until the matrix is fully reconstructed. The index number from the output is used to select the row with the original plain text.

A	B				S
A	C				B
A	S				C
B	A				A
C	A				A
S	A				A

the values of the second column, we obtain Table 2.16. Now we can repeat the process by taking the letter triples revealed by the previous step, i.e., (SAB, BAC, CAS, ABA, ACA, ASA), and sorting them to reveal the contents of the first through third columns, i.e., (ABA, ACA, ASA, BAC, CAS, SAB). This process is repeated until the contents of the original matrix have been fully restored. The original text is obtained by selecting the row number provided in the output of the transform.

It turns out that the Burrows-Wheeler transform operates on a similar theoretical basis to the Prediction by Partial Match (PPM) statistical modelling coders. However the modelling is completely implicit in the process of the Burrows-Wheeler transform. This explains why the Burrows-Wheeler transform is much faster than the explicit modelling methods, such as PPM (Effros 1999) (because there is no modelling work required in the Burrows-Wheeler transform), and also why it does not compress quite as well (because the modelling is implicit, and cannot be tuned for optimal compression).

2.3.7 Synchronisation

Synchronisation points are points in a compressed data stream (in addition to the start of the compressed data stream) from which decompression can be commenced. The value of synchronisation points varies with the application, but in full text retrieval systems, random access is desirable (Witten et al. 1999). However, the best compression algorithms do not exhibit this property.

Arithmetic Coding is not well suited for synchronisation points, because Arithmetic Coding does not introduce clear boundaries between encoded symbols. Moreover, adding such boundaries is often enough to cause Arithmetic Coding to perform worse than Huffman Coding (Bookstein et al. 1993, Witten et al. 1999). This problem is compounded for those algorithms that also employ adaptive statistical models: A decompressor working in such an environment, and attempting to engage mid-stream, lacks the required model context. The various solutions to this problem generally act to worsen the compression performance. For this reason static or semi-static models are preferable when synchronisation is desired.

Huffman Codes also have a extra attraction when synchronisation points are desired, in that they are often self-synchronising. That is, if a decoder joins the stream at any given point, in can rapidly return to the correct phase and decompress correctly after only a few symbols. In fact, it turns out that it is extremely difficult to make a variable length code that lacks this self synchronising property. However, because synchronisation is dependant on the symbols in the message, it is difficult to tell exactly when a code will synchronise (Gilbert and Moore 1959). This makes self-synchronisation problematic in many situations.

Block sorting algorithms like the Burrows-Wheeler Transform (Burrows and Wheeler 1994) described previously, offer natural division points at the end of each block. This can be used to create periodic synchronisation points. In reality, almost any compression

algorithm can be operated in such a way as to output discrete blocks. However, the general rule remains that the smaller the block size, the worse the compression.

The trade-off of block size and compression ratios is of particular concern for nucleic acid and protein sequence retrieval, because the median sequence length is likely to be much shorter than the block size required to reach the compression plateau of most algorithms. It is possible to trade-off random access speed against compression ratio by blocking multiple records together (Witten et al. 1999). However, care must be taken to avoid the danger of choosing block sizes that are large enough to offer decent compression, but prohibit fast random access performance.

2.3.8 DNA Compression

As previously described, DNA consists of the four letter alphabet `A`, `C`, `G`, and `T`. This suggests that it should be possible to store a DNA sequence using no more than two bits per base. DNA sequences may also include wild-card characters (Joint Commission on Biochemical Nomenclature 1983). However, these additional symbols typically occur with very low frequencies. Williams and Zobel (1997b) have shown that these extra symbols can be efficiently encoded and decoded, typically increasing the average compressed message size to only around 2.01 bits per base.

By considering the higher level structure of DNA, such as hairpins, approximate repeats, and relative codon frequencies, it should be possible to improve the compression ratio of individual DNA sequences. A number of algorithms have been proposed, such as those by Korodi and Tabus (2005), Behzadi and Le Fessant (2005), Manzini and Rastero (2004), Chen et al. (2002a, 2001), Li et al. (2001), typically achieving compression ratios of 1.6 – 1.8 bits per base on a small de facto corpus (Table 2.17).

These algorithms utilise similar entropy coding and dictionary based methods to the general purpose compression algorithms discussed previously. In contrast to general purpose

compression algorithms, one distinctive trait that most DNA compression algorithms share is the encoding of approximately repeated strings (rather than only exactly repeated string), as in Chen et al. (2001, 2002a). Many DNA compression algorithms appear to be tailored to using the compression to determine phylogenetic relationships, rather than producing effective and compact storage formats, e.g., Li et al. (2001).

A common problem among DNA compression algorithms is excessive run time for very long sequences. This is addressed in some of the newer algorithms which show promising scalability to compress whole chromosomes (Korodi and Tabus 2005, Behzadi and Le Fessant 2005, Manzini and Rastero 2004). However, there are no algorithms that currently offer both effective compression, and rapid random access to individual sequences within a collection, as would be necessary for efficient combination with an indexed search system.

2.4 Constructing Compact Indices

Indices typically consist of *inverted lists*, that is a list of pointers to each document containing some word, term or analogous structure, in some file. It is often useful to know not only what documents a term appears in, but the position of each occurrence of the term within the document. This means, for English text, that a pointer is required in the index for every word in the text.

Collections of these pointers in an index are referred to as *postings lists*. Assuming that the text consists of n bytes or documents, each posting will require at least $\lceil log_2(n) \rceil$ bits. The entire index will consume $f \lceil log_2(n) \rceil$, where f is the number of pointers required. Such a document level index of an English text database will typically be 50 – 100% of the size of the original text (Witten et al. 1999).

Table 2.17: Compression Performance Of Various DNA Compression Algorithms In Bits Per Base. Results summarised from Behzadi and Le Fessant (2005), Korodi and Tabus (2005), Manzini and Rastero (2004).

Sequence	chmpxx	chntxx	hehcm	humdy	humgh	humhb	humhd	humhp	mpom	mtpa	vaccg	Mean
Algorithm / Size	121k	155k	229k	38k	66k	73k	58k	56k	186k	100k	191k	116k
gzip (Gailly 1993)	2.28	2.33	2.33	2.36	2.06	2.25	2.24	2.27	2.33	2.29	2.25	2.27
bzip (Seward et al. 2001)	2.12	2.18	2.17	2.18	1.73	2.15	2.07	2.09	2.17	2.12	2.09	2.10
Order-2 Arith. Coding	1.84	1.93	1.96	1.92	1.94	1.92	1.94	1.93	1.97	1.87	1.90	1.92
Order-3 Arith. Coding	1.84	1.94	1.96	1.94	1.94	1.93	1.95	1.94	1.97	1.88	1.91	1.93
gzip-4 (gzip of 4 base/byte packed file)	1.86	1.95	1.98	1.95	1.74	1.90	1.91	1.92	1.97	1.88	1.87	1.90
bzip-4 (bzip2 of 4 base/byte packed file)	1.97	2.01	2.01	2.07	1.87	2.00	1.99	2.00	2.01	1.98	1.95	1.99
dna2 (Manzini and Rastero 2004)	1.67	1.62	1.85	1.93	1.37	1.87	1.90	1.91	1.93	1.87	1.76	1.79
BioCompress2 (Grumbach and Tahi 1994)	1.68	1.62	1.85	1.93	1.31	1.88	1.88	1.91	1.94	1.88	1.76	1.78
GenCompress (Chen et al. 2001, Li et al. 2001)	1.67	1.61	1.85	1.92	1.10	1.82	1.82	1.85	1.91	1.86	1.76	1.74
CTW+LZ (Matsumoto et al. 2000)	1.67	1.61	1.84	1.92	1.10	1.81	1.82	1.84	1.90	1.86	1.76	1.74
DNACompress (Chen et al. 2002a)	1.67	1.61	1.85	1.91	1.03	1.79	1.80	1.82	1.89	1.86	1.76	1.73
GeNML (Korodi and Tabus 2005)	1.66	1.61	1.84	1.91	1.01	1.71	1.76	1.88	1.84	1.76	-	1.70
DNAPack (Behzadi and Le Fessant 2005)	1.66	1.61	1.83	1.91	1.04	1.78	1.74	1.79	1.89	1.85	1.76	1.71

However, for nucleic acid and protein databases, the size of an inverted index is pathological, because each and every overlapping k-mer in the database must be indexed in order to obtain maximum sensitivity. Further, while the databases involved are very large, the alphabet size is very small, which makes the ratio of postings to data size large. This predicts an index size of $(n-k)\lceil log_2(n)\rceil$ bits, where n is the number of bases or amino acids in the database, and k is the index width (typically $k \leq 20$). As an example, consider the *Homo sapiens* (Human) UniGene (Pontius et al. 2003, Schuler 1997) build from June 2002, consisting of $n = 2\times10^9$ nucleotides. Each nucleotide base, `A`, `C`, `G` or `T`, can be encoded in two bits. However, each posting will require $\lceil log_2(2\times10^9)\rceil = 31$ bits. There are a total of $(n-k)$ postings to be recorded, and thus the final index will require $(n-k)\times31 \approx n\times31 = 6.2\times10^{10}$ bits. The index will be fifteen times bigger than the data it represents.

Fortunately, much work has been done in the area of index construction and compression, with many practical implementations and implementation issues considered, e.g., McDonell (1977), Buckley and Lewit (1985), Lucarella (1988), Harman and Candela (1990), Fox et al. (1992), Zobel et al. (1993), Moffat and Zobel (1994, 1996). The specific problem of biological sequence database indexing has also been successfully performed by Williams and Zobel (2002a). Much of this work has focused on appropriate entropy coding systems to facilitate the production of much more compact representations of index postings lists, and efficient methods of constructing and maintaining indices of very large collections (Zobel and Moffat 2006).

2.4.1 Compressing Index Postings

Besides the Huffman and Arithmetic Codes, there are other possible entropy coding schemes that have utility in index compression. Two common situations are when the frequency distribution of the symbols is unknown, or when the lexicon size is effectively

unbounded. These are the norm when compressing index postings lists corresponding to words of English text.

There are a range of universal integer codes that can be employed to compress inverted lists. Some, such as the unary, binary and Elias delta and gamma codes (Bentley and Yao 1976, Elias 1975) are fixed codings of all natural integers. That is, they cannot be adapted to concur with an estimation of the expected distribution. The fixed models of these coding schemes have both advantages and disadvantages. An advantage is that additional postings can be added to the end of the compressed list without difficulty, because the model is fixed. On the other hand, if the data differs significantly from the fixed model then compression suffers.

Others, such as the Bernoulli (Golomb 1966, Gallager and Voorhis 1975), Observed Frequency, Skewed Bernoulli (Witten et al. 1992, Bookstein et al. 1992), and Hyperbolic (Schuegraf 1976) codes are parametric, in that they can be tuned to the expected distribution. Many of these can be made *local*, by using the observed frequency and range information associated with each inverted list, in order to select a parametrisation that produces a coding scheme that more closely matches the observed distribution. In either case, the use of a parametric coding entails difficulties when adding new data as this act may alter the parameter, thus requiring the entire list to be recoded.

In addition to the parametrically adaptive coding methods, there are two relatively recent inverted list coding methods, *Interpolative Coding*; and *Selector Coding* that are automatically adaptive, and do not require prior parametrisation, each with differing strengths and weaknesses.

Moffat and Stuiver (1996) devised the Interpolative Coding for ordered lists as a method that recursively divides the interval where the values reside, in order to minimise the number of bits required to precisely place each one. When applied to index postings lists, it normally achieves compressive performance that matches or bests the other coding meth-

ods. The improved compression is mainly due to the adaptive nature of the coder. This allows it to make effective use of non-uniform distributions of indexed terms, known as *clusters*. The drawbacks of the Interpolative Coding method is the computational complexity, being somewhat slower than Golomb Coding (Anh and Moffat 2005, Trotman 2003), and that appending to a list requires recoding the entire list.

Selector Coding, is described by Anh and Moffat (2005). Like Interpolative Coding, Selector Coding is sensitive to clustering, but uses a relatively simple system of fixed width binary selector codes. This allows the method to decode much faster than either Golomb or Interpolative Coding, yet achieves compression factors approaching that of Interpolative Coding. Because selector blocks are self contained, items can be appended to an existing list without recoding the entire list. The discrete selector blocks also make it possible to seek to a desired point in the inverted list without having to decompress it all.

In the case where a database is likely to be updated on a regular basis, Selector Coding is probably the preferred coding method. However, if the database remains static, and compression ratio is more important than decompression speed, then Interpolative Coding may be the method of choice.

A common theme irrespective of which coding methods is employed is that relative rather than absolute ordinal document numbers and word positions are recorded, i.e., the size of the gap between successive instances. This converts a posting list that describes the instances of some term in n documents from being evenly distributed over $[0..n]$ into an extreme distribution, which better lends itself to compression due to its skewed probability. For a fuller discussion of all these coding methods, and their application to index compression, refer to Zobel and Moffat (2006).

2.4.2 Efficient Index Construction

There are a number of possibilities when it comes to constructing an inverted file from a database (Zobel and Moffat 2006). The methods differ depending on the relative size of the collection versus the amount of available RAM and disk space.

If the collection is substantially smaller than the available RAM, then the index can be constructed in RAM in a single pass, and then written to disk. This process is called *In-Memory Inversion*.

If the collection is larger than the available RAM, it is still possible to perform an In-Memory Inversion by making two passes of the collection. During the first pass a skeleton of the index is constructed on disk, and then constructing the actual index during the second pass that successive portions of the collection are indexed in RAM, and then written out sequentially into the skeleton that was produced during the first stage. This has the obvious shortcoming of requiring two passes over the data, which is undesirable since the transfer time of passing over the data is often the largest time cost during index construction. Also, it requires that the lexicon or vocabulary of the collection be held in RAM.

An alternative method that avoids two passes over the data is to produce the list of postings during the single pass. This list of postings will be ordered by document number. The index is then produced by sorting the list of postings by term, thus this method is called a *Sort-Based Inversion*. While this process can be made efficient, it still requires sufficient RAM to hold the entire vocabulary, and in addition requires enough disk space to write the list of postings. The end result is a process that takes roughly the same amount of time as a partitioned In-Memory Inversion.

One method of avoiding the need to keep the entire vocabulary in RAM is to build a number of sub-indices (using either an In-Memory or and then merge them in a *Merge-Based Inversion* (Heinz and Zobel 2003). If this is performed efficiently, it requires only a slight overhead in disk space and is currently the most efficient method (Zobel and Moffat 2006).

2.4.3 Document Reordering And Filtering

The development of coding schemes, like Interpolative Coding and Selector Coding, that can efficiently encode postings lists where the frequency of a given term varies (forming clusters of occurrences, and inter-cluster spaces between them) has prompted a number of authors to consider how such aberrations can be encouraged. This has resulted in a series of studies exploring the possibility of reordering the documents within a database such that more clusters are produced (Blanco and Barreiro 2005, Silvestri et al. 2004b, Blandford and Blelloch 2004, Blandford et al. 2003, Shieh et al. 2003, Blandford and Blelloch 2002), yielding space savings of up to 30%. Significantly, recent advances such as the SPEX algorithm (Bernstein and Cameron 2006) that can sort a database in roughly linear time, making it possible to sort large genomic databases.

Bernstein and Zobel (2005) have explored excluding entire content-equivalent documents from the index of a collection. This is similar to the way in which the UniGene databases are made non-redundant by removing all identical database records. By excluding a percentage of documents from the index in this way, the index can be shrunk. However, in biological sequence searches it can be important to return all identical or nearly identical matches against a query, if only because each may correspond to a different organism, and so the inclusion of the target organism, or more generally speaking, the title of each redundant document in the results is important.

2.4.4 A Compelling Opportunity: Cooperative Compression

One area that does not appear to have been explored in index compression, is making use of the redundancy exposed by dictionary based compression algorithms applied to the database, and enhanced by document reordering methods. That is, to post only one copy of each recurring string. Algorithms such as SEQUITUR and XRAY are able to expose such redundancy in a readily usable format, but this author is not aware of any index system

that uses the exposed redundancy to reduce the index size instead of only the compressed document size.

This approach retains all documents, but should be able to achieve improvements in inverted list compression over document reordering alone. This is because while Witten, Moffat, and Bell (1999) are reluctant to condone the use of stop lists to reduce index size, it is safe to exclude postings that can be recovered during decompression from the structure of the compressed data itself.

Statistical compression algorithms are ill suited to this task, because the redundancies they encode are not readily extractable from the compressed data stream. However, in dictionary based methods the redundancy may be more easily extractable. Of particular interest are LZ77 (Ziv and Lempel 1977) and LZ78 (Ziv and Lempel 1978) and their derivatives. Such algorithms achieve their compression by flagging often lengthy recurrences (i.e., repeated instances of the same string) between nearby regions of a message. If a list of such encoded recurrences was available during index construction and decompression, then only one instance of the recurrent string need be posted in the index, as the others could be computed during decompression.

This approach has the potential to achieve much of the same savings as a stop list or merging of content equivalent documents, but without sacrificing the ability to query on abundant terms, or highly similar documents. In the context of sequence alignment, this benefit could be further leveraged by reusing alignment effort expended on one posting, in the context of the listed recurrences of the same string.

Chapter 3

Materials And Methods

Introduction

This chapter describes the experimental framework used throughout this dissertation. The description begins with the selection of nucleic acid and protein databases, each suited to the goals of cooperative compression. The selection of databases is followed by a discussion regarding the selection of a set of test queries. Having defined the data context in which the algorithms of this dissertation must work, a set of existing algorithms is listed. These algorithms provide a peer-group and point of comparison for the algorithms introduced in this dissertation. In addition, the algorithm of Smith and Waterman is selected as a benchmark to which each of the peers is compared.

The remainder of the chapter addresses the comparison of these algorithms, defining (a) the metrics each algorithm is measured against, and (b) the standardised and automated system used to run and collect the results of each query. The chapter closes by presenting a summary of the relative performance of each algorithm. These summary results are presented in a standardised format that is reproduced in later chapters, with results added for algorithms introduced in this dissertation.

3.1 Selection Of Databases

The thesis of this dissertation is that redundancy, when present in biological sequence databases, can be harnessed to improve the time and space characteristics of sequence search and alignment. This is accomplished by merging the storage and search effort of multiple instances of identical sequence fragments. To make effective use of this redundancy, the recurrences must occur near one other. That is, the database must be *sorted*, which in this context means that similar sequences are formed into clusters. Therefore, the ideal database would be one that is already well sorted. However, an unsorted database could be used, but it would require sorting during the indexing phase, thus incurring considerable computational expense. Therefore pre-sorted and redundant databases are to be preferred, although methods have subsequentially been published that allow the sorting of a genomic database in approximately O(n) time (Bernstein and Cameron 2006).

3.1.1 Nucleic Acid

Two nucleic acid databases were selected for use in this dissertation. One was chosen as being particularly suited to cooperative compression, while the other was chosen as being more typical of genomic data.

The first nucleotide database that was selected is the June 2002 build of the Human UniGene database. This database was selected because the UniGene builds are large sorted nucleic acid databases. Further, because the UniGene database is a transcriptome, it contains substantial redundancy. In these regards, the UniGene databases satisfy the criteria for effective cooperative compression: They contain redundancy, and are pre-sorted by sequence similarity. The FASTA formatted Human UniGene (nucleotide) database contains approximately 1.96×10^9 bases in 3.5×10^6 sequences, and 470 MB of FASTA sequence descriptions. The total size of this FASTA formatted Hs.seq.all database is 2,365 MB.

Table 3.1: Per Chromosome And Total Size Statistics Of The April 2003 Draft Of The Human Genome.

Chromosome	Number of Bases	Percent Known	Percent Wild-Card
1	245,203,898	89.2	10.8
2	243,315,028	97.4	2.6
3	199,411,731	97.1	2.9
4	191,610,523	97.4	2.6
5	180,967,295	98.1	1.9
6	170,740,541	97.7	2.3
7	158,431,299	97.5	2.5
8	145,908,738	97.1	2.9
9	134,505,819	85.6	14.4
10	135,480,874	96.5	13.5
11	134,978,784	96.8	13.2
12	133,464,434	96.9	3.1
13	114,151,656	83.7	16.3
14	105,311,216	82.8	17.2
15	100,114,055	81.0	19.0
16	89,995,999	88.8	11.2
17	81,691,216	94.8	4.2
18	77,753,510	95.9	4.1
19	63,790,860	87.4	12.6
20	63,644,868	93.4	6.6
21	46,976,537	72.2	27.8
22	49,476,972	69.4	30.6
X	152,634,166	96.8	3.2
Y	50,961,097	44.7	55.3
Total	3,070,521,116	92.2	7.8

The second nucleotide database that was selected is the 14th April 2003 build of the Human Genome. As a complete eukaryotic genome it can arguably be considered representative of nucleic acid data. Moreover, this database has been used as the basis for evaluating existing DNA compression algorithms, e.g. GeNML (Korodi and Tabus 2005). Together, these characteristics make it an appropriate and rigorous challenge for the cooperative compression techniques described in this dissertation. Table 3.1 lists the total number of bases, and the proportion of wild-card bases in each chromosome, and for the genome as a whole.

3.1.2 Protein

The protein database that was selected is a release of the GenPept transcript database. This database was selected because of its substantial internal redundancy. As will be shown in a later chapter, one quarter of its constituent sequences are duplicated in their entirety. The GenPept (protein) database is not sorted, and contains approximately 5×10^8 letters (i.e., amino acids) in 1.6×10^6 sequences, and 142 MB of FASTA sequence descriptions. The total size of this FASTA formatted GenPept (protein) database is approximately 620 MB.

3.2 Query Selection

Selecting suitable test queries is problematic. There are two main options in this area: (a) queries with pre-judged results, often by human assessors; or (b) queries with no pre-judged results.

The first category of test queries, i.e., those with pre-judged results, are routinely used in the information retrieval community, e.g., by the TREC community; the assumption is made that the human assessors are accurate and consistent. However, there is evidence to suggest that this is not the case: It is possible that 10 percent or more of human supplied judgements are incorrect or inconsistently applied (Bernstein and Zobel 2005). Notwithstanding this difficulty, such queries have been used in assessing sequence search and alignment algorithms. For example in work by Williams (1999), the CAFE algorithm was assessed using the existing classification of protein sequences into families and super-families. The authors of that paper note that using pre-judged queries, in the form of protein super-families, causes the assessment to be approximate. This is because an algorithm that detects relationships between sequences will be penalised if the human judges have imposed an artificial or erroneous division between them.

The second category of test queries, i.e., those with no pre-judged results, allows a different treatment of this sensitivity assessment problem. Consistency in judgement follows if the human judge is replaced by a trusted deterministic algorithm. Moreover, if the judge algorithm is an optimal, or so called *benchmark* version of the class of algorithms being compared, then accuracy can also be assumed. The algorithm of Smith and Waterman can be used as the benchmark for each of the heuristic algorithms surveyed in Chapter 2, as well as those introduced in this dissertation. Of similar importance, using a fast mechanical judge makes it possible to use practically any data set, rather than the few, and often restrictive data sets that have been subjected to human classification.

While using a deterministic and mechanical judge solves what may be called the objectivity problem, and allows assessment to be performed on unclassified data, it introduces a subtle problem of its own: Each algorithm searches for statistically significant alignments, not biologically significant alignments. This is analogous to Magnetic North versus Grid North on a map: While the link between the two concepts is strong and sound, they are not identical. One area of divergence between statistical and biological significance is that of low complexity regions of sequences, i.e., highly repetitive or compositionally biased sequences. Alignments against such regions may be statistically significant, but not necessarily biologically significant. Ideally, to provide a fair sensitivity comparison of all algorithms against that of Smith and Waterman, any mechanisms in the algorithms being compared that exclude low complexity regions should be disabled, so that all are searching for statistical significance. This criterion excluded the use of POZ scores (Booth et al. 2004) on the basis that POZ-scores attempt to remove the bias induced by low complexity regions.

On balance, it was decided to use a mechanical deterministic judge rather than human judged queries, i.e., to use the algorithm of Smith-Waterman as a benchmark against which all other algorithms will be measured. The standard test queries were generated by randomly selecting two hundred sequences from each database, using the nucleotide sequences

to search the nucleotide database, and the protein queries to search the protein database. Thus, in this dissertation, there are two hundred standard queries for each of the nucleotide and protein databases. In all cases the queries were left in the database so that at least one perfect match would be present. The nucleotide queries ranged from 134 to 3224 bases (mean 625 bases), and the protein queries ranged from 5 to 1855 acids (mean 350 acids).

It is acknowledged that the selection of these queries and the aggregate presentation of results corresponding to them suffers from two deficiencies: (1) More queries could have been used to obtain better confidence in the results presented in this dissertation, and; (2) The range of query lengths does not correspond to the short (several dozen) residue queries often used in high-throughput sequencing.

3.3 Speed And Sensitivity Metrics

Metrics are required to interpret and compare the results of each algorithm. For comparing execution speed, the following simple metrics will be used: the mean, median and total elapsed processing time for the two hundred queries. The total elapsed processing time will be normalised as a ratio of the run time of NCBI-BLAST, reflecting the fact that NCBI-BLAST remains the de facto standard sequence search and alignment program. In all cases searches are performed “warm”, i.e., with the relevant databases fully resident in RAM.

Turning now to sensitivity metrics, Chen (2004) has shown that despite the multitude of sensitivity metrics that are currently used, they give consistent results. In light of this, and following the discussion of this issue in Section 2.1.1.7, a variation of the PatternHunter metric described there is employed in this dissertation.

The variation is to measure *coverage* instead of alignment score. Coverage here refers to the number of residues included in the alignment that correspond with the true alignment, as returned by the benchmark. Using coverage instead of score avoids comparison difficulties

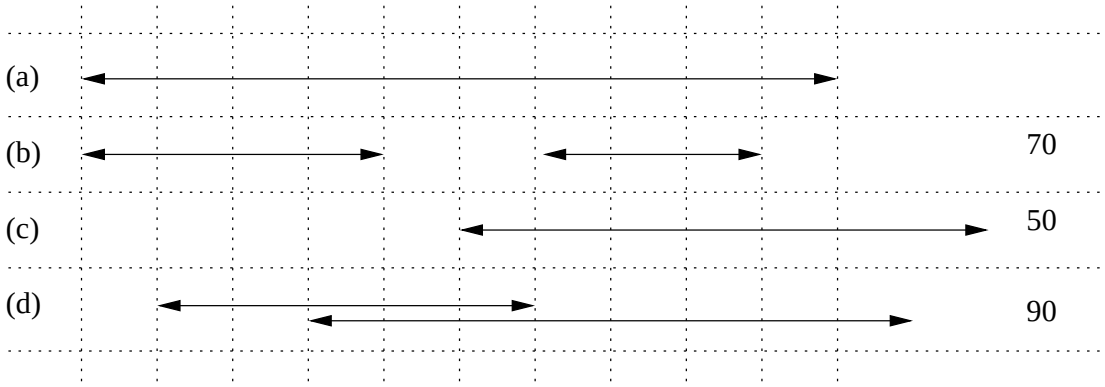

Figure 3.1: Calculation Of PatternHunter Variant Metric. (a) is the perfect, or "golden", alignment against which the other results, (b), (c) and (d) are measured. The numbers on the right hand side are the scores for each of (b), (c) and (d).

among algorithms that report scores differently, e.g., as raw dynamic programming scores versus bits, nats of information, or scores that take into account other alignments. Measuring coverage also makes it easier to resolve situations where multiple alignments occur, or where alignments do not lie entirely within the alignment produced by the benchmark.

The calculation of this metric is demonstrated in Figure 3.1. Alignment (b) scores 70 because the two fragmentary alignments together cover 70% of the reference alignment. Alignment (c) scores only 50, because although the alignment is greater than 50% of the length of the reference alignment, it covers only 50% of the reference alignment. It is apparent how measuring coverage avoids the difficulty of determining how much of the score should be counted. Result (d) shows a further complication, where not only does part of an alignment have to be discounted, but the overlapping parts of the alignment must be counted only once. Again, determining the total score would be difficult if the final score was calculated based on the scores of the alignments. However, it remains trivial to determine the total coverage against the reference alignment.

3.4 Peer Group Of Sequence Search And Alignment Algorithms

A number of the algorithms surveyed in Chapter 2 were bench marked to provide a peer group against which the algorithms introduced in this dissertation were compared. For each algorithm, the benchmarking was performed using the standard queries and databases introduced in the preceding text. Thus two hundred protein and two hundred nucleic acid queries were performed for each algorithm. Appendix A lists the commands and parameters used to invoke of each of these algorithms where a consistent table format is used to describe the set-up and search commands for each invocation of an algorithm. This same format also used to describe the set-up and search commands used for the algorithms introduced in this dissertation. Finally, as far as was practical, the results of each algorithm were obtained in equivalent conditions, with any deviations being noted in the following discussion.

3.4.1 Smith-Waterman (SSEARCH 3.4t25)

Two implementations of the Smith-Waterman algorithm were considered: SeqAln (Hardy and Waterman 1997) and SSEARCH that comes with the FASTA (Pearson 1990) software distribution available at `HTTP://www.ciri.upc.es/cela_pblade/FASTA.htm`. SeqAln was initially used to produce the Smith-Waterman reference results in this dissertation. However, it later became apparent that SeqAln does not always return the optimal (maximally scoring) alignment. Figure 3.2 shows an example of this problem. Therefore SSEARCH was used instead, and all reference results that had previously been produced using SeqAln were reproduced using SSEARCH.

In terms of execution characteristics, the SSEARCH program reads FASTA files directly, no index or other ancillary structures were involved. The processing and access time of the

```
Score: 1753 at (gnl|UG|Hs#S4031081 AGENCOURT_6431325 Homo sapiens cDNA,
        5' end /clone=IMAGE:5503216 /clone_end=5' /gb=BM4
        67009 /gi=18516051 /ug=Hs.284170 /len=1148)[203..1041]
        : (gnl|UG|Hs#S3317510 602385242F1 Homo sapiens cDNA,
        5' end /clone
        =IMAGE:4514242 /clone_end=5' /gb=BG290161 /gi=13046677
        /ug=Hs.284170 /len=787)[1..785]
  203 CTTTTAGCTCACTCCACAAGTAAATGGATTTAATCAAAGGTCACCTATCTGCTTTTGATT   262
      |||||||||||||||||||||||||||||||||||||||||||||||||||||||||  | |
    1 CTTTTAGCTCACTCCACAAGTAAATGGATTTAATCAAAGGTCACCTATCTGCTTTATA-T    59
                                  ...
  916 TGGCCCCTATTTAATTTCCTGAATTTCCAAATAAAATTTTCTCATAACCGGGGCCTCCAA   975
      | | || ||||  ||||  ||| ||  ||||||||  ||||| |||  | |||    |||
  682 TAG-CCATATT--ATTT--TGACTT--CAAATAAA--TTTCT-ATA--CTGGGTAAACAA   729
  976 TAACTACCCTTCATCTCTAACACTTGCCCTTATTCAACGTGGGGCCGGCCAGCTCCTAAA  1035
       ||  |      |     || |  ||     |   |  | ||        |      |||
  730 GAAAAA------AAAAAAAAAAAATG----GAGAAATGGAGGAAAAAAAAAAAAAGAAAA   779
 1036 AAATCA  1041
      ||| ||
  780 AAAGCA   785
```

Figure 3.2: Example Of Incorrect Result From SeqAln. The scoring system is +1 for a match, -3 for a substitution, -5 to open a gap, and -2 to extend a gap. SeqAln over extends the alignment, as can be deduced from the inclusion of the last three bases: TCA vs GCA scores -3 +1 +1 = -1. Thus, the alignment would score higher if it were truncated, and therefore the alignment as shown cannot be the optimal (highest scoring) one.

database were both insignificant compared with the long search times that characterise the Smith-Waterman algorithm.

Finally, the nucleic acid scoring parameters used were match and substitution scores of +1 and -3, respectively, and gap open and extension penalties of -5 and -2, respectively. The protein scoring parameters used were the BLOSUM62 substitution matrix, and gap open and extension penalties of -11 and -1 respectively. All other algorithms were run using the same scoring system.

3.4.2 BLAST (NCBI-BLAST 2.2.6)

Search results were obtained for the two hundred standard queries using pre-compiled Linux binaries for NCBI-BLAST 2.2.6. These results were obtained on a Sun V20z dual processor AMD Opteron server (1.8 GHz, 1 MB L2 cache, 8 GB RAM) running Red Hat Enterprise Linux AS 3.0UP2 Linux (64bit 2.4.x kernel, and 32bit version of NCBI-BLAST), counting only the user process time. BLAST was run in single threaded mode for ease of comparison of run time, since several of the other algorithms did not support multi-threaded operation. The database was formatted using the `formatdb` program. Computers with between 2 GB and 8 GB of main memory were used, ensuring that the databases fit entirely into RAM, and I/O delays were avoided.

The queries were run using three different sets of parameters for BLAST (Tables A.3, A.4 and A.5). These correspond to, respectively: (a) The default parameters of BLAST; (b) The default parameters of BLAST, but with query filtering disabled, and; (c) The default parameters of BLAST, but reporting as many alignments as possible, instead of enforcing the default limits on the both number and statistical significance of reported alignments.

3.4.3 BLAT

Release 32 of BLAT was run on the same platform as BLAST. It was compiled using the default optimisation level selected by the installation scripts ("-O"). BLAT results were produced for both the nucleotide and protein standard queries. Comparison of search time between BLAT and other algorithms was complicated by BLATs use of a central server to host the nucleotide index. This index server takes several minutes before it is ready to respond to queries, but then requires only a few seconds to process the two hundred standard queries. The CPU time required by the server to answer the queries was added to the CPU time consumed by the search tool. But to provide a fair comparison, only the CPU time consumed by the server after receiving the first query was counted.

BLAT protein searches do not use the in-memory index, so the run time reported for the standard protein queries were not manipulated. However, a static overhead is introduced because the entire protein database is read for each query. This overhead was not deduced, as it is a real and unavoidable cost incurred in the search process, unlike the index preparation time for nucleic acid queries that can be paid before processing a query.

3.4.4 Academic Version Of PatternHunter

PatternHunter is designed for nucleotide searching only, so protein query results were not obtained. The freely distributed version of PatternHunter, while written in Java, presents as a Windows .EXE file. Therefore the PatternHunter results were produced on a different hardware/software platform. PatternHunter was run on an AMD Athlon 2100+ 1.7 GHz processor with 1 GB of RAM, 512 KB L2 cache running Microsoft Windows XP SP2. The standard queries were generated with BLAST on this platform as a reference point, establishing that the search speed of this processor was within three percent of the 1.8 GHz Opteron processors used for all the other tests. As this discrepancy was small, no correcting factor was applied to the speed of the PatternHunter runs.

Of greater concern for the comparison of PatternHunter results, is that PatternHunter indexes while it searches. This meant that the run times were hundreds of times greater than those of BLAST or BLAT. Since PatternHunter is designed to align genome against genome, not sequence against database, it could be argued that counting the indexing time constitutes an unfair test. To exclude the constant indexing time, the fastest search time of any standard query against the database was subtracted from the search time of all the standard queries performed by PatternHunter.

3.4.5 FASTA

Version 3.4t25 of the FASTA source code was downloaded from `HTTP://www.ciri.upc.es/cela_pblade/FASTA.htm`. That version was not tailored to run on the Linux/Opteron combination used as the standard test platform. Parameters specific to Motorola PowerPC processors (`-mcpu=970` and `-tune=970`) were removed from the file `src/Makefile.blade.gcc64scl`. The source code then successfully compiled using the command `./compile.sh gcc 64 scalar`.

The standard queries were performed using FASTA formatted databases, as per Table A.7. Two different configurations were used for the protein queries. The configuration labelled (a) in Table A.7, is more thorough than the faster alternative configuration labelled (b).

3.4.6 CAFE

CAFE (Williams 1999, Williams and Zobel 2002a) version 0.14 was downloaded from `HTTP://www.bsg.rmit.edu.au/cafe/`. This code was compiled after using the following command:

```
./configure CFLAGS=-O2 --prefix=/home/paul/opt/cafe
```

The program was built on the same Linux/Opteron test platform as used for the other algorithms. CAFEs alignment display routine was enhanced to report alignments in a format more like that of BLAST, BLAT and DASH to enable the use of a single program to parse the results of all four algorithms. The search parameters for CAFE are tabulated in Table A.8.

3.4.7 Algorithms Introduced In This Dissertation

The algorithms introduced in this dissertation were tested on the same platform as for the other algorithms, i.e., Sun V20z systems with two 1.8 GHz AMD Opteron processors, sufficient RAM to hold the database index in memory, and running Red Hat Enterprise Linux AS 3.0UP2. The database formatting, index construction, and search configuration data for each scenario is described in the appropriate places in Chapters 5 and 7.

3.5 Batching Environment

A batching environment was created to automate the execution, result gathering, and statistical analysis of the standard queries for each algorithm/database combination. This section describes the directory layout and programs that compose the batching environment.

3.5.1 Overview

The batching environment provides the directory structures, programs and semantics to perform the automated execution, comparison and calculation of statistics for groups of searches (batches) obtained from a variety of search algorithms. In this dissertation it is used to compare the relative performance of various sequence search and alignment algorithms.

3.5.2 Directory Structure

3.5.2.1 Top Level Directories

A given instantiation of the batch environment consists of the following directory trees:

```
bin/
R/
cases/
output/
data/
```

This is referred to as a *base directory*. Figure 3.3 presents an example of the directory structure, which is explained in more detail in the following text. The `bin` and `R` directories contain programs and scripts used to automate the batch process. The standard query sequences are stored in the `cases` directory in separate FASTA format files. The summarised output files from searches are collated beneath the `output` directory, and executive and statistical summaries of those results are computed and placed into the `data` directory, ready for analysis using the R statistical computing package (R Development Core Team 2006).

Sub-directories exist within the `output` directory for each search program that is being assessed. For example, the `output` directory may contain sub-directories called `dash`, `blast` and `SeqAln`. Below those sub-directories, a second level exists for each batch of results for a given search program, and are referred to as *batch definition directories*. For some programs there may be only batch, while for others there may be many. Each batch definition directory will contain at least the following files:

```
description
pre
post
```

```
bin/
  makemake comparejobs comparebatches pairwise2summary
  seqaln2pairwise mktrace
R/
  generatesummaries.R
cases/
  query_1 query_2 query_3
output/
  blast/
    normal/
      description template pre post
  dash/
    r7_normal/
      description template pre post
    r7_careful/
      description template pre post
data/
```

Figure 3.3: Example Complete Batching Environment Directory Structure.

```
template
```

```
results/
```

The `description` file contains a human readable description of the mode of operation employed, any special conditions, and any other comments appropriate to the set of results it contains. Figure 3.4 shows a description file for one of the algorithms used in this dissertation.

The `pre` and `post` files are executable scripts that are executed before and after performing a set of searches. They provide a convenient mechanism for setting and cleaning up any special environment before and after searching, e.g., building and removing database indices.

The `template` file contains a single line containing command text and shell variables that will be substituted by the batching environment, to produce the final command required to perform a single specific search. Figure 3.5 illustrates an example template line for running a search with one of the algorithms used in this dissertation.

```
dash -s mode4 , rev12 July 2006,
2x (but uses only 1) AMD Opteron 244 (1.8GHz) CPU,
8GB RAM, 1MB L2 cache.
using Hs.seq.all.r.{np3,nix}, seq limit=20k,
                              residue limit=10M.
        np3 -n -r -9 Hs.seq.all.r
        nix -v -r Hs.seq.all.r.np3
```

Figure 3.4: Example Batching Environment Description File. This file contains a human readable description of the conditions of the batch.

```
/home/paul/bin/dash -s mode2 -b 1000 -p dashn \
  -d $DASH_DB_FILE -i $QUERY -o $OUTFILE
```

Figure 3.5: Example Batching Environment Template File (In the real file, this must appear as a single line). The environment variables QUERY and OUTFILE are initialised by the batching environment. DASH_DB_FILE is a variable inherited from the user's environment.

The `results` directory contains a sub-directory for every search case performed, each of which will contain the run time information for that search, as well as a summarised version of the search results. This is stored in a summarised format for later processing by search comparison tools.

In addition to the mandatory files described above, an additional file, `profile_template`, is also supported. This file is treated identically as the `template` file, except that the batching environment will run the `gprof` command when the search completes to gather profiling statistics over the batch of searches. If a `profile_template` file is used, the `template` file must still exist, as the command it contains is still used to obtain the run time of each query, since an executable that supports profiling is typically much slower than the equivalent optimised executable.

3.5.3 Generation Of Standard Queries

The individual search strings (that is, *queries* or *cases* for short) are selected by running the `pickquery` program. This program randomly selects a single sequence from a FASTA for-

```
#!/bin/csh -f
set n=0
while ( $n < 200 )
  bin/pickquery Hs.seq.all >cases/query_$n
  @ n = $n + 1
end
```

Figure 3.6: Use Of `pickquery` Program To Obtain Standard Nucleic Acid Queries.

```
runbatch -b /home/paul/thesis_data/search_comparison_p \
         -d genpept.fsa -j output/dash/r11_mode4 \
         -w /home/paul/tmp -l 0 -h 199
```

Figure 3.7: Sample Use Of `runbatch` Program.

matted database, including the associated FASTA description line. It is used in conjunction with a simple shell script, such as in Figure 3.6, to provide a set of query sequences.

The above procedure was used on the Human UniGene (nucleotide) and GenPept (protein) databases to provide the two hundred standard query sequences from each. The queries were placed in `search_comparison_n/cases` and `search_comparison_p/cases` directories, respectively.

3.5.4 Execution Of Batches

3.5.4.1 Executing A Batch

Batches of searches are executed by using the `runbatch` program. This program takes as arguments the necessary directory names to precisely identify the batch to perform. For example, Figure 3.7 presents an invocation that was used to perform a batch of protein searches using the DASH algorithm with the two hundred standard queries. The `-b` option tells `runbatch` the base directory for the batch. That directory is assumed to contain the `bin`, `R`, `cases`, `output` and `data` directories, as previously described. The `-l` and `-h` options specify the inclusive range of the standard queries to be executed.

```
> gnl|UG|Hs#S544556
  (1-385) = (1-385), score = 383.000000
> gnl|UG|Hs#S3898566
  (183-385) = (292-488), score = 179.000000
  (11-185) = (15-188), score = 149.000000
  (11-185) = (15-188), score = 148.000000
```

Figure 3.8: Example Of The Terse Alignment Format. Considerable space is saved by excluding the alignment of the two sequences.

```
runbatch -p seqaln2pairwise -b batch_dir -d Hs.seq.all \
         -j seqaln/normal -w /tmp -l 0 -h 199
```

Figure 3.9: Example Invocation Of `runbatch` With Custom Output Filter. The custom filter is `seqaln2pairwise`.

3.5.4.2 Summarisation Of Search Results

The `runbatch` program also fills the role of summarising the results of each search in the batch. This is accomplished by converting the output from the native format of each search program to a terse and simple format, similar to that of Figure 3.8. This format contains only the score, location and extent information of each alignment. This saves considerable space when storing the results of many batches.

By default, the output of each search program is expected to be in the pairwise alignment format used by BLAST, BLAT, and DASH. It is possible to use an external filter when running other search programs that use differing output formats, such as the SeqAln program. This is performed by using the `-p` command line option to `runbatch`, e.g., as in Figure 3.9.

3.5.5 Comparison Of Batched Search Results

Comparison of individual search results is performed by the `comparejob` program. Similarly, the `comparebatches` script compares the results of entire batches of searches. Both of these programs are called automatically if a sequence of commands similar to that of Figure 3.10 is employed.

```
cd (batch environment directory)          # 1
bin/runbatch -j blast/ncbi2.2.6 ...       # 2a
bin/runbatch -j dash/r11_mode2 ...        # 2b
bin/runbatch -j dash/r11_mode4 ...        # 2c
bin/makemake `pwd`                        # 3
cd data                                   # 4a
make                                      # 4b
cd ..                                     # 4c
```

Figure 3.10: Example Command Sequence To Execute And Summarise The Results Of Several Batches.

The first command of Figure 3.10 enters the batching directory. This is followed by commands 2a-c that execute the batches (the command line arguments associated with these commands are abbreviated for clarity). Once the batches are complete, the `makemake` command is used to create a make file in the `data` directory. `makemake` explores the `output` directory, identifying each valid batch. The commands required to compare each batch against every other batch are written into `data/Makefile`.

Finally, commands 4a-c enter the `data` directory and use the `Makefile` to create the full set of summary and statistical data. The `comparejobs` and `comparebatches` commands are invoked to perform the comparison of every pair of batches.

Following this process, and assuming the example directory structure of Figure 3.3, results in the files listed in Figures 3.11 and 3.12. The summarised results of each query is placed in a file called `hits` in the appropriate directory, and the run time is placed in the `times1` file in the same directory. The data directory is populated with the `Makefile` and comparison data, both per batch, and summarised for all batches.

The files consisting of the names of a pair of algorithms, e.g., `blast.ncbi2.2.6.dash.r11_mode2.csv`, contain the query by query comparison data that are used to produce the summary files `lph50s.csv`, `time.csv` and `ratio.csv`. The files that are prefixed by "`ph.`", e.g., `ph.blast.ncbi2.2.6.dash.r11_mode2.csv`, contain the PatternHunter variant scores for the pair of batches contained in the file name.

```
bin/
  makemake comparejobs comparebatches pairwise2summary
  seqaln2pairwise mktrace
R/
  generatesummaries.R
cases/
  query_1 query_2
output/
  blast/
    ncbi2.2.6/
      description template pre post
      results/
        1/
          hits times1
        2/
          hits times1
  dash/
    r11_mode2/
      description template pre post
      results/
        1/
          hits times1
        2/
          hits times1
    r11_mode4/
      description template pre post
      results/
        1/
          hits times1
        2/
          hits times1
```

Figure 3.11: Example Batching Environment Directory Structure After Running Batch: Excludes `data` Directory.

```
data/
  Makefile
  lph50s.csv
  ratio.csv
  time.csv
  blast.ncbi2.2.6.blast.ncbi2.2.6.csv
  blast.ncbi2.2.6.dash.r11_mode2.csv
  blast.ncbi2.2.6.dash.r11_mode4.csv
  dash.r11_mode2.blast.2.2.6.csv
  dash.r11_mode2.dash.r11_mode2.csv
  dash.r11_mode2.dash.r11_mode4.csv
  dash.r11_mode4.blast.2.2.6.csv
  dash.r11_mode4.dash.r11_mode2.csv
  dash.r11_mode4.dash.r11_mode4.csv
  ph.blast.ncbi2.2.6.blast.ncbi2.2.6.csv
  ph.blast.ncbi2.2.6.dash.r11_mode2.csv
  ph.blast.ncbi2.2.6.dash.r11_mode4.csv
  ph.dash.r11_mode2.blast.2.2.6.csv
  ph.dash.r11_mode2.dash.r11_mode2.csv
  ph.dash.r11_mode2.dash.r11_mode4.csv
  ph.dash.r11_mode4.blast.2.2.6.csv
  ph.dash.r11_mode4.dash.r11_mode2.csv
  ph.dash.r11_mode4.dash.r11_mode4.csv
```

Figure 3.12: Example Batching Environment Directory Structure After Running Batch: `data` Directory Only.

Table 3.2: List of Statistical Summary Files Produced By Batch Environment.

File	Contents
`lph50s.csv`	PatternHunter Metric Variant; 50% coverage threshold: A sensitivity measure.
`ratio.csv`	The mean ratio of query execution time (ratio calculated per query, then averaged): A speed measure.
`time.csv`	The mean ratio of batch execution time (ratio of sum of run times): A speed measure.

Scores are included per query, and for acceptance thresholds of 1% through 100% of alignment coverage. These data are used to plot the PatternHunter Variant Metric Score versus Coverage Threshold graphs.

The command sequence also generates summary statistics in comma separated value (CSV) format, suitable for the R statistical language[1]. Table 3.2 lists the statistical summary files produced by the commands, and describes the contents of each.

Where large numbers of batches are being performed, with the intention of comparing multiple algorithms against some benchmark, the exhaustive comparison becomes inefficient. This is because of the inherent O(n^2) time complexity: Comparing n batches against n batches requires n^2 operations. To address this, marker files are added to job directories. The `makemake` command can then be told to compare those batches that have a specified marker file against only a single batch, or set of batches, that have another specified marker file. For example, to compare the two DASH batches against the BLAST batch, but not the other way around, create distinct marker files in each batch directory, and then invoke `makemake` appropriately, as in Figure 3.13 on the next page.

This sequence of commands in Figure 3.13 on the following page compares all batches that contain a file named `left` in their batch definition directory, against those that contain a file named `right` in their batch definition directory. This allows selective comparison, and avoids the quadratic time complexity described earlier.

[1]For whatever reason, R requires the “Comma Separated Format” to use semi-colons as field delimiters, rather than commas.

```
# touch blast/ncbi2.2.6/right    # 3a
# touch dash/r11_mode2 left      # 3b
# touch dash/r11_mode4 left      # 3c
# makemake ‘pwd‘ left right      # 3d
# cd data                        # 4a
# make                           # 4b
# cd ..                          # 4c
```

Figure 3.13: Example Command Sequence To Selectively Compare Several Batches.

3.6 Results For Benchmark Algorithms

The following tables and figures present a summary of the performance of each of the algorithms listed earlier in this chapter. In addition to the speed and sensitivity results, the size of the database and index structures, if any, are also included. These tables and graphs are reproduced in later chapters, with the addition of the results of algorithms introduced in this dissertation.

3.6.1 Database And Index Sizes

Tables 3.3, 3.4 and 3.5 list the total database and index size for each algorithm. The totals are broken down to show the space required for the sequence bodies, sequence descriptions and index structures. All figures are listed in absolute terms (megabytes) and normalised terms (bits per base or bits per acid, as appropriate).

3.6.2 Search Speed

Tables 3.6, 3.7 and 3.8 summarise the search speed of each algorithm. Mean, median and total values are given, and the totals are compared against those of NCBI-BLAST 2.2.6.

3.6.3 Search Sensitivity

Tables 3.9, 3.7 and 3.11 presents the sensitivity of each algorithm measured using the PatternHunter variant. The maximum score possible is 100, with the Smith-Waterman algorithm acting as the benchmark. Figures 3.15 and 3.16 show the same data graphically.

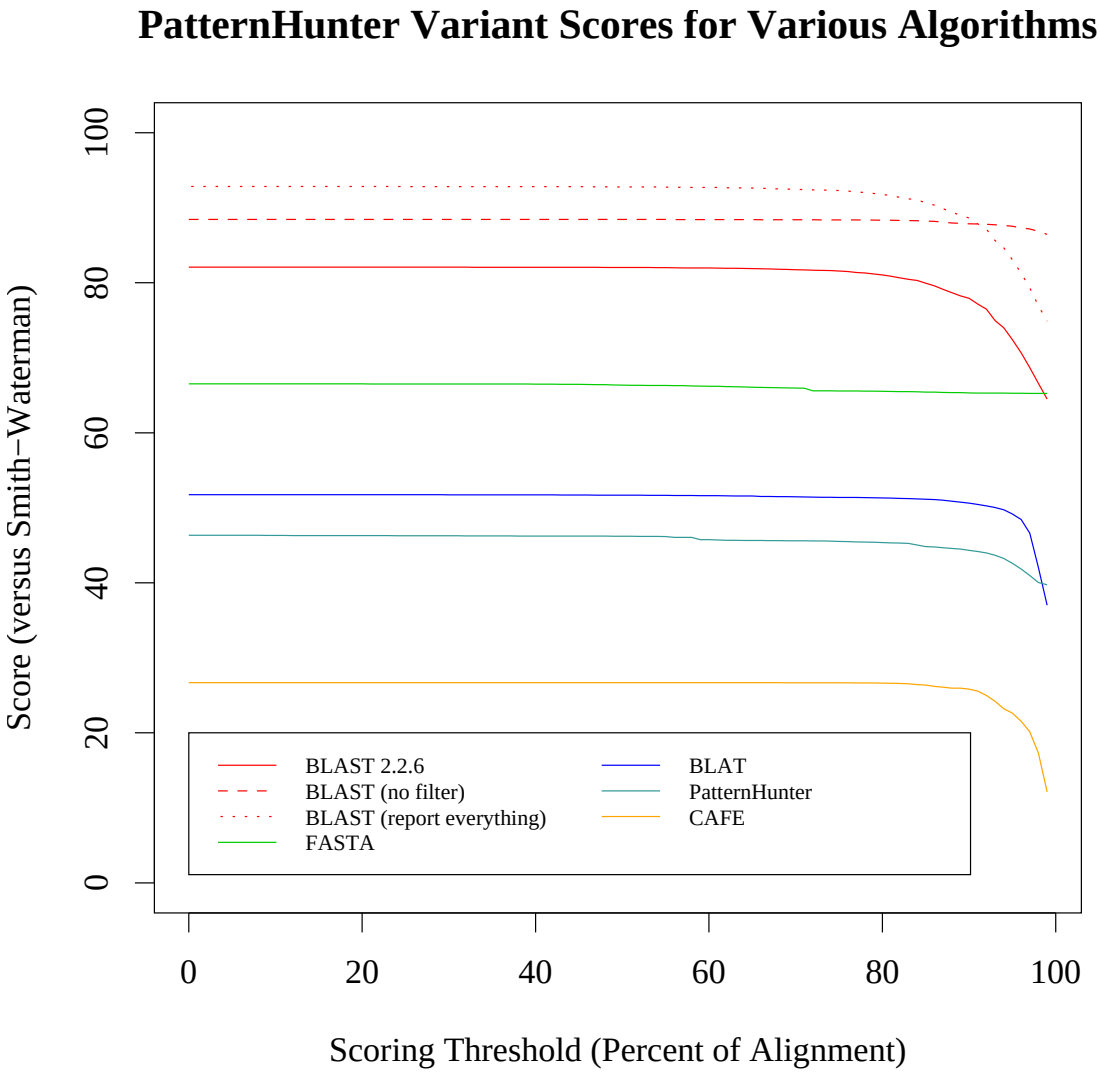

Figure 3.14: PatternHunter Variant Scores (See Section 3.3) Of Algorithms For Nucleic Acid Queries (Against The Human UniGene (Nucleic Acid) Database). The Smith-Waterman algorithm is used as the benchmark.

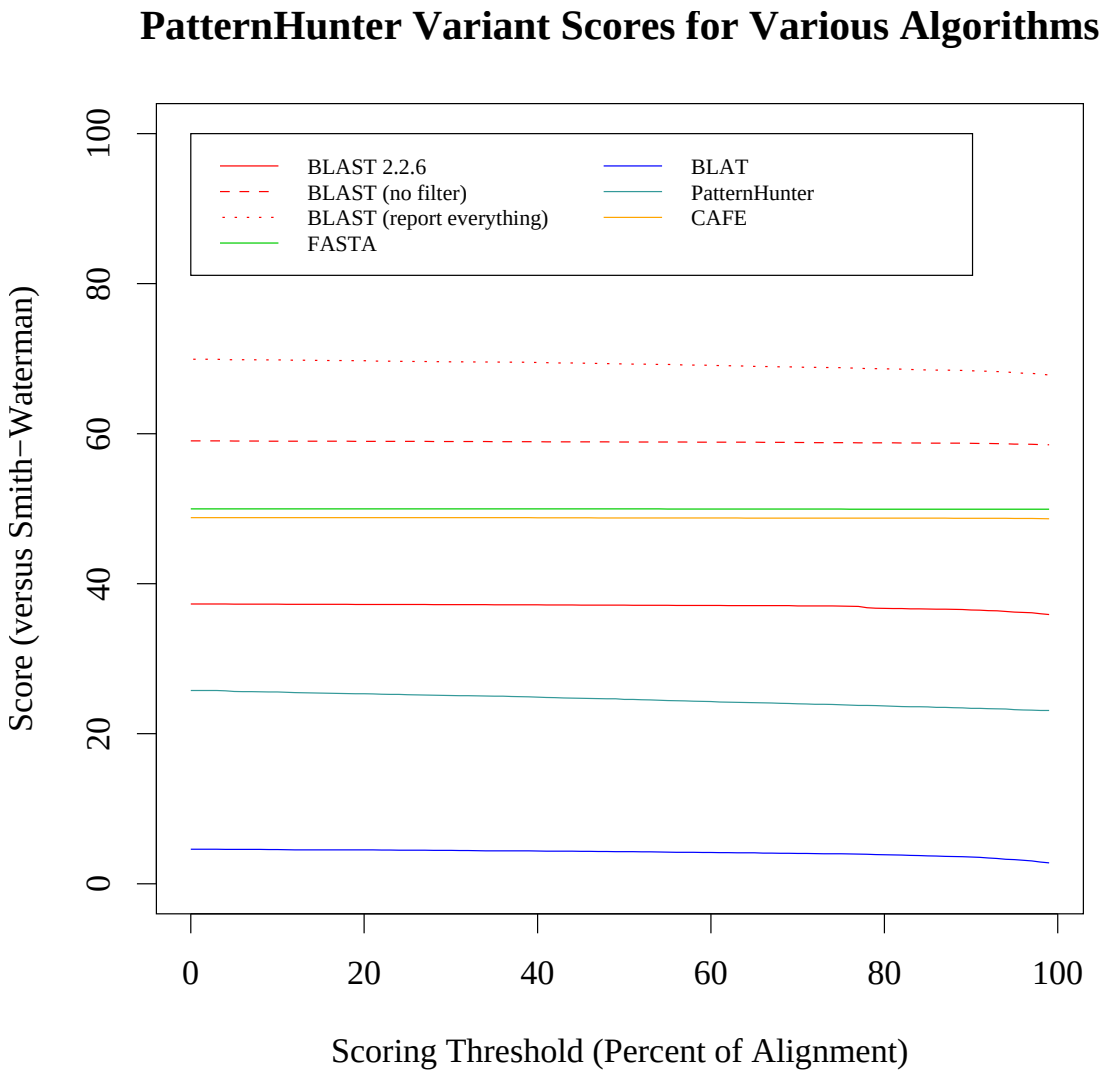

Figure 3.15: PatternHunter Variant Scores (See Section 3.3) Of Algorithms For Nucleic Acid Queries (Against The Human Genome database). The Smith-Waterman algorithm is used as the benchmark.

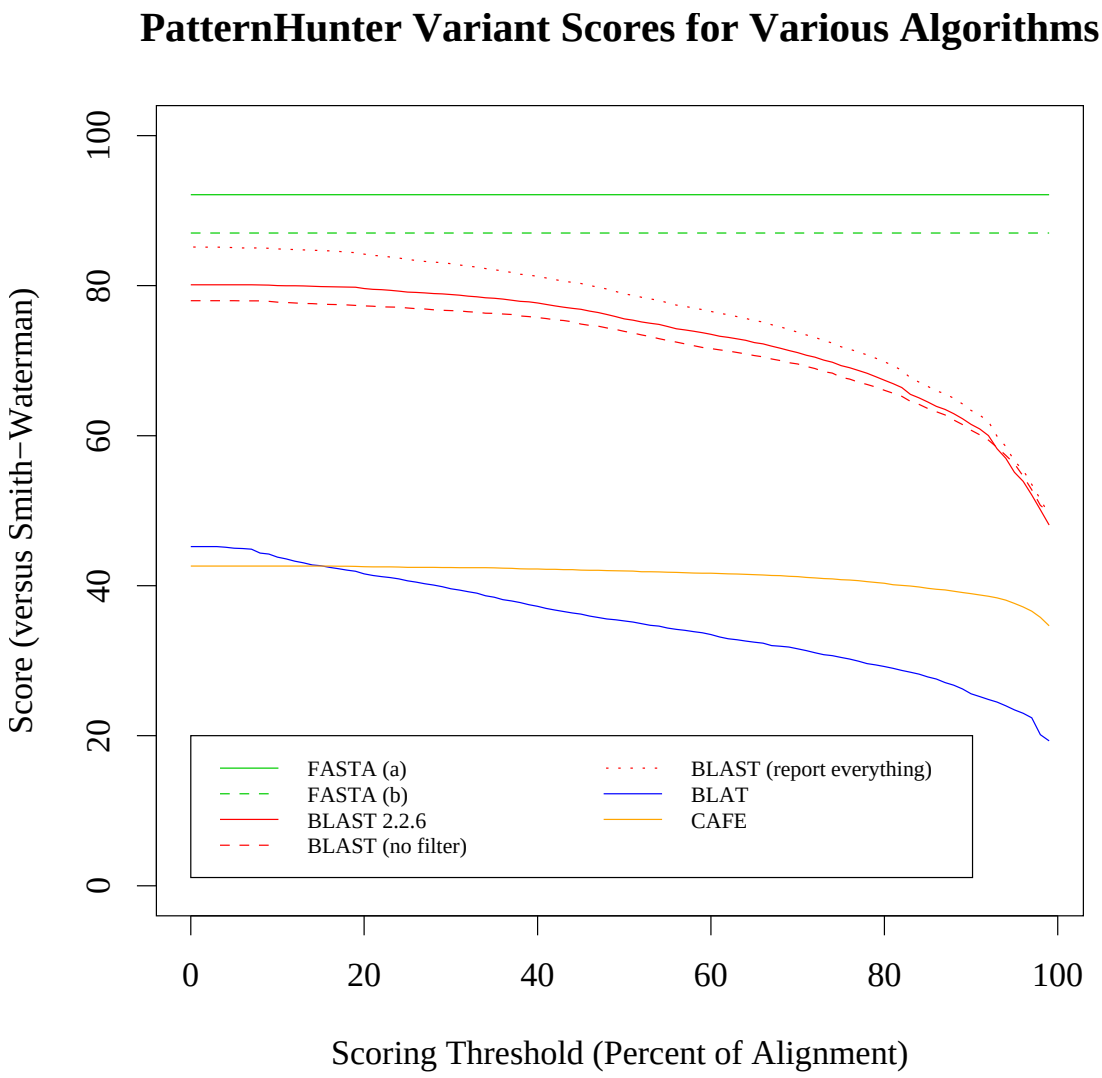

Figure 3.16: PatternHunter Variant Scores (See Section 3.3) Of Algorithms For Protein Queries (Against The GenPept (Protein) Database). The Smith-Waterman algorithm is used as the benchmark.

Table 3.3: Human UniGene (Nucleic Acid) Database And Index Sizes For Surveyed Algorithms In megabytes (MB) And Bits Per Base (B/B)

Format	Bodies Only		Descriptions		Index		Total	
	MB	B/B	MB	B/B	MB	B/B	MB	B/B
Smith-Waterman** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
BLAST (formatdb)	489	2.09	660	2.83	40	0.17	1,189	5.09
BLAT* (faToTwoBit)	630	2.70	-	-	1,088	4.66	1,718	7.36
PatternHunter** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
FASTA** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
CAFE*** (CAFE Index)	496	2.12	102	0.44	6,961	29.79	7,634	32.67

* Indicates that program maintains an index in RAM, and that the database format contains both sequence bodies and descriptions (BLAT).

** Indicates that algorithm indexes during searching (PatternHunter and FASTA), or does not use an index (Smith-Waterman).

*** Indicates that multiple small indices were used instead of one large index, due to technical difficulties (CAFE).

Table 3.4: Human Genome Nucleic Acid Database And Index Sizes For Surveyed Algorithms In Megabytes (MB) And Bits Per Base (B/B)

Format	Bodies Only		Descriptions		Index		Total	
	MB	B/B	MB	B/B	MB	B/B	MB	B/B
Smith-Waterman** (FASTA ASCII)	3,004	8.21	289	0.79	-	-	3,293	9.00
BLAST (formatdb)	736	2.01	433	1.18	32	0.09	1,201	3.28
BLAT* (faToTwoBit)	950	2.60	-	-	1,867	5.10	2,817	7.70
PatternHunter** (FASTA ASCII)	3,004	8.21	289	0.79	-	-	3,293	9.00
FASTA** (FASTA ASCII)	3,004	8.21	289	0.79	-	-	3,293	9.00
CAFE*** (CAFE Index)	987	2.70	9	0.02	9,950	27.18	10,945	29.90

* Indicates that program maintains an index in RAM, and that the database format contains both sequence bodies and descriptions (BLAT).

** Indicates that algorithm indexes during searching (PatternHunter and FASTA), or does not use an index (Smith-Waterman).

*** Indicates that multiple small indices were used instead of one large index, due to technical difficulties (CAFE).

Table 3.5: Protein Database And Index Sizes For Surveyed Algorithms In Megabytes (MB) And Bits Per Acid (B/A) (Against The GenPept (Protein) Database).

Format	Bodies Only		Descriptions		Index		Total	
	MB	B/A	MB	B/A	MB	B/A	MB	B/A
Smith-Waterman* (FASTA ASCII)	479	8.15	142	2.42	-	-	621	10.57
BLAST (formatdb)	473	8.05	194	3.31	231	3.94	899	15.31
BLAT* (FASTA ASCII)	479	8.15	142	2.42	-	-	621	10.57
FASTA* (FASTA ASCII)	479	8.15	142	2.42	-	-	621	10.57
CAFE** (CAFE Index)	480	8.17	22	0.37	1,621	27.59	2,236	38.06

* Indicates that algorithm indexes during searching (FASTA), or does not use an index (Smith-Waterman).

** Indicates that multiple small indices were used instead of one large index, due to technical difficulties (CAFE).

Table 3.6: Comparison Of Search Speed For Various Algorithms Against The Human UniGene (Nucleic Acid) Database.

Format	Search Time (seconds)			Search Time
	mean	median	total	(×BLAST)
Smith-Waterman	16,260	14,070	3,251,827	1660.56
NCBI-BLAST 2.2.6 (Default)	9.79	9.40	1,958.27	1.00
NCBI-BLAST 2.2.6 (No Filter)	21.37	10.16	4,274.56	2.18
NCBI-BLAST 2.2.6 (Report Everything)	49.93	11.33	9,985.01	5.10
BLAT*	2.10	2.07	471	0.21
PatternHunter**	78.37	78.61	15,673.57	8.00
FASTA	530.05	534.69	106,010.29	54.13
CAFE***	32.75	30.76	6,693.46	3.42

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 3.7: Comparison Of Search Speed For Various Algorithms Against The Human Genome Database.

Format	Search Time (seconds)			Search Time (×BLAST)
	mean	median	total	
Smith-Waterman	25,040.00	21,620.00	5,008,305.00	2165.13
NCBI-BLAST 2.2.6 (Default)	11.57	10.95	2,313.17	1.00
NCBI-BLAST 2.2.6 (No Filter)	22.15	11.30	4,430.75	1.92
NCBI-BLAST 2.2.6 (Report Everything)	232.20	12.14	46,434.82	20.07
BLAT*	3.91	3.87	1082.22	0.47
PatternHunter**	164.80	15.30	32,957.64	14.25
FASTA	611	599.9	122,195	52.83
CAFE***	30.52	25.27	6,104.9	2.64

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 3.8: Comparison Of Protein Search Speed For Various Algorithms (Against The GenPept (Protein) Database).

Format	Search Time (seconds)			Search Time
	mean	median	total	(×BLAST)
Smith-Waterman	1674.00	1397.00	334,794.20	66.32
NCBI-BLAST 2.2.6 (Default)	25.24	22.01	5,047.81	1.00
NCBI-BLAST 2.2.6 (No Filter)	35.33	26.48	7,066.00	1.40
NCBI-BLAST 2.2.6 (Report Everything)	71.00	24.55	14,200.32	2.81
BLAT*	85.45	80.40	17,004.37	3.39
FASTA (a)	296.00	273.36	59,199.23	11.73
FASTA (b)	83.36	84.38	16,672.14	3.30
CAFE***	11.88	10.25	2,375.71	0.47

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 3.9: Nucleotide Sensitivity Scores (PatternHunter Variant) For Various Algorithms Versus The Results Of The Smith-Waterman Algorithm (Against The Human UniGene (Nucleic Acid) Database).

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
Smith-Waterman	100	100	100	100	100
BLAST (Default)	82.04	81.64	78.69	74.97	66.58
BLAST (No Filter)	88.43	88.38	87.96	87.73	86.86
BLAST (Report Everything)	92.78	92.36	89.41	85.62	76.96
FASTA	66.41	65.59	65.36	65.28	65.25
BLAT	51.69	51.41	50.88	50.04	42.06
PatternHunter	46.22	45.58	44.59	43.67	40.05
CAFE	26.69	26.68	25.96	24.2	17.32

Table 3.10: Nucleotide Sensitivity Scores (PatternHunter Variant) For Various Algorithms Versus The Results Of The Smith-Waterman Algorithm (Against The Human Genome Database).

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
Smith-Waterman	100	100	100	100	100
BLAST (Default)	37.14	37.03	36.58	36.4	35.99
BLAST (No Filter)	58.9	58.81	58.73	58.67	58.54
BLAST (Report Everything)	69.33	68.82	68.44	68.28	67.92
FASTA	49.97	49.95	49.94	49.93	49.92
BLAT	4.29	4	3.62	3.37	2.9
PatternHunter	24.66	23.91	23.47	23.32	23.11
CAFE	48.77	48.74	48.73	48.71	48.69

Table 3.11: Protein Sensitivity Scores (PatternHunter Variant) For Various Algorithms Versus The Results Of The Smith-Waterman Algorithm (Against The GenPept (Protein) Database).

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
Smith-Waterman	100	100	100	100	100
FASTA (a)	92.12	92.12	92.12	92.12	92.12
FASTA (b)	87.02	87.02	87.02	87.02	87.02
BLAST (Default)	76.15	70.08	62.92	58.27	50.13
BLAST (No Filter)	74.35	68.55	62.09	58.44	50.72
BLAST (Report Everything)	79.56	72.69	64.98	59.93	51.5
BLAT	35.56	30.82	26.73	24.48	20.15
CAFE	42.02	40.96	39.26	38.38	35.77

Part II

Cooperative Compression Of Redundant Proteomic Databases

Chapter 4

DASH: Search & Alignment For Cooperatively Compressed Databases And Indices

Introduction

Cooperative compression of database and index ensembles relies on recurrent strings. Recurrent strings in the database are compressed by storing only one instance of each recurrent string, and recording the other instances using a reference to the stored instance. By always storing the first instance of a recurrent string, and allowing references to earlier instances, it is possible to create chains of references while avoiding the introduction of cycles.

The benefits of chained references are that: (a) only one (the first) instance of the string needs to be stored in the database, and; (b) only one (the last) instance requires posting in the index, as the chain of references can be followed to recursively compute the address of the other instances, regardless of the length of the reference chain. These benefits combine to reduce the size of the compressed database and index ensemble.

There are two challenges that must be faced by any sequence search and alignment algorithm that is paired with cooperatively compressed database and index ensembles: (a) the effective and efficient discovery of recurrences, and; (b) the efficient use of recurrences.

Effective And Efficient Discovery Of Recurrences

To use the recurrence records of a cooperatively compressed database and index ensemble, an algorithm must find alignments that intersect with at least one instance of any given recurrence. Thus, it is advantageous for such an algorithm to find the maximum extent of each alignment. However, as shown by Figures 4.1 and 4.2, heuristic sequence search and alignment algorithms do not always find the full extent of each alignment. As measured by the PatternHunter variant metric, the level of recall drops off as the alignment fraction threshold (i.e., the fraction of a given alignment that must be discovered for it to be counted) is increased. Also, the faster algorithms typically obtain lower scores than the more thorough algorithms, i.e., the faster and less sensitive BLAT and CAFE, versus the slower and more sensitive BLAST and FASTA.

Protein searching stresses different aspects of a search algorithm, and this is reflected in the results where the relative ranking of algorithms alters. For protein searching, FASTA is the most sensitive. However, this thoroughness comes at a considerable speed penalty. CAFE is much faster, and has the most sustained PatternHunter score for protein, suggesting that it would be able to identify almost all recurrences. However, CAFE gains its speed by constructing alignments against very few database records. Therefore, while CAFE has both acceptable run time as well as the most sustained scores, its overall sensitivity is relatively poor in this context. Therefore, because of their poor speed to sensitivity ratios, neither FASTA nor CAFE are attractive as a basis for searching cooperatively compressed database and index ensembles. PatternHunter is also rejected, for the same reason.

Efficient Use Of Recurrences

Assuming that an algorithm can discover recurrences, the second challenge is to be able to make efficient use of them. Cloning of an alignment that intersects a recurrence into its alternative locations should, ideally, make use of the alignment that has been obtained

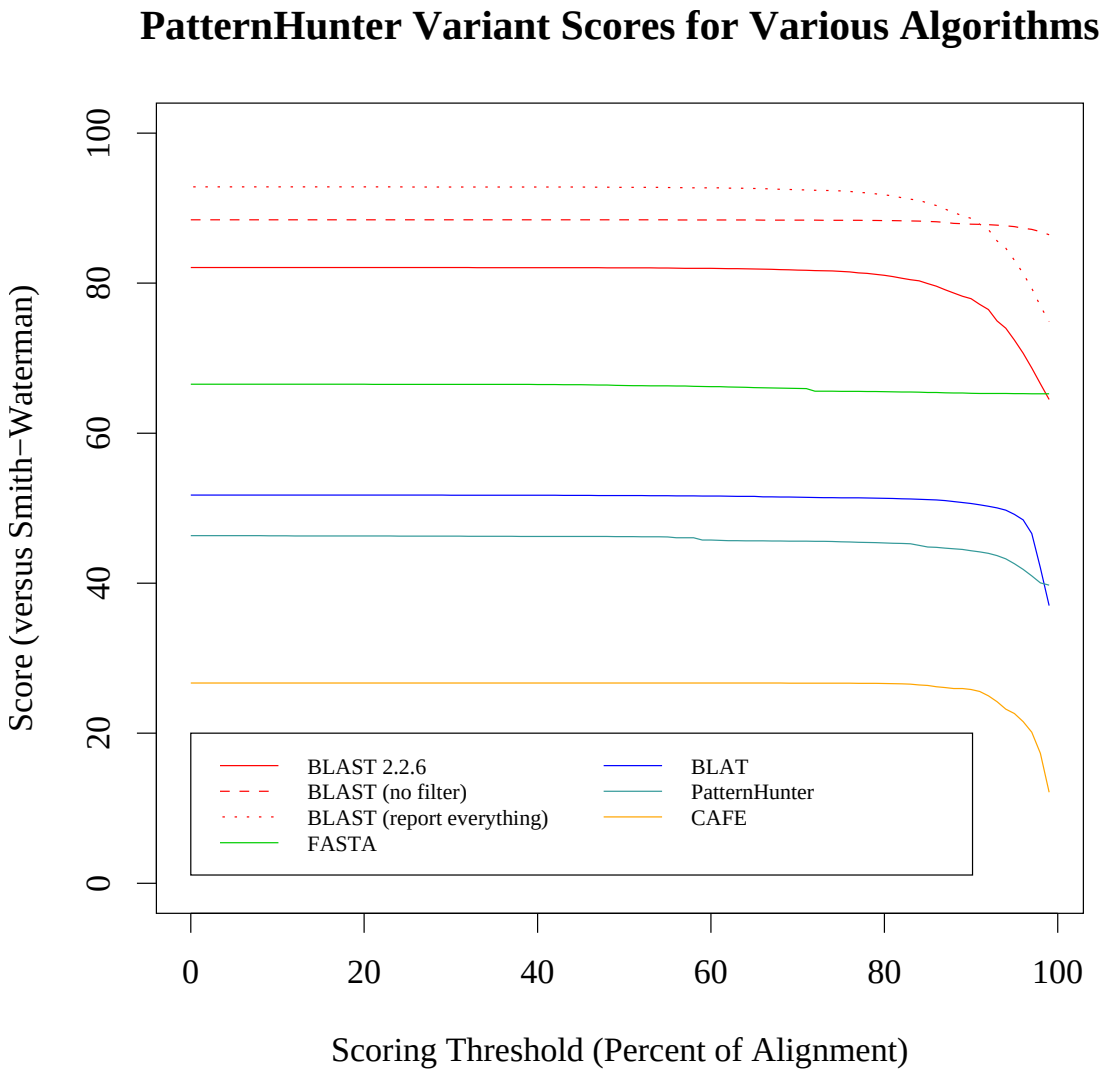

Figure 4.1: PatternHunter Variant Scores (See Section 3.3) Of Various Algorithms (Nucleic Acid) (Against The Human UniGene (Nucleic Acid) Database). The Smith-Waterman algorithm is used as the benchmark.

Table 4.1: Protein PatternHunter Scores 100% Required (PH(100%)), And As A Fraction Of PatternHunter Score 50% Required (PH(50%)) (Against The GenPept (Protein) Database).

Algorithm	PH(50%)	PH(100%)	PH(100%)/PH(50%)
FASTA	92.12	92.12	100%
CAFE	42.02	35.77	85.1%
BLAST	79.56	51.50	64.7%
BLAT	35.56	20.15	56.6%

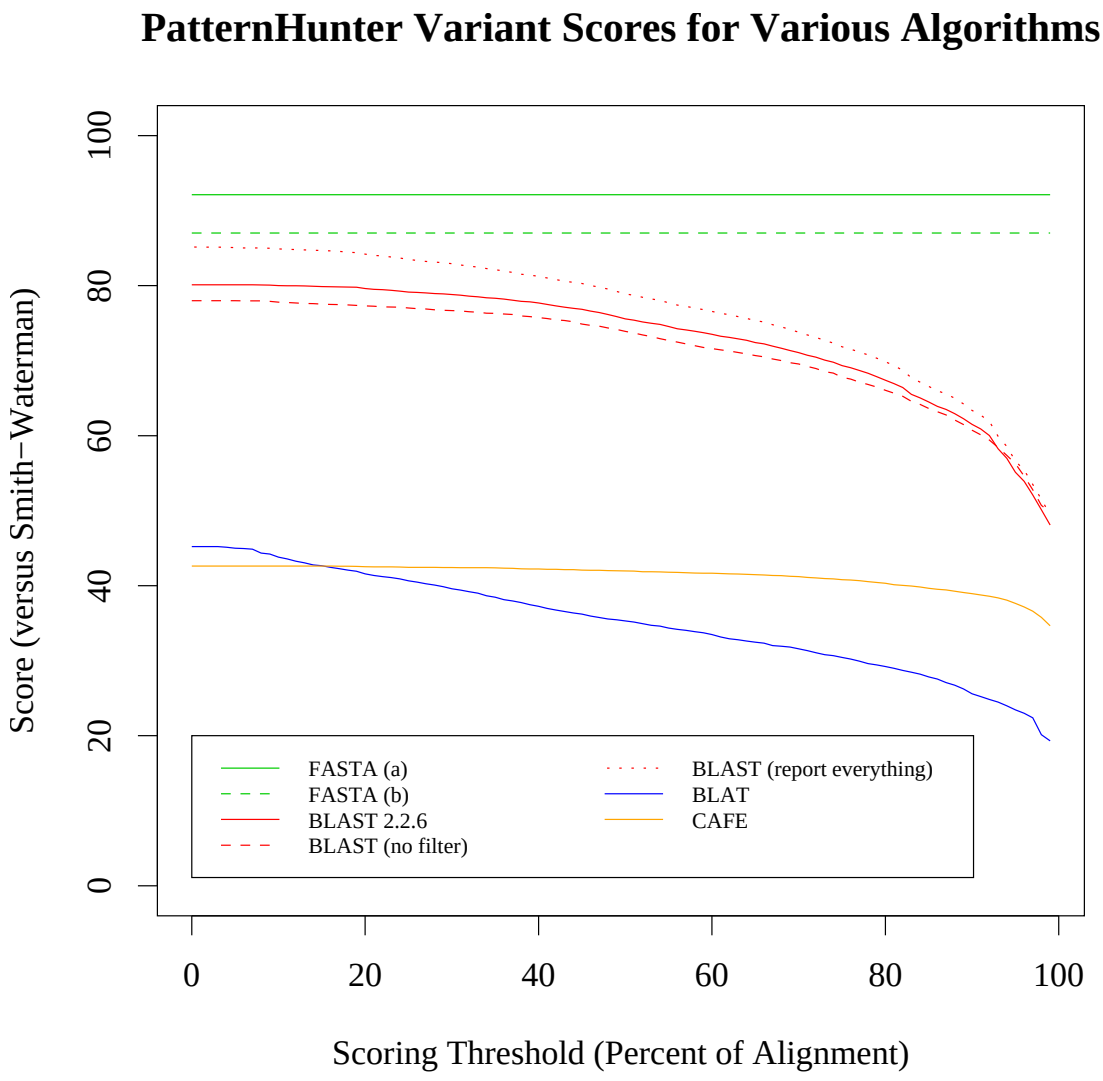

Figure 4.2: PatternHunter Variant Scores (See Section 3.3) Of Various Algorithms (Protein) (Against The GenPept (Protein) Database). The Smith-Waterman algorithm is used as the benchmark.

from the context where it was discovered. This information has been obtained at some computational cost. In the interests of efficiency, such effort should be reused rather than duplicated. It is straight forward to do this by translating the score and extents of the alignment into each new context. This eschews the need to repeat the alignment process. A suitable algorithm would make effective use of such partial alignments, also known as *High Scoring Pairs* (HSPs), when constructing a complete alignment, rather than, for example, performing dynamic programming extension from a central seed, as does BLAST.

This approach of reusing search effort has been explored in the literature, e.g., Kent (2002), Brudno et al. (2003), Kurtz et al. (2004), Li et al. (2004), although usually with the intention of reducing run time. Of the surveyed algorithms, BLAT (Kent 2002) is typical of this approach. While it makes efficient use of HSPs to minimise dynamic programming during assembly, the resulting assembly process is not thorough. As a result, its ability to detect the entirety of an alignment is noticeably compromised, especially when applied to protein searching (Table 4.1).

More precisely, approaches similar to that of BLAT are very good at identifying the regions of near-identity with respect to a given query sequence, but are less proficient at assembling a unified alignment from such fragments. The intervening regions of lower, but perhaps still significant similarity are of little concern to them, and so are neglected in order to obtain fast run times. The emphasis algorithms like BLAT place on regions of near-identity is not surprising, since the primary focus of such algorithms is typically aligning entire genomes, not individual sequences: they are operating at a different scale.

In the context of a cooperatively compressed index, however, insufficient assembly translates into short alignments, which in turn reduces the number of recurrences that will be intersected, resulting in a cascading loss of sensitivity. Therefore the correct assembly of regions of moderate similarity remains of importance in this application. Therefore, because they cannot make effective reuse of search effort, neither BLAST or BLAT are

attractive as a basis for searching cooperatively compressed database and index ensembles in this dissertation.

Summary

To summarise, none of the surveyed algorithms combine reasonable speed with the appropriate sensitivity characteristics that are desirable for the construction and searching of cooperatively compressed databases and indices: FASTA is too slow, PatternHunter and BLAT are too insensitive, CAFE considers too few database records, and BLAST cannot make efficient reuse of alignments against HSPs. Therefore, a new algorithm, DASH (Gardner-Stephen and Knowles 2003, Gardner-Stephen and Knowles 2004, Knowles and Gardner-Stephen 2006, Australian Provisional Patent 2003907016, US Patent Application US 60/637 630, and US Patent Application US 11/019 807) was created, with the following design objectives:

- Identification of as many HSPs as possible in order to fully leverage recurrence records, and thus identify additional alignments;
- Identification of the maximum extent of each HSP, also in order to fully leverage recurrence records in order to identify additional alignments;
- The ability to make efficient re-use of partial alignments when they are identified, so that execution time is contained when recurrences are cloned into new contexts; and
- Comparable time and sensitivity characteristics to the state of the art.

These objectives are addressed in this chapter by creating an effective alignment assembly and extension scheme that aims to maximise alignment coverage, while minimising the computational cost of excessive dynamic programming. This is done by making use of ungapped alignments, also called *diagonals*, and hence the name of the *Diagonal Assembling Search Heuristic* (DASH).

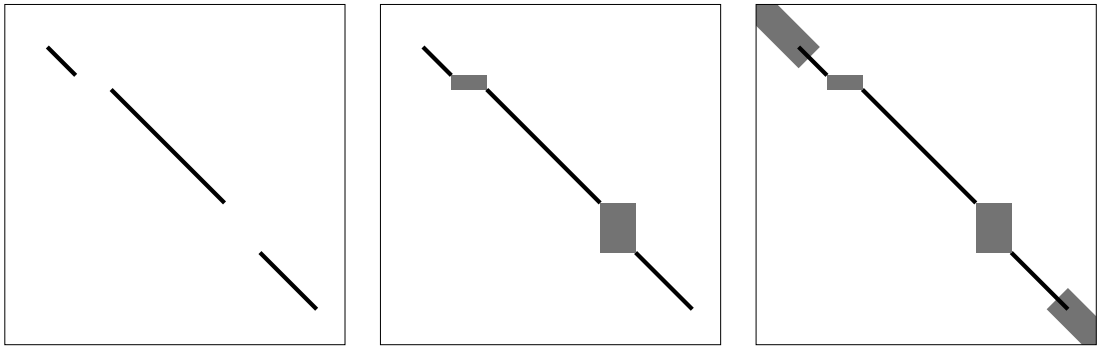

Figure 4.3: The Three Stages Of The DASH Sequence Alignment Algorithm.
Un-gapped alignment occurs to first discover HSPs in Stage 1. The set of HSPs are then optimally assembled in Stage 2. Finally, banded dynamic programming is performed at each end to polish the alignment in Stage 3.

4.1 The DASH Algorithm

The DASH algorithm produces alignments via a three stage process as depicted in Figure 4.3. Un-gapped alignments (HSPs) are first identified using an index (Stage 1), then global sequence alignment is performed for bounded regions between nearby HSPs to produce gapped alignments, which are then assembled using a dynamic programming algorithm (Stage 2). Finally, banded local dynamic programming is performed at both ends of each alignment (Stage 3) to maximise the score and extent of the final product.

As mentioned, the general approach of minimising dynamic programming by chaining sub-alignments is not new, e.g., Kent (2002), Brudno et al. (2003), Kurtz et al. (2004), Li et al. (2004). However, reflecting the unique environment in which it must function, DASH includes a novel combination of mechanisms to reduce run time. This speed advantage is then traded off in order to attain a suitable level of sensitivity by dedicating substantial effort to the optimal assembly of HSPs into gapped alignments. This ensures that the maximum extent of each alignment is discovered in the majority of cases so that extensive use can be made of recurrence records to discover new alignments.

4.1.1 Stage 1: Searching For Non-Gapped Alignments

DASH identifies all non-gapped aligning fragments between the query and all records in a database by using an exhaustive index, in the form of an inverted file, which contains postings for every occurrence of all k-mers that occur in the database, where k is fixed at index construction time. For nucleic acid databases, $k = 8$ was selected by empirical process to balance search time and sensitivity. For protein databases, $k = 3$.

There are multiple ways of using an inverted file to identify alignments. One approach is that used by CAFE (Williams and Zobel 2002a), where the FRAMES structure is used to measure the number of index postings that point to a given database record, or region of a database record. The gathered information is used to identify a set of database records that are likely to contain significant alignments. This approach has the advantage that it does not require the database record bodies to be extracted during this initial process, making it particularly attractive for combination with a compressed database representation.

However, the advantages of the FRAMES method are negated if the database record bodies are required in order to extract the inverted lists that drive the process. This is precisely the situation in the case of cooperative compression, where portions of the index are computed by consulting the recurrence records in the compressed database. Therefore DASH cannot use the FRAMES structure, and instead attempts to generate an HSP for every index posting, or *seed*, that is retrieved.

This indicates a preliminary computational complexity for the seed discovery that is directly and positively proportional to the database size, and exponentially and negatively proportional to the index width. The cost of extending each seed into an HSP will be proportional to the length of the HSP. However, for long HSPs there will be many seeds that identify it. Thus for searches involving many long HSPs the search time will contain a quadratic component. This is undesirable, and to avoid it DASH filters out redundant seeds

that point to HSPs that have already been discovered. The mechanism for this is described in Sub-Section 4.1.1.4.

If index postings are considered as independent alignment candidates, it is relatively inexpensive to perform a BLAST 1 style, i.e., ungapped, extension of each. This allows the discovery of non-exact alignments, also known as High Scoring Pairs (HSPs), which if they span a recurrence, can be translated into other contexts. An HSP is formed for each posting by extending an alignment seed until the maximum score is attained. Extension is terminated by a heuristic, rather than continuing exhaustively. DASH's termination heuristic for nucleotide extension is the presence of two consecutive substitutions, while for protein alignment the heuristic is the presence of two consecutive non-conserving (i.e., negatively scoring) substitutions.

The use of a heuristic sacrifices optimality for computational efficiency. In practice, the sensitivity loss is likely to be negligible, as any sufficiently long alignment will be covered by more than one posting, and hence is very unlikely to escape notice, although it may be discovered as two or more smaller fragments. In DASH, if this occurs, such fragments are likely to be reassembled during the assembly stage.

4.1.1.1 Addressing Selectivity

Achieving acceptable selectivity is also possible when performing an alignment on each candidate, as the same alignment scoring system that is later used to produce the final alignment scores can be used. Refinements to this procedure are introduced that, to use accounting parlance, "write down" the value (score) of certain low complexity alignments relative to others, in preference to the use of query filters that may prevent the discovery of the full extent of an alignment. Explanation of this procedure is followed by a description of the techniques employed to reduce the number of index postings that require evaluation, and so minimise computational complexity.

DASH does no pre-filtering of the query sequence before searching. While aiding sensitivity, and the discovery of alignments spanning recurrence records, it has the potential to result in poor selectivity (i.e., low precision), as low complexity regions in the query can cause any number of irrelevant hits to be ranked ahead of relevant results.

An alternative approach to improving selectivity that is implemented in DASH is to filter those alignments that do get discovered, and ranking those alignments that appear to be low-complexity behind others. This can be implemented by reducing the overall alignment score of alignments containing wild-cards. While not sensitive to all manifestations of low-complexity sequence, it is particularly well suited to addressing the problem of runs of the letter `N` and other wild-cards that occur in many nucleic acid databases. (For protein alignment the use of a scoring matrix obviates the need for any special treatment of wild-cards.)

It is true that this could be realised by the use of a matrix for nucleotide comparisons, instead of a simple reward and penalty scheme. However, the use of a matrix is relatively expensive. Nucleotide sequences can be compared more rapidly by using a look up table, e.g., as in FASTA.

The look up table can also be used to trigger the termination of extension by comparison to the minimum score that can be obtained without triggering the termination condition. If the look up table returns a score less than some trigger value, then extension terminates. If the look up table uses a two bits per base representation, it is practical to simultaneously compute the alignment of 8 bases using a 16 bit look up table. Such a table is fed the binary XOR of the relevant query and database record sequence segment (To search thoroughly, 8 copies of the query sequence must be maintained, one for each phase, an issue that is explained in more detail later). Ones in the XOR will correspond to substitutions in the sequence, and so the alignment score can be computed. Such a look up table fits easily into the cache of a modern processor.

However, the 2-bit look up table approach has difficulty in differentially scoring wild-cards. It is possible to approximate such scoring by randomly casting each wild carded base to exactly one of the possible bases, and then use that as the basis for comparison and scoring. However, this leads to difficulties due to natural random variation, where alignments may be terminated or continued incorrectly.

These difficulties can be solved by using a four bits per base representation where each of the nucleotides is represented using a One Hot Coding (i.e., a coding where only one bit is set in each code, as also used by NCBI-BLAST). The IUPAC codes corresponding to wild cards are logically represented as the union of the codes for each base permitted by that code, as in Table 4.2. The query and subject sequences are now combined using binary AND instead of XOR. Assuming the same 16 bit wide look up table, only four instead of eight comparisons can be done in parallel. However, all possible combinations of wild card base comparison are correctly resolved, as illustrated in the example of Figure 4.4.

Figure 4.4 shows the calculation of the dynamic programming score for the oligonucleotides `ATCG` versus `ANGG`. `A` versus `A` is a match, because the bit corresponding to `A` in each is set. `T` vs `N` is also identified as a match, because the code for `N` has the bit for `T` set, and so the AND results in a non-zero value, which the look up table will treat as a match. However, the codes for `C` versus `G` have no bits in common, so the result is zero, which the look up table will treat as a substitution. Finally, `G` versus `G` is counted as a match for the same reason as `A` versus `A`. Three matches score $3 \times 1 = 3$, and the one substitution scores $1 \times (-4) = -4$, giving a total score of $3 - 4 = -1$.

However, scores that are generated using the above method will not differ for alignments containing wild cards. This can be addressed by using a second look up table that is fed the binary OR of the sequences, and so estimates the probability of a match. This is used to write down (in the accounting sense) the score obtained from the first look up table. The estimate is calculated by counting the number of set bits for each base, and reducing the

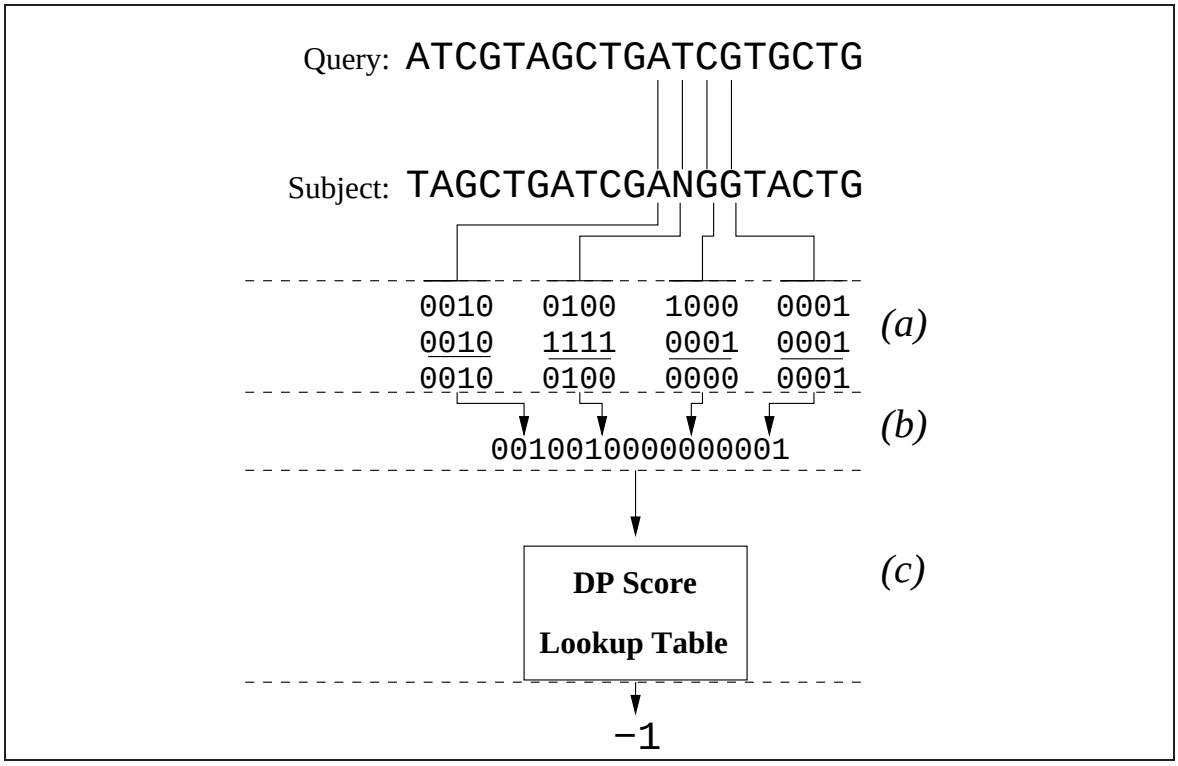

Figure 4.4: Table Look Up For Un-Gapped Alignment And Score, Where Exact Matches Are Scored As +1 And Substitutions As -4.
Each four base segment is compared by: a) ANDing the four bit One Hot Coding of each base; b) Collating these values into a 16 bit value, and c) using that value to calculate the Dynamic Programming Score, by way of a look up table.

Table 4.2: IUPAC-IUB Codes And Their 4-bit Representations.

IUPAC	Base	Description	4-bit
G	Guanine	G	0001
A	Adenine	A	0010
T	Thymine	T	0100
C	Cytosine	C	1000
R	Purine	A or G	0011
Y	Pyrimidine	C or T	1100
M	Amino	A or C	1010
K	Ketone	G or T	0101
S	Strong interaction	C or G	1001
W	Weak Interaction	A or T	0110
H	Not-G	A, C or T	1110
B	Not-A	C, G or T	1101
V	Not-T	A, C or G	1011
D	Not-C	A, G or T	0111
N	Any	A, C, G or T	1111
X	–	–	0000

reward score by a corresponding fraction, according to Table 4.3: If only one bit is set the full reward is conferred; only half the reward is granted if two bits are set, as a match is twice as likely; if three bits are set, then only a third of the reward should be conferred, as a match is three times as likely, finally, if all bits are set, then no reward is given since a match is certain. Thus, the maximum score for any sequence aligned against a string of Ns will be zero.

Alternatively, a single look up table can be used to return the inflated score, and the write down process can be performed when computing the final score of each alignment after assembly. The DASH algorithm uses the latter approach of writing down when computing the final score of the alignment after assembly, because: (a) it is simpler to implement, requiring only one look up table during alignment; and, (b) assuming that some alignments are discarded during the assembly phase, it results in less work overall.

Table 4.3: Differential Scoring Against Wild Card Bases.

Bits clear in base	P(mismatch)	% of Reward
3 (A, C, G, T)	3/4	100
2 (R, Y, M, K, S, W)	2/4	50
1 (H, B, V, D)	1/4	33.3
0 (N)	0/4	0

4.1.1.2 Stop Words

It is possible to filter out stop words, i.e., excessively frequent terms in the database, to accomplish a similar role to that of a low complexity filter, like that of CAFEFILTER (Williams 1999). This makes the assumption that low complexity regions contain motifs that occur with excessive frequency. If an alignment consists entirely of such regions, then no index postings that point to the alignment will be considered, and so the alignment will be ignored. Conversely, if the alignment contains a mixture of low and high complexity regions, then the high complexity regions will be posted in the index; only one such posting is required for the alignment will be discovered in its entirety.

This requirement for an alignment to contain a high complexity region (which in the current context means a region that contains motifs that do not occur with excessive frequency) appears to exclude a significant fraction of meaningless alignments, without missing too many relevant ones. The definition of excessive frequency has been visited only empirically in this dissertation, with exclusion thresholds of $2.5 \times E$ for nucleic acids and $10 \times E$ for proteins, where E is the frequency expected by random, providing a reasonable trade-off of search sensitivity versus selectivity and index size.

It is acknowledged that the filtering of stop words makes the servicing of certain queries inefficient or impossible (Witten et al. 1999). However, for a heuristic search algorithm this would appear to be a reasonable trade-off, because: (a) the guarantee of exhaustive search has already been lost; and, (b) the desire to exclude low complexity sequences from the results implies that low complexity queries should also be ignored.

A significant benefit of this method versus the commonly used filters SEG (Wootton and Federhen 1993, 1996), DUST (Hancock and Armstrong 1994) and XNU (Claverie and States 1993), is that in no case is only part of an alignment returned, because the masking occurs against the index, and not the query or database itself. This is of particular concern for searching the cooperatively compressed databases and indices of this dissertation, where discovery of the maximum extent of an alignment is of importance. It also has the beneficial side effect of significantly reducing the number of postings that must be stored and evaluated, so reducing index size and execution time. Other opportunities for reducing the number of index postings that must be processed are now considered.

4.1.1.3 Limiting Alignment Numbers In Flight

The number of expected alignments of some length, l, in a random database of n words and alphabet size a, is $\frac{n}{a^l}$. For nucleic acid databases n is typically very large, often $\gg 10^9$, and $a = 4$. This means that if l is allowed to be small, there will be a vast population of alignments, adversely affecting both memory requirements and search time. To ameliorate this situation, two methods are employed in DASH.

The first measure is to text-partition the database into manageable sized units: 10^7 letters or 10^5 database records, which ever occurs first. This limits the number of alignments required in memory at any given point in time.

The second measure is to place a lower limit on l, such that a balance between resource requirements and sensitivity is achieved. The range $10 \leq l \leq 14$ was determined empirically to be reasonable for both nucleic acid and protein alignments, with the selection of a specific value dictating the trade-off between sensitivity and execution speed.

By limiting the number of alignments that must be held in memory at one time in this way, the foundation is being prepared for the creation of an algorithm that has at worst a linear relationship between database size and search time.

4.1.1.4 Suppression Of Repeated Discovery Of Long Alignments

For long exact or near-exact alignments, there will be many (possibly hundreds or thousands) of index postings pointing to the one alignment. If each of these postings were allowed to trigger the calculation of what must be an identical alignment, computation would become extremely inefficient. To prevent this, a list of previously discovered alignments is maintained. The list is consulted whenever an index posting is considered: If the candidate lies within the bounds of any alignment found in the list, it is suppressed, as it would only rediscover the alignment that it lies within.

A naive implementation of the suppression list requires $O(n)$ time to consult the list for each index posting considered, where n is the number of alignments discovered. This would be reduced to $O(log_2 n)$ if a binary tree were used. However, by ordering the list of alignments into the same order that the index postings will be considered, and proceeding through the list in tandem with the inverted list for each term, the cost can be effectively reduced to $O(n)$ per *term* (the list is traversed only once per term). The cost is thus reduced to $O(\frac{n_{items\,in\,list}}{n_{postings\,per\,term}})$. If the number of postings per term is equal to or greater than the number of items in the list, then $\frac{n_{items\,in\,list}}{n_{postings\,per\,term}} < 1$, and hence the cost per posting is at worst $O(1)$. This scheme is implemented in the DASH algorithm.

4.1.1.5 Locally Adaptive Query Striding

There is a cost associated with retrieving the inverted list for each term (i.e., k-mer) of the query. Similarly, there is a real cost in evaluating the set of postings that each inverted list contains. It is possible to reduce this cost by considering only some arbitrary fraction of the query terms. This will be at the cost of sensitivity, because alignments will only be detected if an exact correspondence exists in k consecutive residues of the query and database records. Reducing the degree of overlap of each query term used diminishes the ability to detect short alignments.

Previous work described by Barton (1996) has shown that if too few query terms are searched, sensitivity suffers substantially. None the less, it is reasonable to allow DASH to optionally exclude some query terms, since moderate striding over query terms can realise helpful increases in speed, without causing intolerable harm to sensitivity. This is termed *query striding*.

The simplest form of query striding is to use a stride length, S, where $S-1$ terms are excluded following each term that is retained. Therefore a stride length of $S=2$ would exclude one term for every term that is retained, resulting in only $\frac{1}{S}=\frac{1}{2}$ of query terms being evaluated.

The form of query striding that has just been described is the special case where $S=S_{max}$, for each and every stride, where S_{max} is the maximum stride length. Query striding can be generalised by allowing S to take any value in that satisfies $1 \leq S \leq S_{max}$. If this generalisation applies to each and every stride, then it is possible to optimise the stride selection in order to minimise the total term frequencies, and hence the number of postings that must be processed, and consequentially the work required.

This minimisation is possible because terms differ in frequency. This is a characteristic that is enhanced by partitioning the database (thus preventing excessive averaging of frequencies over an entire database). Thus, it is possible to select the terms that result in the lowest total frequency, while still satisfying the constraint on the maximum distance between pairs of successive selected terms.

Consider the example of Figure 4.5, consisting of seven consecutive overlapping query terms. The query terms are `ABCD`, `BCDE`, ..., `GHIJ`, and their frequencies are as listed, e.g., $F(\texttt{ABCD})$=24.

If $S_{max}=4$, the naive algorithm ($S=S_{max}$) would satisfy the constraint that at least every 4^{th} term be considered, by selecting the 4^{th} term (`DEFG`) as the first and only stride, yielding a total frequency of 32. This corresponds to line (a) in the figure, where the chosen term is

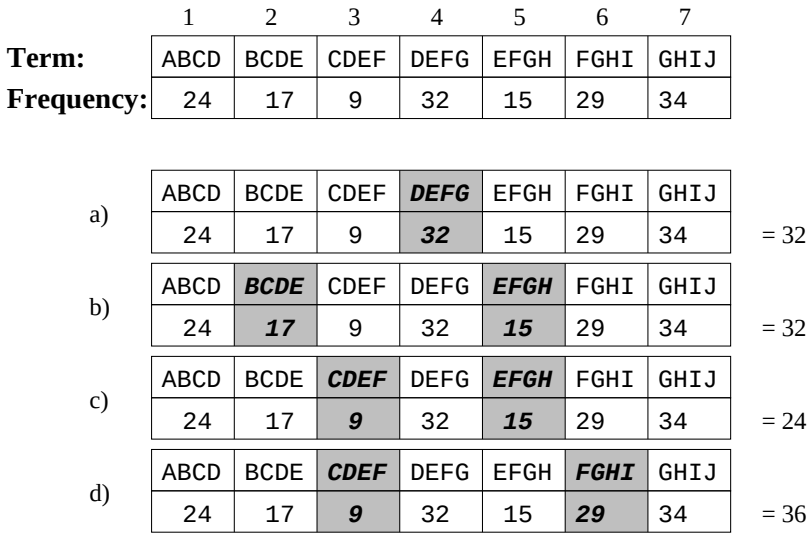

Figure 4.5: Example Of Optimising Striding, $S_{max} = 4$.

marked with a grey background. Alternatively, more than the minimum number of terms can be selected, e.g., lines (b), (c) and (d) in the figure, which represent three such selections. In the example, line (c) results in a total frequency of 9 + 15 = 24. This is lower than any other possibility, including option (a) where only a single term was selected.

The minimisation of the total term frequencies is a classic finite optimisation problem, and can be solved using dynamic programming in $O(length_{query} \times S_{max})$ time. Since the both the query length and S_{max} are generally small, the cost of this process is negligible, so any savings it can introduce are essentially free. Therefore this scheme, Dynamic Programming Optimised Query Striding, is implemented in DASH.

4.1.1.6 Combined Effect Of Alignment Candidate Reduction Measures

Combining the methods described in this section, the number of index postings, and hence alignment candidates, that are actually evaluated can be substantially reduced. Table 4.4 shows a typical breakdown of the percentages of postings considered, and those rejected by striding, frequency based exclusion (i.e., stop words), and suppressed by previously

Table 4.4: Effect Of Index Posting Evaluation Reduction Strategies.
Results are the aggregates of two hundred protein queries of: candidates squelched by a previous alignment (suppressed); skipped by query striding (avoided); stopped due to excessive frequency (excluded); and finally, those actually considered (remaining).

	Index Postings			
max. stride	Suppressed	Avoided	Excluded	Remaining
1	0.57 %	0 %	5.47 %	93.97 %
2	0.21 %	52.28 %	1.73 %	45.78 %
5	0.07 %	84.53 %	0.09 %	15.31 %
10	0.03 %	94.15 %	0.01 %	5.81 %

discovered alignments. (These data were extracted from the queries that were executed as part of this dissertation).

Only limited improvement is possible when $S_{max} = 1$, since this corresponds to evaluating every term in the query. In that case only stop words and suppression of existing alignments can be enforced, but query striding is of no use. However, for maximum stride lengths of 2, 5 and 10, a naive striding algorithm should, respectively, be able to avoid an average of $1 - \frac{1}{2} = 50\%$, $1 - \frac{1}{5} = 80\%$ and $1 - \frac{1}{10} = 90\%$, of all index postings. The value of the optimised query striding algorithm is demonstrated by its ability to consistently reject more than the expected fraction of postings (52.28% versus 50%, 84.53% versus 80% and 94.15% versus 90%). For the tabulated range of maximum stride lengths, the avoidance of highly frequent terms by the optimal striding algorithm causes the rapid decline in the number postings that need to be excluded as stop words. This also gives further weight to the reasonableness of excluding stop words in conjunction with the DASH algorithm, since the probability of a stop word being consulted is substantially reduced.

In apparent contrast to the potency of query striding, Table 4.4 reveals that relatively few alignment candidates are suppressed by previously discovered alignments. At first glance, this suggests that this suppression may not be worth the added complexity and effort it requires. This would be true, if the suppressed candidates were randomly selected, and so were very short on average (alignment lengths follow an extreme distribution

(Erdos and Renyi 1970)). However, this suppression works precisely on those candidates that produce alignments of the greatest length. *All* needless repetition of alignment discovery effort is therefore avoided, independent of the raw quantity or percentage of alignment candidates suppressed, thus ensuring that HSP discovery is O(n) and not O(n^2) with respect to query length, even when many high quality alignments occur. This remains true, even as the maximum stride length increases, thus reducing the overall number of alignment candidates requiring suppression.

To summarise, by combining the effects of stop words, optimal query striding, and candidate suppression the total number of alignment candidates that DASH must process, and hence the corresponding work required, can be reduced by an order of magnitude or more. This is in return for the modest reduction in sensitivity expected due to query striding, as detailed in a later chapter. Perhaps more importantly, these measures work to ensure that the HSP discovery process is never worse than O(n) with respect to query length or database size.

4.1.2 Stage 2: Optimal Assembly Of HSPs

As previously described, the first stage of the DASH algorithm discovers ungapped alignments between the query and the records in the database. This means that there will be zero or more ungapped alignments against each record in the database. Those records with no alignments are simply ignored, while those with a single alignment are immediately passed to the final stage to be extended using gapped alignment. However, additional work is required when more than one alignment is found against a given record.

In general, there are two possibilities when multiple alignments exist. First, the alignments may be independent, and should be treated as though each existed in isolation. On the other hand, various combinations of the alignments may represent fragments of larger overall

alignments. In that case, the fragments must be correctly assembled in order to realise the overall alignments.

Several goals are possible during the assembly process. One goal is to maximise the total score of all reported alignments, i.e., $\sum s_i$, $1 \leq i \leq n$, where s_i is the score of the i^{th} alignment, and n is the number of alignments. However, that goal need not prefer the inclusion of all regions between the identified alignments, causing a blind spot toward recurrences occurring in those regions when used in conjunction with cooperatively compressed database and index ensembles. Moreover, a single alignment with score, e.g., $\frac{3}{2}s$, has higher statistical significance than two alignments, each with score s, when multiple alignment statistics are used[1]. Therefore, it is considered preferable to join two alignments with scores $s_1$and s_2 to produce a joined alignment with score s_j, even if $s_j < s_1 + s_2$, provided that $s_j > s_1$, $s_j > s_2$. DASH performs such a process, with the aim of producing the final alignments with the highest overall scores.

Consider the example of Figure 4.6, where seven HSPs (denoted by heavy black lines) have been identified and scored. In addition, the regions between nearby HSPs have been explored and global dynamic programming used to determine the maximum score for the region between each plausible pairing of HSPs. These regions are shaded in grey to indicate that they are subjected to dynamic programming. The assembly options are drawn as double ended arrows, and are marked with their scores.

All assembly options for a given HSP are processed in a single dynamic programming episode, by exploring the minimal rectangle that contains all relevant HSP termini. This allows the reuse of dynamic programming effort when multiple possibilities exist. However, there are situations where this approach can be detrimental, because using the single area may be of much larger area than the sum of the individual areas that would be required

[1]While it is possible to combine the significance of separate alignments (Altschul and Gish 1996a, Collins et al. 1988, Mott 1992, Waterman and Vingron 1994), it makes sense to assemble alignments as fully as possible. The opposite extreme, reporting each alignment as a collection of alignments, each one base long, is not only absurd, but also hides the real significance of the complete alignment. Therefore complete assembly is preferable.

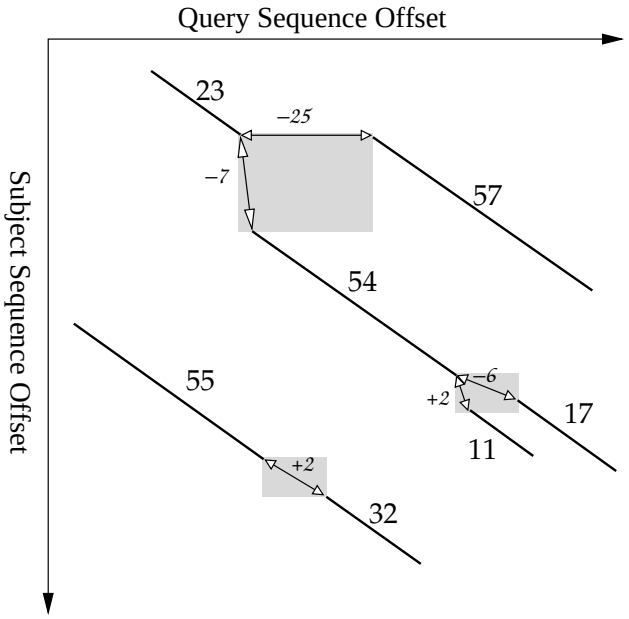

Figure 4.6: Hypothetical Complex HSP Assembly Situation.

if each HSP were considered separately. The large dynamic programming region near the top of the figure illustrates such an instance. Although the existing implementation of the DASH algorithm does not do so, it would be possible to test when this situation arises, and to cluster the candidates into sub-groups that require less over all dynamic programming.

Once the set of HSP and inter-HSP scores are known, dynamic programming is used to determine the maximum score that can be obtained from creating assemblies of the HSPs. Figure 4.7 depicts the result of performing this step on the current example. Only those transitions that produce a higher combined score are retained, hence the absence of a join between the two alignments at the top of the figure. The score for each chain of HSPs is implicitly calculated during this process. For example, the alignment scoring 81 receives that score because of its own score of 17, plus the inter-HSP score of -6, plus the chained score of 70 for the HSP it connects to. That score of 70 is in turn calculated from the sum of $54-7+23$. Each unique high scoring chain of HSPs is recorded as an alignment. In the example, this results in alignments with scores of 57, 81, 83 and 89, some of which re-use certain HSPs.

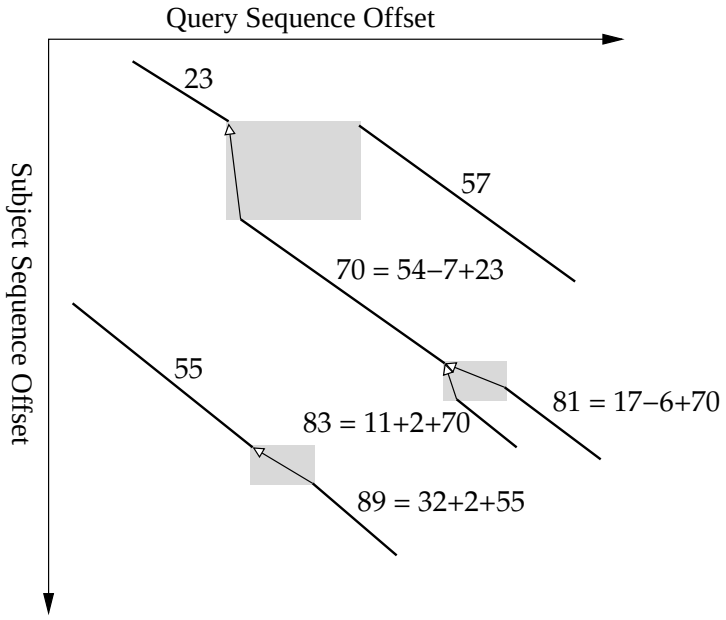

Figure 4.7: Hypothetical Complex HSP Assembly Situation.

By formulating the HSP assembly problem as a dynamic programming exercise, and by ordering the possible graph of HSPs, this problem is reduced to O(n^2), where n is the number of HSPs discovered against a single sequence in the database (rather then against the database as a whole). Empirically, it has been found that generally there are very few HSPs that can be plausibly assembled, resulting in a behaviour that is closer to O(n) than $O(n^2)$ on average.

4.1.3 Stage 3: Alignment Finishing Using Adaptive Banded Dynamic Programming

At the completion of the first two stages, the DASH algorithm has a list of gapped alignments and their scores. However, the regions beyond each end of the alignment have not been examined to determine if the assembled alignments can be further extended. This is the subject of the third and final stage of the DASH algorithm, where a dynamic programming procedure is employed to discover whether further extension is possible. In order

to reduce execution time, a heuristic approach is used in place of a completely exhaustive algorithm.

FASTA and NCBI-BLAST 2 implement successful heuristic algorithms that are relevant to extending alignments in this way: FASTA limits the extension to a band around the alignment region, while BLAST ceases extension if the maximum score on the dynamic programming front drops more than some quantity below the best score seen in the extension. DASH combines these two approaches, and adds: (a) a recessed starting point; (b) a slow start heuristic, and; (c) adaptive placement of the band.

The recessed starting point trims a short length from the end of an HSP in order to allow the discovery of a more optimal alignment in the case that the (ungapped) HSP has been over extended, and deviates from the optimal gapped alignment.

The slow start heuristic is used to make the band initially narrow, when it is much less likely that the alignment will drift far from the main diagonal of the alignment. On average, this results in approximately 35% less dynamic programming effort, and with practically no reduction in sensitivity. Then throughout extension, the band is repeatedly recentred around the highest scoring point on the dynamic programming front.

Adaptive band placement is best illustrated with an example. Figure 4.8 depicts the operation of the adaptive band placement. Initially the upper most row is explored, and its highest scoring cell (indicated by the upper most dot) is identified. As this dot is near the mid-point of the row, i.e., not too near either the left or right edge of the band, the next band is placed exactly one cell to the right the current band. The dynamic programming process is then executed, yielding the highest scoring cell in this second row, as indicated by the second-upper-most dot. Again, it occurs near the mid-point, so the third band is placed exactly one cell right of the current band. Repeating this process for the third row, the highest scoring cell is now near the left edge of the row. This causes the dynamic band placement algorithm to place the fourth row at the same horizontal position as the third.

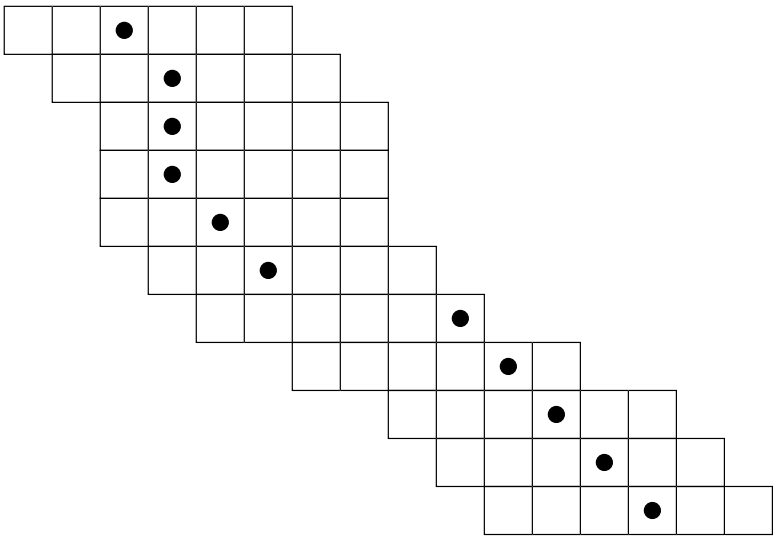

Figure 4.8: Simple Example Of Adaptive Band Placement During Dynamic Programming. Each column represents a band. The solid black dots represent highest scoring cell in each band.

This process repeats for the fifth row, before the highest scoring cell again occurs near the mid-point. However, when the sixth row is evaluated the highest scoring cell is found to occur near the right edge. This causes the dynamic band placement algorithm to place the seventh row two cells to the right. This occurs again for the eighth row, after which the adaptive band placement has recentred the band around the highest scoring cell.

Figure 4.9 presents the dynamic programming activity for an alignment of a protein query that was executed using DASH. This dynamic programming activity results in the alignment shown in Figure 4.10. The example demonstrates all aspects of the adaptive banded dynamic programming algorithm, including the slow start heuristic. As the example is extracted from a real query, there are two artifacts that should be noted: (a) the alignment occurs in the last 24 residues of the database record. This explains why the dynamic programming figure appears truncated on the right hand-side. (b) The full evaluation of the first row is an optimisation that improves dynamic programming efficiency, by avoiding the need for a bounds check in the computation of each cell, therefore that row can be ignored throughout the following discussion.

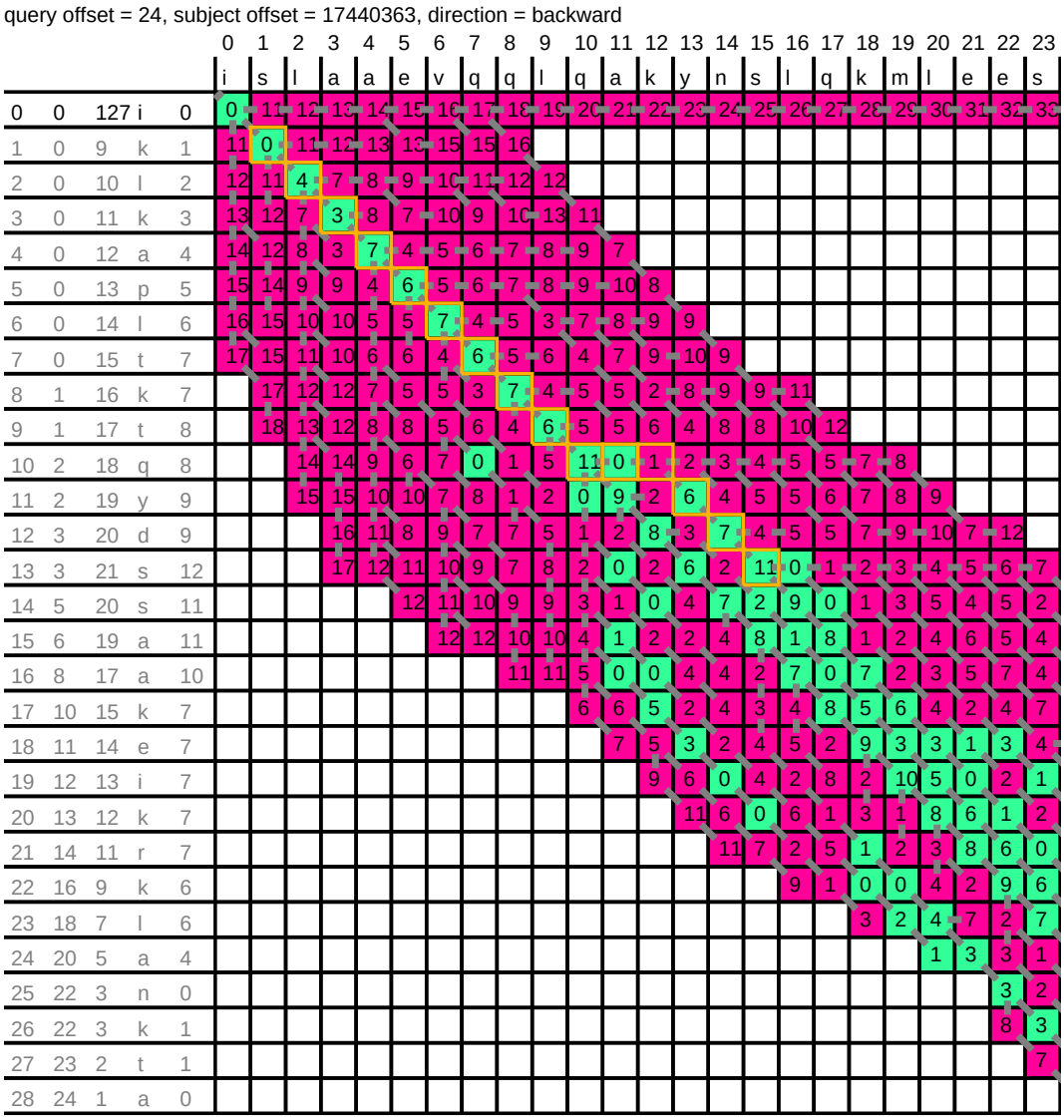

Figure 4.9: Example Of DASH Adaptive Banded Dynamic Programming.
Scored using the BLOSUM62 matrix, with penalties of -11 to open and -1 to extend a gap. The fields above each column are, from top to bottom: (a) the number of the column, and; (b) the amino acid in that position of the query sequence.
The fields to the left of each row are, from left to right: (a) the row number; (b) the first column of the dynamic programming band for that row; (c) the width of the dynamic programming band for that row; (d) the amino acid in that position of the database record; and, (e) the relative position of the highest scoring cell in the dynamic programming band for that row.
Note the explored band preferentially widening on the side nearest the maximum score. Purple and green coloured cells indicate negative and positive dynamic programming scores, respectively. Orange boxes indicate the optimal alignment path. Small grey ticks between cells indicate the back trace path from each cell toward the origin. The extension terminates at the furthest point with the maximum score seen (+11).

```
ISLAAEVQQLQAKYNS
| | | + + |   |+|
IKLKAPLTKTQ--YDS
```

Figure 4.10: Alignment Resulting From Figure 4.9 (final score = +11).

For the remaining rows of the figure, until the bands begin to be truncated by the limits of the dynamic programming space (rows 1 – 14), the width of the successive dynamic programming bands increases, i.e., the rows start narrow, and then progressively widen, owing to the slow start heuristic. The widening preferentially occurs on the side nearest the maximum value in each row in order to adapt to where the best alignment appears to be. The band is also moved in absolute terms to help maintain this centre, and hence the horizontal progression of each row.

As previously described, the progression of the band for each successive row is monotonic, thus maintaining the centre around the same diagonal. The two exceptions to this are if the highest scoring cell in a given row occurs in the left or right quarter of the band. In those cases, the row either maintains position, effectively shifting the centre line one cell to the left (as in after rows 8, 10, 12 and 25), or shifts two positions to the right, effectively shifting the centre line one cell to the right (as in after rows 13, 15, 16, 21, 22, 23 and 24). In cases where there are two or more cells that each have the maximum score for a given row, the algorithm naively chooses the right most one. More sophisticated regimes are possible, such as trying to keep all maximally scoring cells in the band, but have not been explored.

Overall the dynamic programming effort has an upper bound of O(n^2) with respect to query length. However, in practice dynamic programming is only performed for relatively few alignments, and the slow start heuristic combined with the termination heuristic ensure that in most cases only a small constant amount of effort is expended, again resulting in an typical aggregate cost that is much less than O(n^2).

4.1.4 On Search Time Complexity

It has previously been observed in this dissertation that predictable execution time and sub-linear execution time are desirable characteristics for a sequence search and alignment algorithm to possess. In pursuit of this goal, each stage of the DASH algorithm has been designed to be no worse than linear in practice. Predictable execution time is further encouraged in DASH in much the same way that it is in CAFE. That is by ensuring that the fine searching (the HSP assembly and alignment finishing stages in DASH) occur only once per hit. Meanwhile, sub-linear execution time is encouraged by allowing the use of cooperatively compressed databases, an issue that is addressed in later chapters.

Another issue that requires addressing is that traversing recurrence chains of unknown length may create an unpredictable contribution to search time. This is a significant issue, particularly if the data are to be disk-resident, as each step in the chain will trigger a costly disk seek.

This issue has been mitigated to some degree by partitioning the database and requiring that all links in a chain point backwards in the partition, thus ensuring that chains are limited in length. It is further addressed by the properties of the search system that make it possible to stream the compressed database and index from disk to RAM in a batched search environment. These properties are discussed in detail in Chapter 7 where DASH is combined with an appropriate data and index representation. Given the compactness of the compressed structures that result this solves the problem, in principle at least.

4.2 Search Parameters

4.2.1 Tunable Parameters

The current implementation of the DASH algorithm contains a number of tunable parameters, described below grouped by their function. Empirical exploration of these parameters has led to the production of canonical parameter sets, referred to as *modes* in the rest of this dissertation. These are more fully defined following the general definition of each individual parameter.

4.2.1.1 Tunable Alignment Properties

Maximum Expected Value of Alignments (MaxE). This specifies the expected value (by random chance) that is used as the threshold for determining the length and score of alignments to report. Typical values are in the range 0.1 to 1000.

Does The Maximum Expected Value of Alignments Constitute a Limit or a Quota (MaxE-Limit). If this boolean is true, then the maximum expected value of alignments constitutes a limit. In practical terms, setting this flag reduces the *minimum alignment length* by one, and so slightly increases sensitivity at the cost of speed.

Minimum Alignment Length (MinAlnLen). This specifies the minimum final length an alignment must reach in order to be reported, and is not a manually tunable parameter. It is overwritten at run time by the value derived from the *Maximum Expected Value of Alignments* and the size of the database being searched. It forms part of the parameter vector in order to keep all parameters located in one place. This reduces the number of function arguments that must be passed to various functions, and boosts overall speed with some compilers.

4.2.1.2 Tunable Query Striding Parameters

The Maximum Stride Length Allowed Between Investigated k-mers of The Query Sequence (MaxStride). The maximum stride length (S_{max}) is calculated as the minimum of this parameter and the difference between the minimum HSP length and k. This parameter then provides a mechanism for capping the maximum stride length, regardless of the minimum HSP length. Typical values are between 1 (no query striding) and 50 (practically equivalent to using *minimum HSP length* minus k).

The Number of Neighbours to Consider for Each k-mer of the Query Sequence (Neighbours). This specifies the number of neighbouring k-mers, in the style of BLAST 2, to consult, at each stride through the stride path. A value of one indicates that only the exact k-mer from the query sequence should be used. All k-mers activated at each position of the query sequence are taken into account when calculating the optimal stride path. Typically this value is set to 1 to indicate that BLAST style neighbouring k-mer support is disabled.

4.2.1.3 Tunable HSP Properties

Minimum HSP Length (MinHSPLen). This prescribes the minimum length an HSP must reach in order to be recorded and be a candidate for alignment assembly. Smaller values increase sensitivity, but not as quickly as they increase memory and computational burden. Typical values are between 10 and 14 residues.

Maximum Continuous Incongruence Length Tolerated in HSPs (MaxSubst). This parameter restricts the number of consecutive non-conservative substitutions that will be tolerated in an HSP. Moderate values, e.g., one or two, result in fewer and longer HSPs compared to not tolerating any substitutions. Typically this value is set to one, which means that two consecutive substitutions will terminate HSP extension.

4.2.1.4 Tunable DP And HSP Assembly Parameters

Perform DP at Ends (DPEnds). This is a boolean flag that indicates whether dynamic programming will be attempted during the third stage of the search process. If it is false, then gapped alignments will still be produced, however the full extent of many alignments is unlikely to be determined, especially for protein searches. However, it does allow for very rapid searching. Normally, however, this flag is set to true.

Measure Before or After Performing DP at Ends (MeasureFirst). If this is true, then alignments will be measured against *Minimum Alignment Length* before performing dynamic programming at each end. This contains the search time by reducing the number of dynamic programming extensions performed. The alternative is measuring after, and results in greater sensitivity at the expense of the computational cost due to dramatically increased dynamic programming activity. Typically this value is set to true to cause measurement to occur before extension, preferring execution speed over sensitivity.

HSP Joining Cut Off Score (JoinAbort). This specifies the minimum score a global dynamic programming alignment between two HSPs can achieve before abandoning the attempt to join the HSPs. Large negative values result in higher sensitivity, but at the cost of increased processing time. Typical values are in the range -10 to -20.

The Maximum Distance Between HSPs Before Joining Them Will Not Be Considered (JoinRange). This distance places a limit on the distance between HSPs when joining is attempted, therefore placing an upper bound on the work required in the second search phase when HSPs are assembled. Typical values are in the range 50 to 100 letters.

The Width of the Dynamic Programming Band Used for The Gapped Extension of Alignments (DPWidth). This specifies the maximum width of the banded dynamic programming front during extension at each end of a gapped alignment. The band dynamically centres itself around the optimal path discovered to date, reducing the band width required for adequate sensitivity. Therefore values between 20 and 64 are normally sufficient.

The Maximum Length of Dynamic Programming Used for The Gapped Extension of Alignments (DPMax). This sets the internal maximum length for a single episode of dynamic programming. Multiple episodes of dynamic programming will be used if longer dynamic programming is warranted. This parameter has no effect on sensitivity, it only effects the memory efficiency of the banded dynamic programming process. This value is normally set to 512, so that all practically every dynamic programming request is handled in a single episode.

The Length of Recession at the Ends of an Alignment Used During The Dynamic Programming of Alignments (RLen). This parameter is used primarily for protein searches to allow recovery from over extended HSPs at the end of alignments. By stepping back several residues into the alignment and starting the dynamic programming process from there, any incorrect over extension can be replaced with the optimal path. Typical recession lengths are between 5 and 10 letters.

The Cut-Off Score Used During the Gapped Extension of Alignments (X_g). This parameter performs the same function as X_g in NCBI-BLAST 2. In both cases it provides the termination criteria for gapped extension of alignments. In DASH it limits only the length of dynamic programming, as the dynamic programming band remains constant width (discounting the influence of the slow start heuristic). X_g is typically set to terminate extension when the alignment score drops by between -10 and -25 compared to the previously observed maximum.

Dynamic Programming Slow Start Bandwidth Divisor (SSDivisor). This value specifies the ratio of the maximum to initial bandwidth. Larger values cause dynamic programming to commence with a narrower band, thus reducing the computation burden. Typically this value is set to 8, where it makes almost no impact on sensitivity, but reduces dynamic programming effort by approximately 35%.

Table 4.5: DASH Canonical Parameter Sets For Nucleic Acid Searching: M2.

Name	M2 (Nucleic Acid)		
Parameter	*Value*	*Parameter*	*Value*
MaxE	1	MeasureFirst	Yes
MaxELimit	Yes	JoinRange	100
MinAlnLen	18	JoinAbort	-10
MaxStride	50	DPWidth	40
Neighbours	1	DPMax	512
MinHSPLen	18	RLen	5
MaxSubst	1	X_g	20
DPEnds	Yes	SSDivisor	8

Table 4.6: DASH Canonical Parameter Sets For Nucleic Acid Searching: M4.

Name	M4 (Nucleic Acid)		
Parameter	*Value*	*Parameter*	*Value*
MaxE	10	MeasureFirst	Yes
MaxELimit	No	JoinRange	100
MinAlnLen	10	JoinAbort	-20
MaxStride	50	DPWidth	60
Neighbours	1	DPMax	512
MinHSPLen	10	RLen	5
MaxSubst	1	X_g	25
DPEnds	Yes	SSDivisor	8

4.2.1.5 Canonical Parameter Sets

The canonical parameter sets are defined for nucleotide and protein searching respectively, and presented in Tables 4.5, 4.6, 4.7 and 4.8. The M2 modes are intended to run an order of magnitude faster than NCBI-BLAST 2.2.6, while the M4 modes are intended to run in comparable search time as NCBI-BLAST 2.2.6, with the best sensitivity available. It is likely that these are not the optimal parameter sets in pursuit of their respective goals. However, for the purposes of this dissertation all that is required is a fixed reference point that can be used to compare the effect of different database and index representations. Therefore further optimisation of these parameter sets is left as a future exercise.

Table 4.7: DASH Canonical Parameter Sets For Protein Searching: M2.

Name	M2 (Protein)		
Parameter	*Value*	*Parameter*	*Value*
MaxE	1	MeasureFirst	Yes
MaxELimit	Yes	JoinRange	100
MinAlnLen	14	JoinAbort	-10
MaxStride	5	DPWidth	60
Neighbours	1	DPMax	512
MinHSPLen	14	RLen	5
MaxSubst	1	X_g	20
DPEnds	Yes	SSDivisor	8

Table 4.8: DASH Canonical Parameter Sets For Protein Searching: M4.

Name	M4 (Protein)		
Parameter	*Value*	*Parameter*	*Value*
MaxE	1	MeasureFirst	Yes
MaxELimit	Yes	JoinRange	100
MinAlnLen	14	JoinAbort	-10
MaxStride	1	DPWidth	60
Neighbours	1	DPMax	512
MinHSPLen	14	RLen	5
MaxSubst	1	X_g	25
DPEnds	Yes	SSDivisor	8

```
-- Parse command line arguments
parse_command_line_arguments()
-- Retrieve the array of parameters based on the
-- setting selected
active_settings = settings_array[selected_setting]
-- Perform the search
foreach index_partition in the indexed database
  -- 1. Search each index section
  discover_hsps(index_partition,query)
  -- 2. Assemble the HSPs into alignments
  assemble_HSPs()
  -- 3. Perform DP at each end of the assembled alignments
  finish_alignments()
end foreach
-- Output search results
output_results_using_selected_format()
```

Figure 4.11: Pseudo Code For The DASH Search Algorithm: Overview.
Command line parameters are parsed to determine the database and query to search, and to select the search parameters. Then the three stage alignment process is conducted for each database partition. Finally, the search results are output in the selected format.

4.3 DASH Search Program (dash)

The DASH algorithm was implemented in C to produce the `dash` program. This program parses command line arguments to select protein or nucleotide searching, the database and query to search, and the search mode. It implements the three stages of the DASH algorithm as previously described. A pseudo code overview of this is included in Figure 4.11. Pseudo code for each of the three steps is presented in Figures 4.12, 4.13 and 4.14, respectively.

4.3.1 Scoring, Statistics And Output Format

DASH scores alignments using either match and mismatch scores (nucleic acid) or a substitution matrix (protein). To this are added penalties for opening and extending gaps. In addition, the scores of wild cards are written down in order to moderate the score of alignments that contain them. Apart from this last addition, this scoring system is essentially

```
-- Find the optimal stride path
stride_path = optimal_stride_path(query,index,max_stride)
-- For each stride in the stride path ...
foreach q in stride_path
  -- get the k bases at position q in the query sequence
  qt = query_segment(q,k)
  -- get and decompress postings list for this term
  postings_list = index.nmer_addresses[qt]
  uncompress(postings_list)
  -- Iterate through postings for this stride
  foreach m in postings_list
    -- Try to non-gapped extend the tuple match at
    -- position m:q
    if try_ungapped_extension(m,q) == GOOD then
      -- Work out which database record the HSP occurs in
      r = which_dbrecord(m)
      -- Write discovered HSP to the HSP tree for that record
      write HSP(m,q) to HSP_tree[r]
    end if
  end foreach m
end foreach q
```

Figure 4.12: Pseudo Code For The DASH Search Algorithm: HSP discovery.
The optimal stride path is determined using the maximum stride length and k-mer frequency distributions obtained from the index of the current database partition. The strides in the path are then iterated to obtain the address of each index posting. These alignment candidates are then tested. Those resulting in valid HSPs are recorded in the HSP tree for the database record they occur in.

```
-- Foreach record in the database partition
for s = 1 to index_partition.record_count
  -- consider the HSPs found in that record
  foreach hsp in HSP_tree[s]
    window = array of nearby diags to the ‘‘left’’ of diags
    bestneighbour = element of window that combined with HSP
                    has the best dynamic programming score
    if bestneighbour is not NULL then
      -- record that HSP joins to bestneighbour on the left
      hsp.join_left = bestneighbour
      -- Record the dynamic programming path that joins them
      bestneighbour.backtrace = dp_path(HSP,bestneighbour)
   end if
  end foreach
  -- Join the diagonals to form gapped chains of HSPs
  foreach HSP in HSP_tree[s], in descending query offset order
    -- Check if this HSP has already been processed
    if ( HSP.processed != true ) then
      -- Render this HSP into an alignment
      alignment = (empty alignment)
      alignment.append(HSP)
      -- Find the HSP joined to the left of this one
      left_HSP = HSP.join_left
      while left_HSP is not NULL
        -- Render the DP backtrace between the two HSPs
        alignment.append(left_HSP.backtrace)
        -- Render the left HSP
        alignment.append(left_HSP)
        left_HSP.processed = true
        -- See if left_HSP joins to another HSP on its left
        left_HSP = left_HSP.join_left
      wend
      -- calculate score, and record alignment
      alignment.calculate_score()
      list.append(gapped_alignments,alignment)
   end if
 end foreach
next s
```

Figure 4.13: Pseudo Code For The DASH Search Algorithm: HSP Assembly.
HSPs are assembled by first identifying the candidates for joining to each HSP. These candidates are screened to identify the best HSP, if any, that each can join to. The list is traversed a second time to output the gapped alignments, marking the HSPs that are output along the way. Since HSPs marked in this way already form part of a longer and higher scoring alignment that has already been output, they are ignored during the list traversal.

```
foreach a in gapped_alignments
  if ( measure_before_performing_DP ) then
    -- Measure alignment before DP extension.
    -- Discard it if it has too low a score or is
    -- too short.
    if ( a.length < minimum_alignment_length ) then
      -- discard and try next a
      goto next_a
    end if
    if ( a.score < minimum_alignment_score ) then
      goto next_a
    end if
  end if
  -- Perform DP at each end
  do_banded_dp(a,left)
  do_banded_dp(a,right)
  -- Recalculate alignment score and length
  recalculate_score(a)
  if NOT ( measure_before_performing_DP ) then
    -- Measure alignment after DP extension.
    -- Discard it if it has too low a score or is
    -- too short.
    if ( a.length < minimum_alignment_length ) then
      goto next_a
    end if
    if ( a.score < minimum_alignment_score ) then
      goto next_a
    end if
  end if

   -- Record final alignment
   list.insert(search_results,a)
next_a:
end foreach
```

Figure 4.14: Pseudo Code For The DASH Search Algorithm: Dynamic Programming Ends Of Alignments.
If a gapped alignment meets minimum score and length requirements it receives a comprehensive banded dynamic programming treatment at each end. However, if the appropriate settings are selected, then the dynamic programming extension is always performed, and the length and score assessment is performed after. Either way, the dynamic programming treatment begins recessed within the ends of the alignment to assist in the detection of gaps around those regions. Any alignment that passes the length and score criteria is recorded in the list of results.

```
DASHN 1.0.9 [Nov-2004]

                                                                    Score    E
Sequences producing significant alignments                           (dp)   val.
...
gnl|UG|Hs#S1090875 Homo sapiens mRNA for JM23 protein, complete ...  330 4.75076e-188
gnl|UG|Hs#S2138821 Homo sapiens FtsJ homolog 1 (E. coli) (FTSJ1)...  330 4.75076e-188
gnl|UG|Hs#S3800695 602866556F1 Homo sapiens cDNA, 5' end /clone=...  330 4.75076e-188
gnl|UG|Hs#S2155111 601116752F1 Homo sapiens cDNA, 5' end /clone=...  330 4.75076e-188
gnl|UG|Hs#S3340458 602426736F1 Homo sapiens cDNA, 5' end /clone=...  329 1.93307e-187
gnl|UG|Hs#S4277359 AGENCOURT_6632171 Homo sapiens cDNA, 5' end /...  327 3.19776e-186
gnl|UG|Hs#S3842279 603081789F1 Homo sapiens cDNA, 5' end /clone=...  326 1.30007e-185
gnl|UG|Hs#S3615612 602764031F1 Homo sapiens cDNA, 5' end /clone=...  326 1.30007e-185
...
>gnl|UG|Hs#S2155111 601116752F1 Homo sapiens cDNA, 5' end
           /clone=IMAGE:3357348 /clone_end=5' /gb=BE257786
           /gi=9128265 /ug=Hs.23170 /len=632
 Score = 330
 Identities = 375/382 (98%)
 Strand = Plus / Plus

Query:  1 ccggcccg-ccgaacctgggcgatccacgatgccgagtttgccacgctgcgacagccaat 59
          |||||||| | ||||||||||||||||||||||||||||||||||||||||||||| ||
Sbjct:  3 ccggcccgccggaacctgggcgatccacgatgccgagtttgccacgctgcgacagcccat 62

Query: 60 aggcttgc--ccccccggcattcgggtggactacgaacacaaactgaagccctaggactt 117
          ||||||||  ||||| ||||||||||||||||||||||||||||||||||||||||||
Sbjct: 63 aggcttgccccccccgggcattcgggtggactacgaacacaaactgaagccctaggactt 122

Query: 118 gtcgcccgtttgcgctctcgccgaggcacaggctgctcgcggaccacccntgcntccga 176
           |||||||||||||||||||||||||||||||||||||||||||||||| ||| ||||
Sbjct: 123 gtcgcccgtttgcgctctcgccgaggcacaggctgctcgcggaccaccc-tgc-tccg- 178
...
```

Figure 4.15: Example DASH Output.

identical to that of BLAST, FASTA and many other sequence alignment programs. Expected value statistics are calculated using the method of Altschul (1993), and the format of the output is essentially the same as that of NCBI-BLAST 2.2.6, with an example excerpt displayed in Figure 4.15.

4.4 Results

The DASH algorithm as described at this point does not include an index format, and since the nature of the index format is a critical component of a complete sequence search and alignment system, influencing both speed and sensitivity, it is not appropriate to present general search results here. That is the subject of later chapters. However, it is possible to examine some behaviours of the DASH algorithm that are independent of the index format.

4.4.1 Illustrated Example Of Alignment Assembly

The following example depicts a real assembly as emitted by the internal instrumentation in the `dash` program during testing. Thick blue lines correspond to HSPs, while red and purple regions correspond to those areas subjected to dynamic programming. The thin black lines with bars at each end correspond to reported alignments. The textual annotations to the right of the diagram indicate the end points and score of each reported alignment.

Figure 4.16 shows the case of the alignment of two very similar sequences. The primary alignment is the diagonal line that reaches from the bottom left to top right corner. Note that the dynamic programming begins within the end of each HSP. This is the practical effect of the recessed starting point. The smaller HSPs, symmetric about the main diagonal, reflect the existence of repeat regions within the sequence.

The extension of the alignments that occurs for each of the shorter HSPs demonstrates the slow start and adaptive features of the dynamic programming algorithm: The dynamic programming bands begin quite narrow (red) and progressively broaden until the maximum width is reached (purple), or truncated by the limits of the dynamic programming space, thus saving substantial effort. The adaptive placement of each successive band is particularly evident in the apparent wandering of the bands once they reach full width.

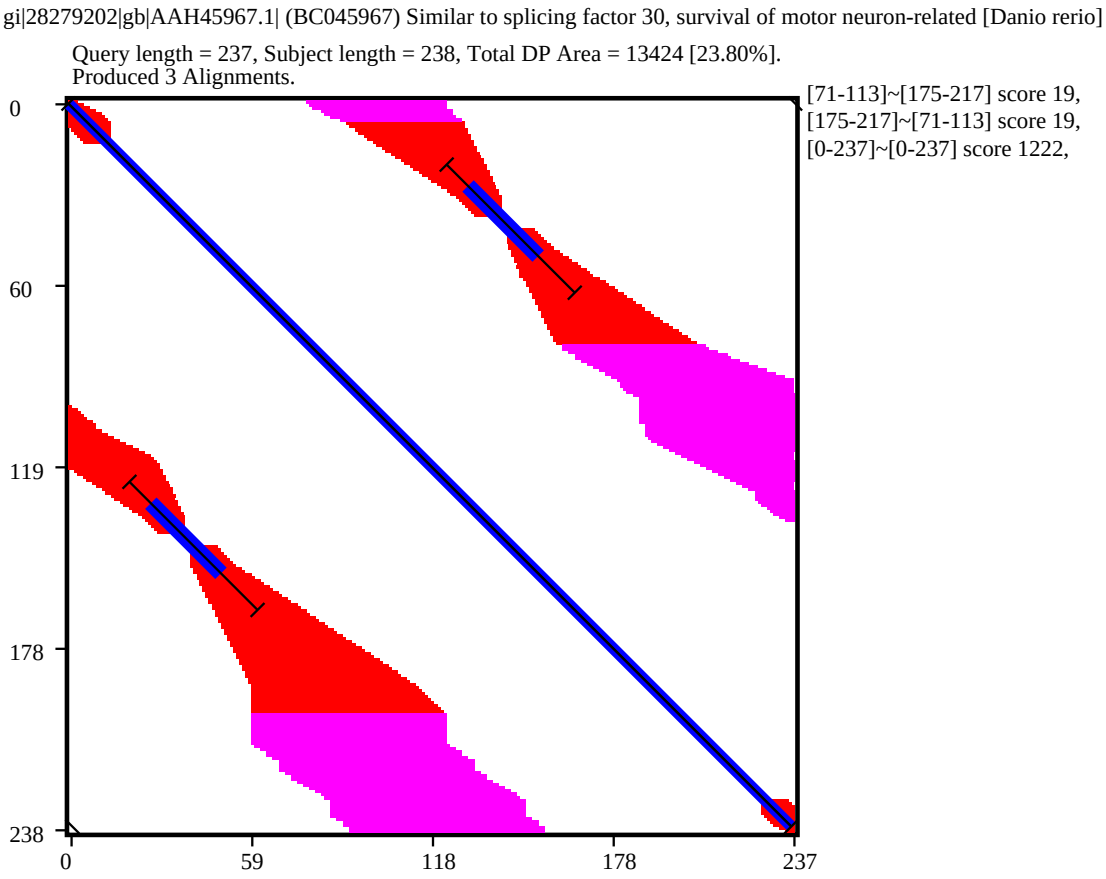

Figure 4.16: Example Of Alignment Between Two Very Similar Sequences. HSPs are marked blue. Dynamic programming regions are either red (slow start heuristic), or purple (after slow start heuristic) . The effect of the slow start banded dynamic programming heuristic is clearly visible in the widening bands (red).

```
>gnl|UG|Hs#S3290308 AL570239 Homo sapiens cDNA /clone=CS0DI004YD20
          /gb=AL570239 /gi=12926354 /ug=Hs.284170 /len=867
 Score = 277
 Identities = 465/524 (88%)
 Strand = Plus / Minus
Query:  1  cttttagctcactccacaagtaaatggatttaatcaaaggtcacctatctgcttt-ata 58
           |||||||||||||||||||||||||||||||||||||||||||||||||||||||| +
Sbjct: 660 cttttagctcactccacaagtaaatggatttaatcaaaggtcacctatctgcttttrat 600
Query: 59  tgtctaggtcacagtaatccctaggatttttcaccgcttattctttgttgtctttttga 117
           |||||||||||||||||||||||||||||||||||||||||||||||| |||||||||| |
Sbjct: 601 tgtctaggtcacagtaatccctaggatttttcaccgcttattcttt-ttgtctttttca 542
Query: 118 acaaacatattatccgaatttttttttctgcaagccactgatagtctctgctaactagct 176
            ||  | | || ||||||||||||||||||  ||||||| | |||||||||  || |||
Sbjct: 543 -catccttcttctccgaatttttttttctgct-gccactgttcgtctctgctccctcgct 485
Query: 177 taattgacctttttacaaagtttgatccccaagcatcctcaagctaaatcattgaatac 235
           | |||| ||||||| |||||||||||||||| || ||||||  || |||||||||||||
Sbjct: 486 tcattgccctttttccaaagtttgatccccatgcttcctca-tcttaatcattgaatac 427
Query: 236 ttcaatcaggatattatctggctttactttacaaataaaaaccaaatc-tttgtcaaca 293
           |||||||||||||||||| ||||||||| ||||||||  ||||||| ||||||||||
Sbjct: 428 ttcaatcaggatattatctg-ctttactttccaaataaac-ccaaatcttttgtcaaca 370
Query: 294 ggatgaaacccatcttaaaggaaagaaaaggaattggtgtgaagagagaagttagagaa 352
           || |||| |||||||||||||| | || || |||||||||||||+| |||| | | +
Sbjct: 371 ggttgaaccccatcttaaaggaacgcaacggtattggtgtgaagagwgcagtttgygtn 311
Query: 353 gggaaatgcaggtgaattactatctgtgtccatcaggaagtttgtcctgttaaccaaat 411
           |||+||||||| |||||||||||||||||| |||||||||||||||||||||||||||
Sbjct: 312 gggnaatgcag-tgaattactatctgtgtccttcaggaagtttgtcctgttaaccaaat 253
Query: 412 ggttactgcactacca-ggttactggtttattttccagggagctgataaagcaggagaa 469
           ||||||||||||+||| |||||||||||||||||||||| |||| | ||||||||| |
Sbjct: 254 ggttactgcactmccagggttactggtttattttccagggtgctgtttaagcaggagca 194
Query: 470 ctgttgctgcatgttttctatttggactccgtcacaatatggtaggatatccctc 524
           |||||||||||||||||| |||+| ||||+||||| | ||||||+ | ||||||

Sbjct: 195 ctgttgctgcatgttttctttttkg-ctccktcacattttggtagkttctccctc 142
```

Figure 4.17: DASH Alignment of S3317510 Versus S3290308.
Note that the alignment consists of two long high quality HSPs which DASH has correctly stitched together into a single high scoring alignment.

4.4.2 Example Of Superior Alignment Assembly

Figures 4.17 and 4.18 depict the alignment of the same pair of sequences as returned by DASH and BLAST respectively. Note that DASH constructs a single full alignment, while BLAST constructs only fragmentary alignments. This failing on the part of BLAST has the significant effect that the lower scores of the fragments cause the alignment to be placed 44 positions further down the list of results.

```
>gnl|UG|Hs#S3290308 AL570239 Homo sapiens cDNA /clone=CS0DI004YD20
           /gb=AL570239 /gi=12926354 /ug=Hs.284170 /len=867 Length = 867
 Score =  335 bits (169), Expect = 1e-89
 Identities = 322/367 (87%), Gaps = 7/367 (1%)
 Strand = Plus / Minus

Query: 150 gccactgatagtctctgctaactagcttaattgacctttttacaaagtttgatccccaag 209
           ||||||| | |||||||||  || |||| |||| ||||||| |||||||||||||||| |
Sbjct: 513 gccactgttcgtctctgctccctcgcttcattgccctttttccaaagtttgatccccatg 454

Query: 210 catcctcaagctaaatcattgaatacttcaatcaggatattatctggctttactttacaa 269
           | ||||||  || |||||||||||||||||||||||||||||||| ||||||||| |||
Sbjct: 453 cttcctcat-cttaatcattgaatacttcaatcaggatattatctg-ctttactttccaa 396

Query: 270 ataaaaaccaaatcttt-gtcaacaggatgaaacccatcttaaaggaaagaaaaggaatt 328
           |||||  |||||||||| ||||||||| |||| |||||||||||||| | || || |||
Sbjct: 395 ataaac-ccaaatcttttgtcaacaggttgaaccccatcttaaaggaacgcaacggtatt 337

Query: 329 ggtgtgaagagagaagttagagaagggaaatgcaggtgaattactatctgtgtccatcag 388
           |||||||||| | |||| | |  ||| ||||||| |||||||||||||||||| ||||
Sbjct: 336 ggtgtgaagagwgcagtttgygtngggnaatgcag-tgaattactatctgtgtccttcag 278

Query: 389 gaagtttgtcctgttaaccaaatggttactgcactacca-ggttactggtttattttcca 447
           |||||||||||||||||||||||||||||||||| ||| ||||||||||||||||||||
Sbjct: 277 gaagtttgtcctgttaaccaaatggttactgcactmccagggttactggtttattttcca 218

Query: 448 gggagctgataaagcaggagaactgttgctgcatgttttctatttggactccgtcacaat 507
           ||| |||| | ||||||||| |||||||||||||||||| ||| | |||| ||||| |
Sbjct: 217 gggtgctgtttaagcaggagcactgttgctgcatgttttctttttkg-ctcctcacatt 159

Query: 508 atggtag 514
           ||||||
Sbjct: 158 ttggtag 152


 Score =  182 bits (92), Expect = 1e-43
 Identities = 112/116 (96%), Gaps = 2/116 (1%)
 Strand = Plus / Minus

 Query: 1   cttttagctcactccacaagtaaatggatttaatcaaaggtcacctatctgctttata-t 59
           ||||||||||||||||||||||||||||||||||||||||||||||||||||||||  | |
Sbjct: 660 cttttagctcactccacaagtaaatggatttaatcaaaggtcacctatctgcttttratt 601

Query: 60  gtctaggtcacagtaatccctaggatttttcaccgcttattctttgttgtctttttt 115
           |||||||||||||||||||||||||||||||||||||||||||||||| ||||||||||

Sbjct: 600 gtctaggtcacagtaatccctaggatttttcaccgcttattcttt-ttgtctttttt 546
```

Figure 4.18: BLAST Alignment of S3317510 Versus S3290308.
Note that while the alignment consists of two high quality HSPs, BLAST fails to assemble them into a single high-scoring alignment. This failing causing such alignments to appear further down the list of results than their significance warrants, possibly causing them to be ignored.

4.5 Summary

In summary, the DASH algorithm has been defined as a three stage process of: (1) HSP discovery; (2) assembly, and; (3) finishing by adaptive banded dynamic programming. These procedures have been described with a view to explain how they support the objectives of this dissertation by constructing an algorithm that can make efficient use of a cooperatively compressed database and index ensemble, and offer the attractive characteristics of predictable execution time, and the potential for sub-linear execution time with respect to uncompressed database size. Therefore the current implementation is used as the vehicle for testing the thesis of this dissertation in Chapters 5 and 7.

Chapter 5

FOLDDB: First Steps In Cooperatively Compressed Databases And Indices

Introduction

In an English text, each indexed word is discrete, i.e., words in English text do not overlap one another. For example, to index the first two words of this sentence, “For” and “example” would be indexed, but “or e” and “r ex” would not. Therefore, to create an exhaustive index for all words in an English text requires only one pointer per word. The average word length is in the range of five to six letters (Witten et al. 1999). Consequentially, the number of pointers required in the index will be one-fifth to one-sixth the number of letters in the text.

An English word can be represented using perhaps five bits per letter, and so requires of the order of 30 bits per word. Using an appropriate coding scheme, the inverted list for a given word can be compressed to around 6 bits per posting, or around 12 – 18 bits per posting, if the position of a word within a record is required (Witten et al. 1999). In either case, the volume of the index (6 – 18 bits per word) will be somewhat smaller than the uncompressed text (30 bits per word).

However, in the case of biological sequence databases, words overlap because there are no natural word breaks: A word is instead a series of k consecutive letters — often called a

k-mer. Assuming that the word length $\ll n$, where n is the number of letters in the database, there will be $\approx n$ words: Thus the per-word index storage cost is incurred for practically every letter in the database.

In the case of nucleotide or amino acid strings, respectively, a letter can be encoded in 2 or 5 bits. However, the size of the compressed inverted list is still 6 – 18 bits for each posting, and, as previously mentioned, a posting is required for practically every letter. Therefore, the compressed index of a biological sequence database will be somewhat larger than the uncompressed text. The excessive relative size of compressed biological sequence database indices makes it attractive to discover more efficient representations.

Aims

An opportunity for improving biological sequence database index compression exists when redundancy in the database is sufficient that entire database records may be *subsumed* by a *superior* database record, i.e., if some database records are sub-sequences of others. This is in some ways similar to the work of Bernstein and Zobel (2005) on the content equivalence of documents. However, here the relation is generalised from equal-to $(=)$ to greater-than-or-equal-to $(\geq)$. For example, if a database contains the two records "capillary" and "pill", the latter can be subsumed by the former, since "capillary" contains the entire text of the record "pill": The record "pill" provides no content that is not already contained in "capillary", i.e., "capillary" $\geq$ "pill". (Partial redundancy, as between "capillary" and "laryngitis", or between "capillary" and "filler", is addressed in Chapter 6).

Subsumed sequences do not need to be indexed, since any HSP discovered in the superior record can be translated into the subsumed record. Because the entire subsumed sequence exists in the superior sequence, the entire alignment can be translated — avoiding duplication of HSP discovery, assembly and finishing activities. Since these activities are no

longer duplicated, search time is reduced. This is *record folding*, and it is presented as a first generation cooperative compression scheme.

The remainder of this chapter describes the implementation of an index format and construction program, FOLDDB. This index format is integrated into DASH, and results are given for the standard queries used in this dissertation, both with and without record folding. The results show that simultaneous index space and search time savings are possible when record folding is employed. Sensitivity is also slightly improved. It is, therefore, a proof of the thesis of this dissertation, in that it excludes material from the search without reducing the thoroughness of the search, thus reducing reducing search space and time requirements without compromising search sensitivity.

5.1 FOLDDB Index Structure And Algorithm

This section defines the FOLDDB indexing algorithm and index structure, that implements record folding of biological sequence databases. The FOLDDB algorithm is optimised for fast memory resident searching, rather than size. Therefore, aside from record folding, few measures are used to contain the index size, and only to ensure that the index could fit into the RAM of the computers used during development.

5.1.1 FOLDDB Index Structure

5.1.1.1 Text-Partitioned Structure

The final size of an inverted list is not the only difficulty when constructing the index of a database. The memory requirements during index construction grow with increasing lexicon size (i.e., number of unique “words” or *k*-mers in the database) and database size. Because fixed-width words are used when indexing a biological sequence database, the

maximum size of the lexicon can be determined before indexing commences, easing the construction problem somewhat. However, the database size remains unbounded.

One technique for dealing with the unbound database size is to divide the input text into a number of fixed sized partitions, and to index each in turn (Witten et al. 1999). These partitioned indices can be efficiently combined to form the final index.

However, there is no actual requirement to combine the index partitions: each is a valid index in its own right. The drawback of not combining the partitions is that some compression leakage occurs. The degree of leakage reduces as the partition size increases, suggesting that relatively large partitions should be acceptable. Moreover, certain advantages arise in the current situation from maintaining the partitioned structure. These advantages pertain to search efficiency and parallelism.

Search efficiency benefits include the enhancement of query striding, because of the greater variation in word frequencies that results from averaging less data. Further, by considering only a portion of a larger database at any one point less state is required to be held in RAM. While this can be resolved in software by using advanced data handling arrangements, it is more difficult to do the same in hardware. As the DASH algorithm was originally designed to operate in hardware as well as software the decision was made in favour of simplicity and hardware compatibility.

Parallelism is aided in that each partition can be simultaneously constructed or searched without contention. While it is possible to efficiently create and maintain a single index (Lester et al. 2005), there remain certain advantages to maintaining a partitioned index. A partitioned index can be fully RAM resident on a cluster of computers in a much simpler way than can be a single large index. This is particularly attractive when using a heterogeneous cluster to perform the searching. Again, when implementing a search system in hardware (or a combination of hardware and software workers), it greatly simplifies the

process if each worker can operate on discrete units of data, ensuring that the search process can be performed without synchronisation or consistency considerations.

5.1.1.2 Partition Layout

The FOLDDB index format divides the indexed database into a number of partitions. Each partition contains up to 65,000 database records, including their lengths, FASTA format descriptions, and a compressed inverted list for every *k*-mer, for some fixed *k*.

The FASTA format descriptions are stored in plain ASCII text, as they are relatively small when compared to the combined size of the inverted lists and the sequence bodies. The nucleotide sequences are stored using four bits per base in order to allow encoding of all 15 IUPAC nucleotide codes, and rapid access by the DASH search engine that uses the same internal representation during HSP construction. Protein sequences are stored in ASCII format, hence consuming eight bits per base.

None of these representations are particularly space efficient, reflecting that this index structure is optimised for speed, not size. (A more space efficient representation is addressed in the NP3 database representation of Chapter 6 and the NIX index representation of Chapter 7).

5.1.1.3 Compression Of Inverted Lists

The entries, i.e., postings, in the inverted lists are 32 bit values that encode the offset of the first letter of the *k*-mer the posting relates to, relative to the start of the partition. The record numbers, i.e., ordinal position of the record in the partition, are not explicitly recorded, as they can be computed from a table of cumulative record lengths.

A number of coding schemes could be applied to the inverted lists, such as the δ- and γ-Elias codes (Bentley and Yao 1976, Elias 1975), global Bernoulli (Gallager and Voorhis

1975, Golomb 1966), local Bernoulli (Witten et al. 1992, Bookstein et al. 1992), skewed Bernoulli (Moffat and Zobel 1992, Teuhola 1978), Hyperbolic (Schuegraf 1976), batched frequency (Moffat and Zobel 1992), Interpolative (Moffat and Stuiver 2000, 1996), or Selector (Anh and Moffat 2005, 2004).

Instead, a fast ad hoc byte-aligned approach was devised, based on the prefix omission method described by Choueka et al. (1988) which is normally used for compressing lexicons: for each set of four successive pointers a control byte indicates the number of bytes that must be replaced in the previous pointer in order to produce the new pointer. The modified bytes then follow. Figure 5.1 demonstrates how this scheme works in practice. The eight non-zero eight-digit hexadecimal numbers on the left hand side of the figure are the index postings that are being compressed. For each successive word, determine the number of consecutive bytes that must be replaced in the previous index posting in order to transform it into the new posting. Processing begins assuming that the hexadecimal value 00000000 has already been recorded. Thus for the posting 00038924 the bottom three bytes must be replaced. Two control bits are then generated indicating the number of bytes replaced minus one. This is repeated for each index posting. After every fourth index posting the control bits are collected into a single control byte, which is written to the output stream, followed by the data bytes it indicates.

Certain storage inefficiencies occur because there are only two control bits. For example, bytes are wasted if the lowest order byte(s) are identical, but higher order bytes differ, as in the case of index postings 000581dc and 000ab0dc. While only the middle two bytes differ, our scheme must write out three bytes. Nonetheless, this scheme has the compelling advantage of being very fast, because it almost exclusively uses byte-aligned operations.

This ad hoc approach coded below the zero-order entropy of each byte of the 32 bit pointers when indexing the Human UniGene (nucleic acid) database. While it is certain that this does not provide the best compression possible, it does allow for very rapid decompression

```
                #  ctl data         control   data
00000000
00038924        3  10  24 89 03    10011000 24 89 03 87 fe 02 81 05 a9
0003fe87        2  01  87 fe
00058102        3  10  02 81 05
000581a9        1  00  a9
000581dc        1  00  dc           00100110 dc dc b0 0a 64 b5 72 89 0b
000ab0dc        3  10  dc b0 0a
000ab564        2  01  64 b5
000b8972        3  10  72 89 0b

output  98 24 89 03 87 fe 02 81 05 a9 26 dc dc b0 0a 64 b5 72 89 0b
```

Figure 5.1: Fast Ad Hoc Index Posting Compression Algorithm. Only the low order bytes that have changed are recorded for each successive index posting (as indicated by the shaded bytes in the index postings). A control byte is required for each four postings to indicate the number of bytes recorded for each index posting.

---

of the inverted lists. Combined with the clear text representations of the sequence bodies and descriptions, the net result is a database and index ensemble which, while hardly compact, imposes little processing overhead during the search process. This makes it a useful baseline for observing the effect of using record folding to cooperative compress biological sequence database and index ensembles.

5.1.2 Excluding Stop *k*-mers

In addition to the inverted list compression scheme described above, one other measure is used to control the size of the index, stop *k*-mers. Stop *k*-mers are analogous to stop words in English full text retrieval systems: They are terms that are excluded from the index on the basis of their excessive frequency in the database.

Argument can be made against the exclusion of stop words from an index, on the basis that it makes it impossible to efficiently search for the excluded query terms (Witten et al. 1999). However, these arguments need to be interpreted in the context of biological sequence databases.

In order to raise the biological precision of database queries, the query itself is often filtered to mask so-called *low complexity* regions, which generally correspond to over frequent terms in the database, and hence stop *k*-mers. Among the most common low complexity regions are contiguous runs of a single base or acid, e.g., AAAAA...AAA. There are a number of algorithms commonly used to filter these and similar structures from queries, and included in that list of algorithms are SEG (Wootton and Federhen 1993, 1996), DUST (Hancock and Armstrong 1994), XNU (Claverie and States 1993) and CAFEFILTER (Williams 1999). Therefore, it appears that stop *k*-mers play an important and credible role in maintaining precision in biological sequence sequence search and alignment.

Because of the correlation between stop *k*-mers and regions of low biological complexity (such as long runs of consisting of only one or two of the nucleotides), it is much less reasonable to search for a nucleic acid or protein query that is composed exclusively of stop *k*-mers, than it is to search for an English phrase that consists solely of stop words. Moreover, because each overlapping *k*-mer is indexed, it is extremely difficult to construct a query that is composed entirely of stop *k*-mers, without it being composed entirely of a single base or amino acid.

Consider a sequence composed of a run of A's followed by a run of T's. The frequency of the all A *k*-mer and the all T *k*-mer may be such that both are stop *k*-mers. However, it is unlikely that the *k*-mers consisting of one through $k-1$ A's, followed by $k-1$ through one T's, i.e., AA...AT, ..., AT...TT, would all occur with sufficient frequency that they would also be stop *k*-mers; if only one of the *k*-mers is not a stop *k*-mer, that is sufficient for the query to succeed. Therefore recall is unlikely to suffer.

While recognising that the use of any sort of query filter in biological sequence database searching results in some loss of sensitivity (Williams 1999), FOLDDB employs stop *k*-mers as an effective method of index size reduction without fear of excessive loss of sensi-

tivity. The observed frequency of stop *k*-mers is recorded however, so that DASH's query striding algorithm can function correctly.

The decision to exclude stop *k*-mers was made in order to restrain the size of the index, and is a reflection of the primitive index compression used, rather than on the feasibility of indexing stop words. Indeed, Bell et al. (1993) have shown that with an efficient compression scheme, stop words contribute only slightly to the total index size. Thus, while the FOLDDB index structure does not index stop words, this deficit is addressed by its sequel, NIX, in Chapter 7.

5.1.3 Record Folding As A Prototype Of Cooperatively Compressed Indexing

The key feature of FOLDDB is that it implements a prototypical cooperative compression scheme called record folding. This takes the form of merging the storage (and thus index postings) of records that are sub-sequences of others in a database. The containing sequence is called the *superior* sequence. Each of the folded records are recorded in an appendix to the FASTA description of the relevant superior record.

Alignments that occur against a superior record are translated into the context of each folded record, as illustrated in Figure 5.2. The inner square represents the folded record, while the outer square represents the superior record. Alignment discovery proceeds for the superior record according to the three stages of the DASH algorithm. After this, the intersecting portion of the alignment is translated into the context of the folded record.

The two advantages of this approach are: (a) its lack of impact on the search process, and (b) its computational efficiency. The search process is not impacted, because the unfolding of alignments can be performed as a post-processing step, as described above. The computational efficiency arises for a similar reason; as the search process need only process the

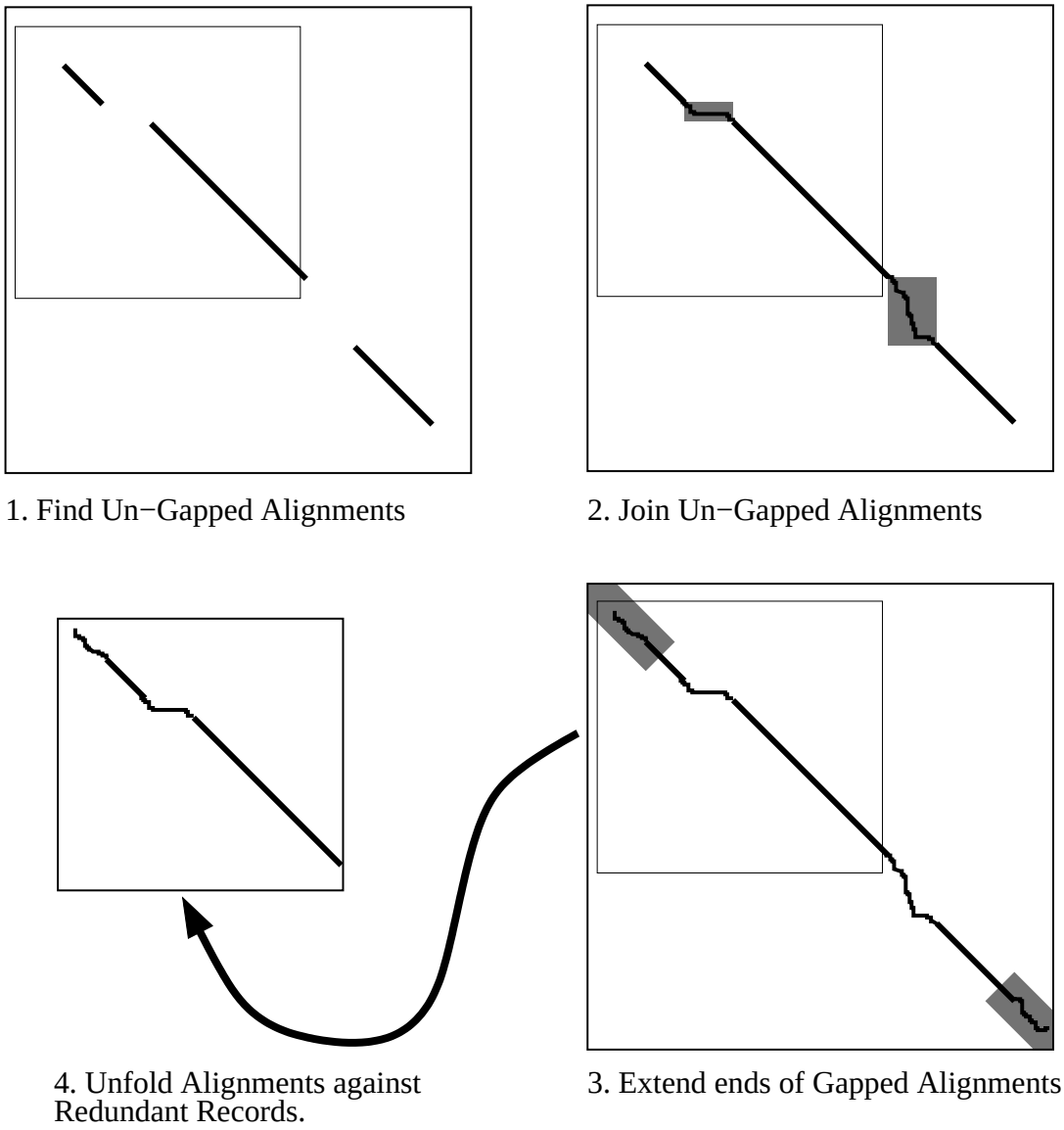

Figure 5.2: Example Of Alignment Unfolding for a Folded Record.

superior record, finding any alignment against a folded record re-uses the work previously carried out against the superior record.

One disadvantage of this approach is that the initial discovery of the foldable records in an unsorted database is time consuming. The heuristic algorithm used in this dissertation is to take each sequence in the database and search forward in the database for candidate sequences. For n sequences this requires $(n-1)+(n-2)+\cdots+(n-(n-1))+(n-n) = \sum_1^{n-1} n$ actions, and thus completes in approximately $O(n^2)$ time. However, this occurs only during index construction, and so the cost is an acceptable one. However, once a database has been constructed, additional sequences could be added in O(n) time, i.e., the time required to search the index.

5.1.4 Construction Of Folded Database Index

A two phase process is used to build the index. Database partitioning, index compression and stop k-mer exclusion are used in both phases; the database is sectioned into partitions that extend until they just exceed either a record count or letter count quota. Within each partition, the inverted lists are compressed using the ad hoc method previously described, with stop k-mer exclusion discarding any very long lists. Pseudo-code for this index construction process is presented in Figure 5.3. Finally, the indices of each partition of the database are concatenated to form a single index file for the database.

The first phase produces a temporary index that is used during the second phase to discover record folding opportunities. Construction of the temporary index is relatively fast, requiring only a few minutes per giga-base on a 750 MHz SUN Ultra-SPARC III processor.

The second phase of the index construction process involves looking for record folding opportunities using the index that was produced during the first phase. The DASH algorithm performs this search by exhaustively searching for every record in the index against every other record in the index. The pseudo code for this process is presented in Figure 5.4.

```
k = index word width
for m = 1 to (database_section_length-k)
  -- Check if this is the beginning of a new record
  if (a record starts here) then
    -- record sequence description
    record_description[record_count] := sequence description
    -- record starting location of the record
    start_of_record[record_count] := m
    -- increment number of records
    increment record_count
  end if

  -- Obtain the k-mer at this position in the database
  t = the k-mer beginning at position m in the database
  -- Record location of this n-mer
  addresses_of_occurrences[t][number_of_occurrences[t]] = m
  -- Increment the number of occurrences of this k-mer
  increment number_of_occurrences[t]
  -- Write the residue to the index
  write residue[m] to index
next m
-- Write list of sequence descriptions
write record_description[0 .. record_count] to index
-- Write list of record starting points
write start_of_record[0 .. record_count] to index

-- Write table of n-mer addresses for each n-mer with
-- a frequency lower than the low selectivity cut-off
for t in (the complete set of k-mers)
  -- Write the number of occurrences of the n-mer to the index
  write number_of_occurrences[t] to index

  -- If the number of occurrences does not reach the cut-off,
  -- write the complete list, otherwise write an empty list
  if number_of_occurrences[t] < low_selectivity_cut_off then
    -- Compress the list of addresses
    compress addresses_of_occurrences[t]
    -- Write the compressed list of addresses to the index
    write addresses_of_occurrences[t][0 .. number_of_occurrences[t]]
  else
    -- Too many occurrences, write only an empty list
    write (empty list) to index
  end if
next t
```

Figure 5.3: Pseudo Code For Index Construction Process.
This fragment describes the process used to index a partition of the database (removal of folded records occurs later). The partition is traversed, and the frequency of each k-mer is calculated, and the address of each occurrence recorded. The starting location and textual description of each record in the database partition is noted. Once the traversal is complete, the list of descriptions is written out to the index. The frequency of each k-mer is also written, along with the compressed list of addresses where it occurs. However, if the k-mer is over abundant in this database partition, then only the frequency is written: The addresses where it occurs are not written.

For this application, the DASH algorithm is re-tuned for rapid approximate searching; it searches for only one *k*-mer per record, resulting in very few index postings requiring investigation. Further, the minimum HSP length is set to the length of the query sequence, and the minimum score to the identity score of the query. In this way, only exact alignments that span the entire query sequence are identified. This allows searching for more than 200 records per second in the ≈ 2 giga-base Human UniGene (nucleic acid) database. Nonetheless, this process required several hours for the Human UniGene (nucleic acid) and GenPept (protein) databases.

This algorithm is fundamentally O(n^2), and better algorithms exist, such as the SPEX variant developed for genomic databases (Bernstein and Cameron 2006) that uses document fingerprinting to perform the all-to-all comparison in roughly O(n) time. However, that algorithm was not published when this work was undertaken. While the O(n^2) algorithm of this dissertation is not the most efficient possible, it was sufficient for testing the hypotheses of this dissertation.

The second phase of index construction is concluded by taking the list of foldable record pairs in the database, and rebuilding the index with those records folded. If more than one superior record is identified for any given foldable record, the longest superior record is chosen. In the case of a tie, the superior record that is nearest the end of the database wins. This minimises the number of superior records required to contain all foldable records. If superior records were chosen arbitrarily from the valid possibilities, then chains or cycles of reference could result, complicating the process.

All superior records are tagged with the description of any folded record(s) they contain. The pseudo code for writing the cooperatively compressed index and recording the record folding information is presented in Figures 5.5 and 5.6 respectively. This index writing process takes only a few minutes, and is performed using a user specified stop *k*-mer frequency

```
-- Search for each record in the database
for s = 1 to index.record_count
  -- Obtain the record from the database
  query = index.record[s]
  -- Search for it in the database.  Look for only
  -- exact full length alignments.
  alignments = dash.search.exact(query,index)
  -- Exclude the query sequence itself from the results
  alignments = alignments - subset(alignments,query)
  -- Keep only the longest alignment(s)
  alignments = subset(alignments,longest)

  -- Keep only the last alignment in the list
  alignments = alignments[alignments.last]

  -- Reject the alignment host if it is the same length as
  --   the query, and occurs earlier in the database
  if (alignment.host.length == query.length) then
    if ( alignment.host.number < query.number ) then
      -- Alignment host is a record that is identical to the
      -- query in content and length, and occurs earlier in the
      -- database.  Therefore it is a record that we make
      -- redundant, so don't say that we are made redundant by
      -- it!
      alignment = (empty list)
    end if
  end if

  -- If we still have an alignment, record it as the superior record
  -- of this record.  Also record its location in the superior record
  if ( alignment != (empty list) then
    -- Mark this record as foldable
    record_is_foldable[s] = true
    -- Record which record contains it
    record_superior[s] = alignment.superior
    -- Record where it occurs in the superior record
    record_offset[s] = alignment.subject_start
    -- The foldable record is also recorded against the superior
    -- record for during index reconstruction
    list.append(made_foldable_by[alignment.superior], query)
  end if
next s
```

Figure 5.4: Pseudo Code For Index Construction Process.
This fragment describes the process used to identify foldable records within a database. The temporary index is queried to find the superior records, if any, of each record. Only exact full-length alignments are used. The alignments are sifted to leave only the longest and latest appearing superior record. If this superior record is the same length as the query sequence, it is checked that it occurs later in the database to prevent cycles forming. If an alignment passes these tests, it is a superior record, and the query sequence is marked foldable.

threshold. The resulting record folded index will vary in size according to the degree of redundancy it has identified and removed.

5.2 Searching Folded Databases With DASH

As previously described, the only change made to the search process to support the searching of database and index ensembles cooperatively compressed using record folding is the addition of an alignment unfolding post-processing step. Alignments are unfolded by checking each to see if they occur in a superior record. If so, the list of folded records the superior record contains is consulted. Where any of the folded records intersect with the alignment, the intersecting portion of the alignment is translated into the context of the folded record. This contributes negligibly to the total search time, and has the advantage that it has time complexity proportional to the number of alignments, and not the size of the database. Thus we have the potential for sub-linear search time versus data-base size.

5.3 Method

To evaluate the effect of the record folding cooperative compression technique, control, i.e., non-record folding, indices of the Human UniGene (nucleic acid) and GenPept (protein) databases were constructed. Stop *k*-mer frequency thresholds of $2.5 \times E_{random}$ and $10 \times E_{random}$ were used, respectively. These thresholds were determined empirically as being small enough to enable the indices to fit into the main memory of the computers on hand, and large enough to not significantly detract from search sensitivity. Folded versions of these indices were then created, using the same stop *k*-mer frequency thresholds.

DASH and FOLDDB were compiled as single-threaded 32bit-applications, with optimisation flag `-O2` passed to the C compiler. The configurations of Tables A.9 and A.10 were

```
k = index word width
skipping = false
for m = 1 to (database_section_length-k)
  -- Check if this is the beginning of a new record
  if (a record starts here) then
    if (record_is_foldable[sequence description]) then
      -- This record is marked as foldable, so don't record it
      skipping = true
    else
      -- record is not foldable, so record it normally
      -- record sequence description
      record_description[record_count] := sequence description
      -- record starting location of the record
      start_of_record[record_count] := m
      -- increment number of records
      increment record_count
    end if
  end if

  -- Only write record data out if the record is not
  -- redundant
  if (skipping == false) then
    -- Obtain the k-mer at this position in the database
    t = the k-mer beginning at position m in the database

    -- Record location of this k-mer
    addresses_of_occurrences[t][number_of_occurrences[t]] = m

    -- Increment the number of occurrences of this k-mer
    increment number_of_occurrences[t]

    -- Write the residue to the index
    write residue[m] to index
  end if
next m
```

Figure 5.5: Pseudo Code For Index Construction Process.
The index is rebuilt, using the information of which records can be folded. This is identical to the first pass of index construction, except that folded records are not included in the index.

```
-- Write list of record descriptions
write record_description[0 .. record_count] to index
-- For each non-folded record in the database ...
for s = 1 to record count
  -- ... see if it is the superior of any other records
  if (made_foldable_by[s] != (empty list) then
    -- Yes: write the information for each record made
    --   redundant by this record
    for r = 1 to made_redundant_by[s].length
      -- Write description of record
      write made_foldable_by[s][r].description to index
      -- Write offset of folded record in its host
      write record_offset[made_foldable_by[s][r].record] to index
    next r
  end if
next s
-- Write list of record starting points
write start_of_record[0 .. record_count] to index

-- Write table of n-mer addresses for each n-mer with
-- a frequency lower than the low selectivity cut-off
for t in (the complete set of k-mers)
  -- Write the number of occurrences of the n-mer to the index
  write number_of_occurrences[t] to index

  -- If the number of occurrences does not reach the cut-off,
  -- write the complete list, otherwise write an empty list
  if number_of_occurrences[t] < low_selectivity_cut_off then
    -- Compress the list of addresses
    compress addresses_of_occurrences[t]
    -- Write the compressed list of addresses to the index
    write addresses_of_occurrences[t][0 .. number_of_occurrences[t]]
  else
    -- Too many occurrences, write only an empty list
    write (empty list) to index
  end if
next t
```

Figure 5.6: Pseudo Code For Index Construction Process.
This fragment describes how the record descriptions and other information are written. This is similar to the first pass, but with the addition of the information necessary to record that each folded record is stored in its respective superior record.

used for the DASH M2 and DASH M4 variants, respectively. Protein queries were run twice: (a) with an ordinary index, and; (b) with an index cooperatively compressed using record folding. All queries were executed using the batching environment described in Chapter 3.

Record folding had practically no effect on the Human UniGene (nucleic acid) database, folding only 0.1% of the 3.6 million records. This is because there are multiple codons that encode each protein, and therefore it is uncommon to see identical nucleotide sequences, even when they encode exactly the same protein. Therefore, only the results for the ordinary index of the Human UniGene (nucleic acid) database are presented. Finally, because record folding targets large scale repetitions, this method was not attempted on the Human Genome database, because that database contains little redundancy of this kind.

5.4 Results And Discussion

Tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6, and Figures 5.7 and 5.8 present the database and index size, search speed and sensitivity data for the negative-control indices built from the Human UniGene (nucleic acid) and GenPept (protein) databases, and the record folded index built for the GenPept (protein) database. As previously mentioned, record folding had practically no effect on the Human UniGene (nucleic acid) database. Therefore no results are reported for the Human UniGene (nucleic acid) database with a record folded index.

5.4.1 Effect Of Sequence Folding

Considering the control indices first, Tables 5.3 and 5.5, and Figure 5.7 show that DASH+FOLDDB is competitive with the surveyed algorithms for nucleotide searching. DASH M2 posts the fastest search time among the group, while simultaneously delivering

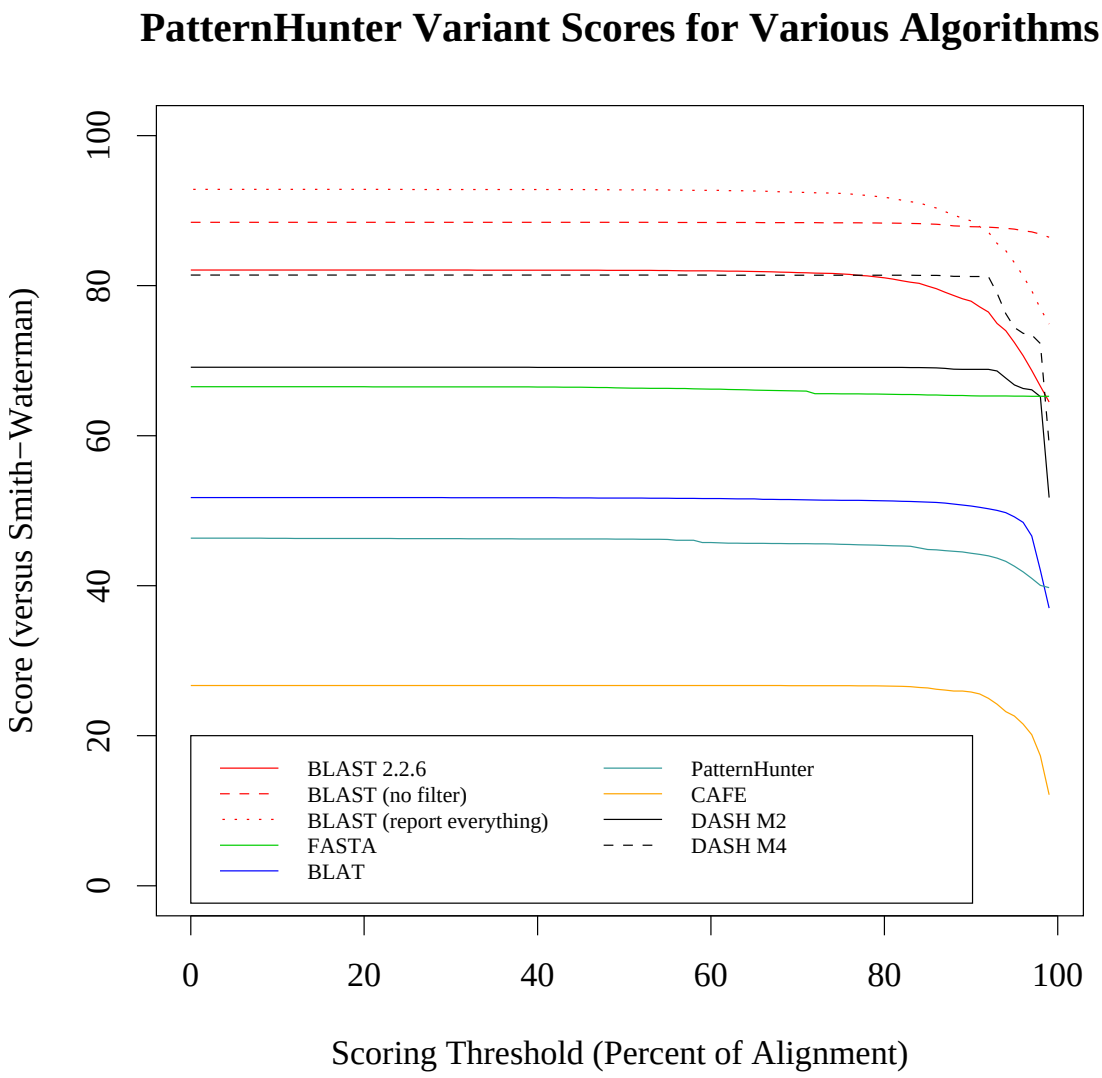

Figure 5.7: PatternHunter Variant Scores (See Section 3.3) For Nucleic Acid Queries (Using The Human UniGene (Nucleic Acid) Database). The Smith-Waterman algorithm is used as the benchmark.

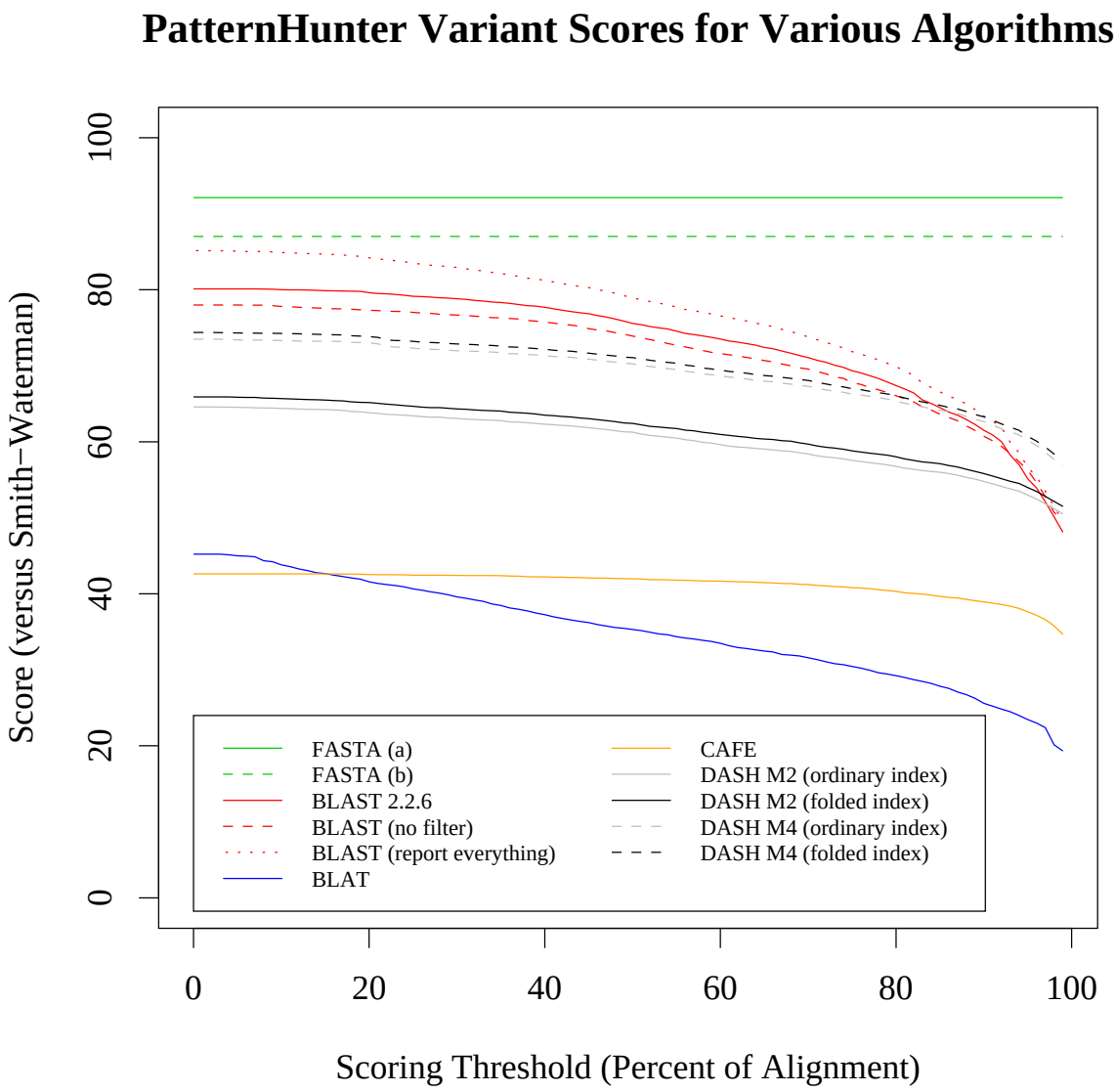

Figure 5.8: PatternHunter Variant Scores (See Section 3.3) For Protein Queries (Using The GenPept (Protein) Database). The Smith-Waterman algorithm is used as the benchmark.

good sensitivity. The situation is similar for protein searching, with Tables 5.4 and 5.6, and Figure 5.8 showing that DASH+FOLDDB is competitive with the surveyed algorithms. However, the FOLDDB indices are very large, second in size only to that of CAFE.

Turning to the GenPept index that was cooperatively compressed using record folding, Table 5.4 shows that search times were reduced by about one seventh. This shows clear evidence of sub-linear search time with respect to (uncompressed) database size.

Table 5.6 and Figure 5.7 show that this speed improvement does not involve a sensitivity trade-off. Rather, sensitivity scores are slightly increased (Two-Sided Wilcoxon Signed Ranks Test (Wilcoxon 1945), p-value = 0.00014).

The question arises: Is this gain in sensitivity due to the use of cooperative compression, or due to some other cause? There is one potential cause that must be considered, that is the potential for improved sensitivity caused by the normalisation of some *k*-mer frequencies that results from folding similar records. Whenever a record is folded, the frequencies of the *k*-mers it contains are slightly reduced. This effect can be sufficient to cause some *k*-mers to drop below the stop *k*-mer threshold, allowing them to be included in the index. In this way the search process becomes more thorough.

Fortunately, any gains that result from the inclusion of more material in the index can be disambiguated from any gains that result from cooperative compression. This is because only the use of cooperative compression can allow the discovery of alignments that do not contain an exactly corresponding region at least as long as the index is wide.

In other words, we can be sure an alignment resulted from cooperative compression alone if it is not possible for it to be discovered using any *k*-wide index. If even a single alignment can be produced that does not include *k* consecutive matching amino acids, then cooperative compression must be responsible for its discovery. Refer to Figure 5.9 where several of the 1,102 examples are presented of this occurring with the DASH M4 queries against the GenPept (protein) database.

Therefore, and as a result of cooperative compression, search sensitivity has been increased, while search time and space requirements have been reduced, thus confirming the thesis of this dissertation. Moreover, with a finer grained approach there is the hope of further sensitivity gains, because in that case the entire population of discovered HSPs could be cloned before being thinned out during the later stages of the DASH algorithm.

Finally, as Table 5.2 shows, the record folding process reduced the indexed database size from 1,669 MB to 1,374 MB, thus demonstrating that it is possible to construct a search system with sub-linear space requirements. Despite this one sixth reduction in size, the FOLDDB indexed GenPept (protein) database is still substantially larger than that required by all other algorithms, with the exception of CAFE.

5.4.2 Effect Of Query Length On Search Time

It has been previously stated that a desirable characteristic of a sequence search and alignment tool is that search time be predictable based on the search space. For a situation where the database is constant, this translates to the search time being predictable based on the query length. This response is plotted in Figures 5.11, 5.13, 5.10 and 5.12 for both DASH and BLAST and for both the UniGene nucleotide database and UniGene protein database to give an indication of the behaviour of the two algorithms. For both databases both algorithms exhibits a more or less linear relationship between query length and search time. However, DASH suffers from excessive search time for a few queries, and greater general variability in search time than BLAST for protein queries.

As DASH+FOLDDB does not involve the traversal of recurrence chains (recall that sequence unfolding is done after alignments have been discovered), we must conclude that the DASH algorithm itself, while generally faster than BLAST, is in fact has a search time less readily predictable than does BLAST.

```
> gi|28279202|gb|AAH45967.1| (BC045967) Similar to splicing factor 30,
  survival of motor neuron-related [Danio rerio]
  vs gi|32488904|emb|CAE03655.1| (AL606691) OSJNBa0060N03.20
  [Oryza sativa (japonica cultivar-group)]
: (130-191) = (577-641), score=96.
Query: 130 DEIDGKPKSKKELQAEQREYKKKKAQKKVQRMKELEQER---EDQKSKWQQFNNKAYSKN 186
           ||++ | +|+ | + | ++ |+++ +|+++|+|+ |+|| + | +|+| ++ ++|| +|+
Sbjct: 577 DELERKKRSQDEKRKELEKQKQEEERKELDRQKQREEERKAKELEKQKQREEERKALEKQ 636
Query: 187 KKGQ 190
           |+|+
Sbjct: 637 KQGE 640

> gi|7228887|gb|AAF42677.1|AF226529_1 (AF226529) membrane protein GNA1220
  [Neisseria meningitidis]
  vs gi|25004946|emb|CAB03018.2| (Z81072) C. elegans STL-1 protein
  (corresponding sequence F30A10.5) [Caenorhabditis elegans]
: (114-150) = (136-172), score=89.
Query:  114 RSVIGRMELDKTFEERDEINSTVVAALDEAAGAWGV 149
            || +|+++|| +|+|+|+ +|+++| |+++|++ ||+
Sbjct:  136 RSEVGKINLDTVFKERELLNENIVFAINKASAPWGI 171

> gi|28279202|gb|AAH45967.1| (BC045967) Similar to splicing factor 30,
  survival of motor neuron-related [Danio rerio]
  vs gi|28830036|gb|AAO52526.1| (AC116957) similar to Xenopus laevis
  (African clawed frog). DNA ligase I (EC 6.5.1.1)
  (Polydeoxyribonucleotide synthase [ATP]) [Dictyostelium discoideum]
: (33-189) = (347-497), score=77.
Query:   33 LQKDLQEVIELTKDLLTSQPAEGTTS--TKSSETVAPSHSWRVGDHCMATWSQDGQVYEA 90
             | +  +|+|+++ ||+    ||+ +|  |++|++   + + +|     || ++ + | ||
Sbjct:  347 LTSPKKETIDIS-DLFKRANAEAKSSVPTSTSKNSKTNKKQKVDHKPTATTKKPSPVLEA 405
Query:   91 EIEEIDNENGTAAITFAGYGNAEVMPLHMLKKVEEGRIRDEIDGKPKSKKELQAEQREYK 150
             +      ++ |++ | +  ++| ++    +++ ++ + ++ | +| |+ +  ++| ++ |
Sbjct:  406 K------QSTTTTTTTTTTSTATTISSKSISSPSKKEEKEVITSK-KQVEATKVEVKKEK 458
Query:  151 KKKAQKKVQRMKELEQEREDQKSKWQQFNNKAYSKNKK 188
             +|+ +|+ +  +| |+|+||+ +| ++++++ |+++++
Sbjct:  459 EKEKEKEKEDDEEEEEEEEDDDEKLEDIDEEEYEEEEE 496
```

Figure 5.9: Several Alignments From Search Results Due To Cooperative Compression. The index width was $k = 3$, but none of these alignments contain three consecutive matching acids, confirming that they could not have resulted from direct index look ups, but must instead be the result of cooperative compression.

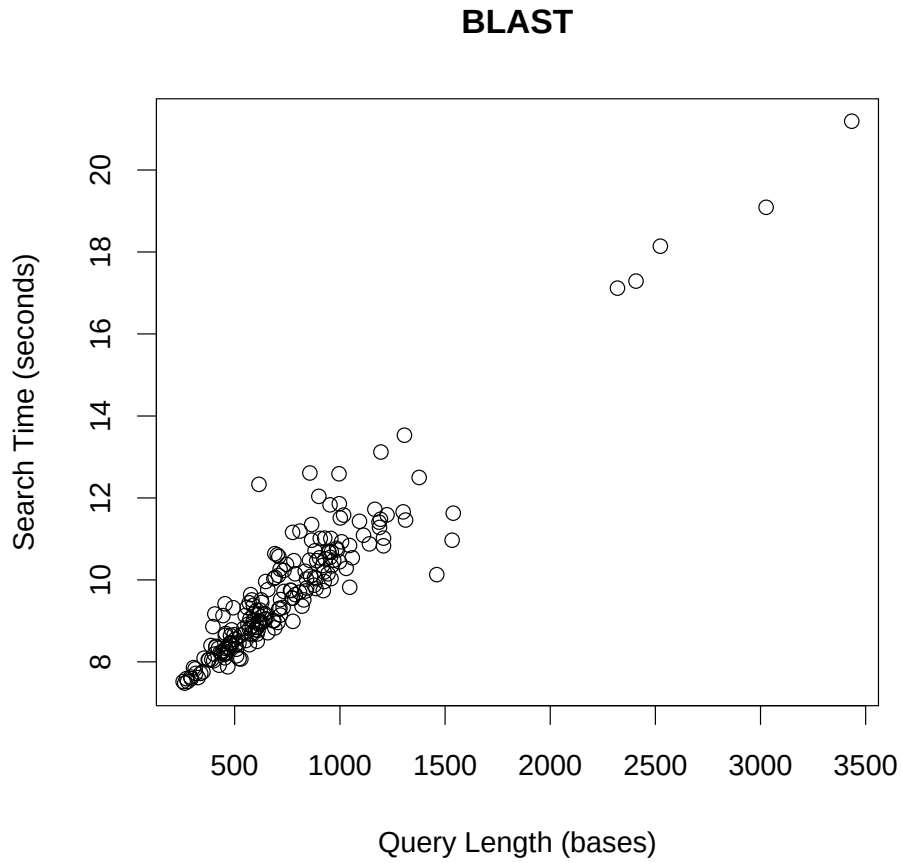

Figure 5.10: Search Time Versus Query Length For BLAST Searching The UniGene Nucleotide Database.

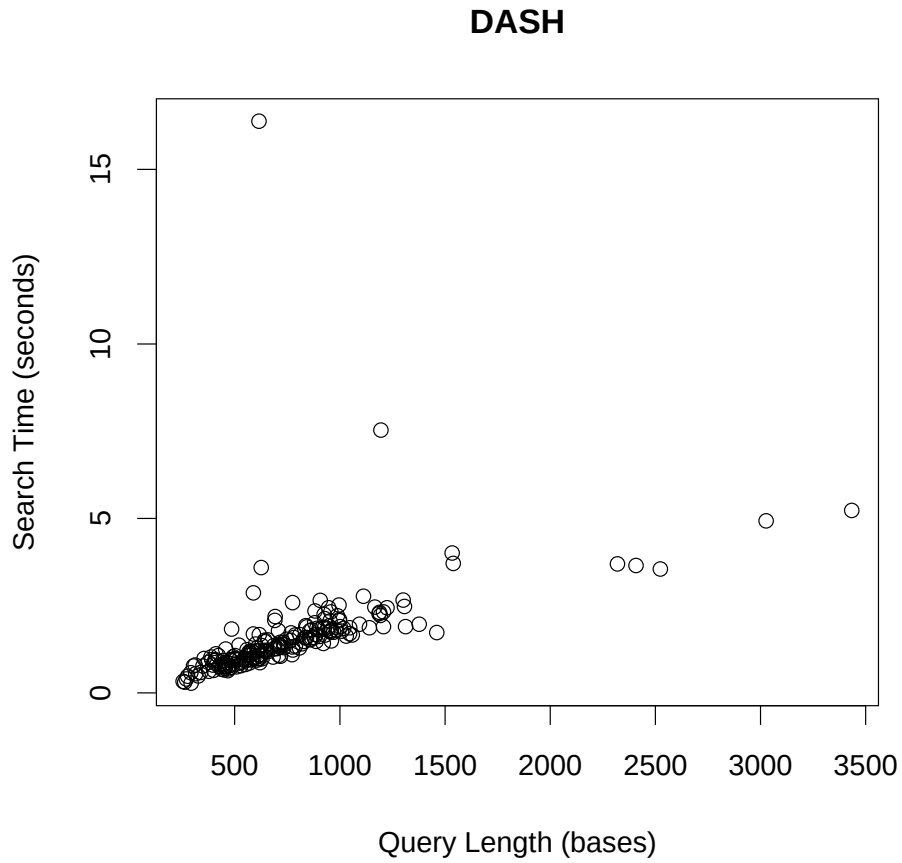

Figure 5.11: Search Time Versus Query Length For DASH (Mode 2) Searching The UniGene Nucleotide Database.

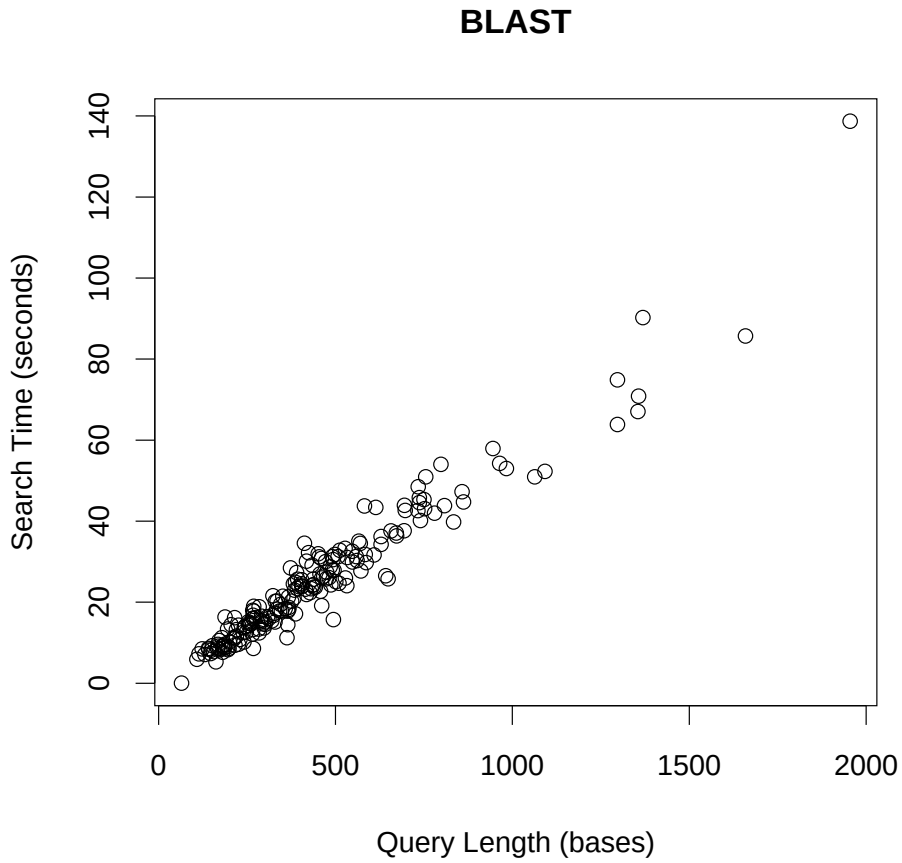

Figure 5.12: Search Time Versus Query Length For BLAST Searching The UniGene Protein Database.

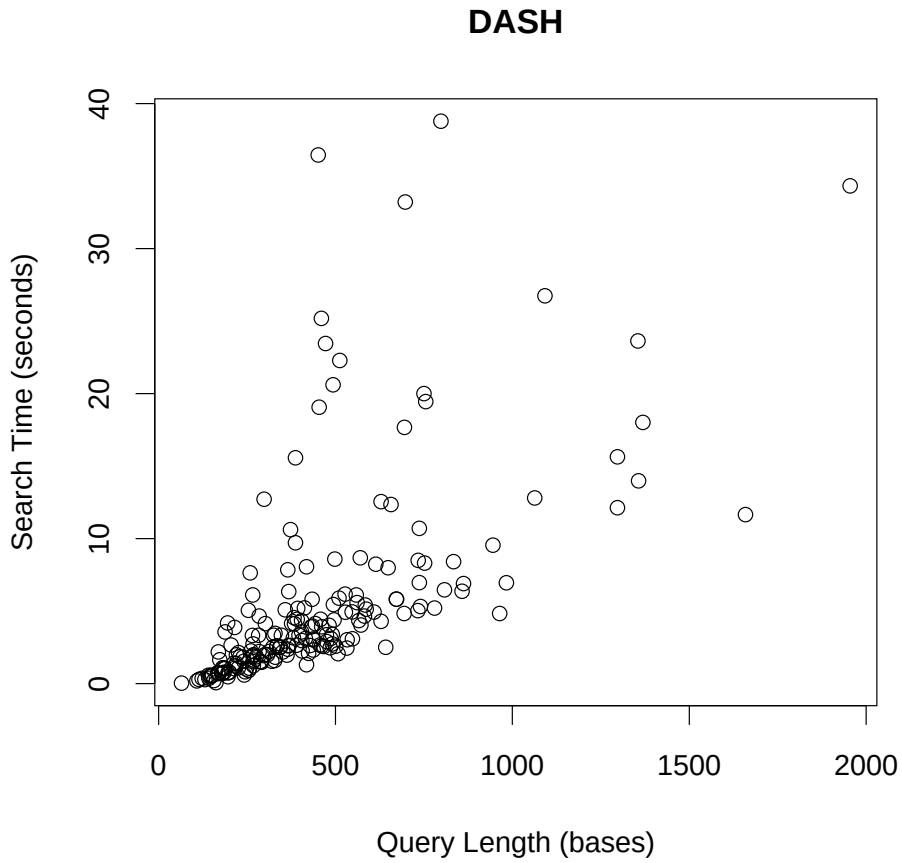

Figure 5.13: Search Time Versus Query Length For DASH (Mode 2) Searching The UniGene Protein Database.

5.5 Conclusions

It has been shown that search time and index space savings can be made by employing the record folding method of database and index cooperative compression — provided abundant redundancy of whole records exists. These time and space savings were accompanied by slightly improved sensitivity. These results support the thesis of this dissertation, that it is possible to reduce the time and space costs of the search process, while increasing the sensitivity of the search. In the present case, the search space and time requirements were reduced, respectively, by 18% and between 6% and 13%. At the same time sensitivity increased slightly, but consistently.

Importantly, these results simultaneously improving search time, space and sensitivity are applicable to all adequately redundant nucleic acid databases. The rapidly growing EST and shotgun genome sequencing databases of GenBank would meet these criteria. Thus the findings of this chapter offer one solution to the continued management and search of such data.

Together, these positive results suggest that it is worth exploring finer grained cooperative compression techniques, so that: (a) similar gains can be made for databases that contain common sequences, but where foldable records are rare or non-existent, such as in the Human UniGene (nucleic acid) database or the Human Genome (nucleic acid) database, and; (b) the sensitivity gains of the thesis of this dissertation can be tested in less redundant databases.

Such a fine grained cooperative compression technique is developed in the following two chapters, where, respectively, (a) a compact database representation, NP3, is developed that makes available information about common sequences, and (b) a more compact index structure, NIX, is devised that makes use of the common sequences encoded in an NP3 formatted file.

Table 5.1: Human UniGene (Nucleic Acid) Database And Index Sizes In Megabytes (MB) And Bits Per Base (B/B). Results are shown for the ordinary index only, as this database did not differ noticeably when using the folded index.

Format	Bodies Only		Descriptions		Index		Total	
	MB	B/B	MB	B/B	MB	B/B	MB	B/B
DASH+FOLDDB (ordinary index)	936	4.00	470	2.04	2,200	9.39	3,606	15.43
Smith-Waterman* (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
BLAST (formatdb)	489	2.09	660	2.83	40	0.17	1,189	5.09
BLAT* (faToTwoBit)	630	2.70	-	-	1,088	4.66	1,718	7.36
PatternHunter** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
FASTA** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
CAFE*** (CAFE Index)	496	2.12	102	0.44	6,961	29.79	7,634	32.67

* Indicates that program maintains an index in RAM, and that the database format contains both record bodies and descriptions (BLAT).

** Indicates that algorithm indexes during searching (PatternHunter and FASTA), or does not use an index (Smith-Waterman).

*** Indicates that multiple small indices were used instead of one large index, due to technical difficulties (CAFE).

Table 5.2: GenPept Protein Database And Index Sizes In Megabytes (MB) And Bits Per Acid (B/A).

Format	Bodies Only		Descriptions		Index		Total	
	MB	B/A	MB	B/A	MB	B/A	MB	B/A
DASH+FOLDDB (ordinary index)	470	8.00	142	2.42	1,057	17.99	1,669	28.40
DASH+FOLDDB (folded index)	392	6.67	142	2.42	840	14.30	1,374	23.39
Smith-Waterman* (FASTA ASCII)	479	8.15	142	2.42	-	-	621	10.57
BLAST (formatdb)	473	8.05	194	3.31	231	3.94	899	15.31
BLAT* (FASTA ASCII)	479	8.15	142	2.42	-	-	621	10.57
FASTA* (FASTA ASCII)	479	8.15	142	2.42	-	-	621	10.57
CAFE** (CAFE Index)	480	8.17	22	0.37	1,621	27.59	2,236	38.06

* Indicates that algorithm indexes during searching (FASTA), or does not use an index (Smith-Waterman).

** Indicates that multiple small indices were used instead of one large index, due to technical difficulties (CAFE).

Table 5.3: Comparison Of Nucleotide Search Speed (Using The Human UniGene (Nucleic Acid) Database). Results are shown for the ordinary index only, as this database did not differ noticeably when using the folded index. DASH (M2, ordinary index) is the fastest algorithm here.

Format	Search Time (seconds)			Search Time
	mean	median	total	(×BLAST)
DASH+FOLDDB (M2, ordinary index)	1.582	1.345	316.46	0.16
DASH+FOLDDB (M4, ordinary index)	22.20	19.45	4439.34	2.27
Smith-Waterman	16,260	14,070	3,251,827	1660.56
NCBI-BLAST 2.2.6 (Default)	9.79	9.4	1,958.27	1.00
NCBI-BLAST 2.2.6 (No Filter)	21.37	10.16	4,274.56	2.18
NCBI-BLAST 2.2.6 (Report Everything)	49.93	11.33	9,985.01	5.10
BLAT*	2.10	2.07	471	0.21
PatternHunter**	78.37	78.61	15,673.57	8.00
FASTA	530.05	534.69	106,010.29	54.13
CAFE***	32.75	30.76	6,693.46	3.42

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 5.4: Comparison Of Protein Search Speed (Using The GenPept (Protein) Database). DASH (M2, folded index) is the fastest algorithm here.

Format	Search Time (seconds)			Search Time (×BLAST)
	mean	median	total	
DASH+FOLDDB (M2, ordinary index)	6.76	4.51	1,352.41	0.27
DASH+FOLDDB (M2, folded index)	5.68	3.99	1,136.36	0.23
DASH+FOLDDB (M4, ordinary index)	34.68	21.90	6,935.75	1.37
DASH+FOLDDB (M4, folded index)	29.08	19.99	5,815.57	1.15
Smith-Waterman	1674.00	1397.00	334,794.20	66.32
NCBI-BLAST 2.2.6 (Default)	25.24	22.01	5,047.81	1.00
NCBI-BLAST 2.2.6 (No Filter)	35.33	26.48	7,066.00	1.40
NCBI-BLAST 2.2.6 (Report Everything)	71.00	24.55	14,200.32	2.81
BLAT*	85.45	80.40	17,004.37	3.39
FASTA (a)	296.00	273.36	59,199.23	11.73
FASTA (b)	83.36	84.38	16,672.14	3.30
CAFE***	11.88	10.25	2,375.71	0.47

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 5.5: Nucleotide Sensitivity Scores (PatternHunter Variant) Versus The Results Of The Smith-Waterman Algorithm (Using The Human UniGene (Nucleic Acid) Database). Results are shown for the ordinary index only, as this database did not differ noticeably when using the folded index. Apart from BLAST, DASH is the most sensitive algorithm here, being much more sensitive than BLAT, which is the next fastest algorithm after DASH.

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
DASH (M2, ordinary index)	69.11	69.10	68.85	68.61	65.19
DASH (M4, ordinary index)	81.41	81.39	81.24	79.00	72.28
Smith-Waterman	100	100	100	100	100
BLAST (Default)	82.04	81.64	78.69	74.97	66.58
BLAST (No Filter)	88.43	88.38	87.96	87.73	86.86
BLAST (Report Everything)	92.78	92.36	89.41	85.62	76.96
FASTA	66.41	65.59	65.36	65.28	65.25
BLAT	51.69	51.41	50.88	50.04	42.06
PatternHunter	46.22	45.58	44.59	43.67	40.05
CAFE	26.69	26.68	25.96	24.2	17.32

Table 5.6: Protein Sensitivity Scores (PatternHunter Variant) Versus The Results Of The Smith-Waterman Algorithm (Using The GenPept (Protein) Database). DASH is beaten in sensitivity only by the FASTA algorithm, which is around 50$\times$slower than DASH.

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
DASH+FOLDDB (M2, ordinary index)	61.47	57.88	55.32	53.80	51.18
DASH+FOLDDB (M2, folded index)	62.66	59.16	56.38	54.79	52.16
DASH+FOLDDB (M4, ordinary index)	70.48	66.66	63.31	61.30	57.67
DASH+FOLDDB (M4, folded index)	71.28	67.38	63.96	61.92	58.39
Smith-Waterman	100	100	100	100	100
FASTA (a)	92.12	92.12	92.12	92.12	92.12
FASTA (b)	87.02	87.02	87.02	87.02	87.02
BLAST (Default)	76.15	70.08	62.92	58.27	50.13
BLAST (No Filter)	74.35	68.55	62.09	58.44	50.72
BLAST (Report Everything)	79.56	72.69	64.98	59.93	51.5
BLAT	35.56	30.82	26.73	24.48	20.15
CAFE	42.02	40.96	39.26	38.38	35.77

Part III

Cooperative Compression Of Less Redundant Nucleic Acid Databases

Chapter 6

NP3: Compressing Sorted Nucleic Acid Databases

Introduction

Size And Entropy Of Biological Sequence Databases

There is little doubt that biological sequence databases are growing at a rate that exceeds both that of hard disk storage capacities and Moore's Law. However, while the volume of sequenced data is increasing exponentially, the increase in entropy must be closer to linear, perhaps even sub-linear. Four factors that strongly suggest that entropy grows more slowly than database size are: (a) the similarities between species; (b) the homology among individuals of a species; (c) the occurrence of similar sequences within an organism, and; (d) the super-redundant shotgun sequencing method commonly used in high-throughput sequencing. Taken to the logical conclusion, where every living thing on the planet is sequenced, it would be absurd to expect the entropy of the resulting database to be the sum of the entropy of each genome. Otherwise sequence alignment would not be a useful tool, as there would be no similar sequences to align.

It is reasonable to expect then, that it should be possible to produce compression algorithms that take advantage of this shared entropy to produce compact representations.

As the volume of sequencing grows, and the redundancy increases, it is not unreasonable to envisage future compression algorithms achieving an order of magnitude better than the 1.6 bits per base of current algorithms, such as the work of Korodi and Tabus (2005), Behzadi and Le Fessant (2005), Manzini and Rastero (2004), Cheng et al. (2003) and Chen et al. (2002a, 2001, 1999). Indeed, GenBank Release 161 contains about 12×10^9 bases of data derived from the human genome[1], even though the human genome is only about 3×10^9 bases in length. The question is no longer whether such redundancy exists in GenBank, but rather in how to make effective use of the redundancy.

Document Reordering (Reassignment Of Ordinal Document Numbers)

Reordering of documents in a database, or equivalently, the reassignment of ordinal document numbers, has been demonstrated to increase the spatially local redundancy, i.e., clustering, available when compressing inverted files (Blanco and Barreiro 2005, Silvestri et al. 2004a,b, Shieh et al. 2003, Blandford and Blelloch 2002). This approach has been shown to yield reductions in compressed inverted list sizes of 20% – 30%, by promoting clustering of similar content (Blanco and Barreiro 2005, Silvestri et al. 2004b,a, Shieh et al. 2003, Blandford and Blelloch 2002).

While individual DNA sequences are notoriously difficult to compress, groups of similar DNA sequences compress much better (Williams and Zobel 1997b). For example when compressing a collection of genomes, compression should be improved by collating sequences by similarity, rather than by organism. This is the application of document reordering to DNA compression. The challenge is the efficient reordering of the large databases involved.

Recently, work has been done that demonstrates that document reordering of large databases is possible in linear time and space (Silvestri et al. 2004b), suggesting that

[1] ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt, [On line; accessed 20 September, 2007]

the approach may become more popular in the future. The feasibility of document reordering of biological databases is demonstrated by the regular release of sorted UniGene databases (Pontius et al. 2003), such as the Human UniGene (nucleic acid) database, and more recently by the publication of an O(n) algorithm for sorting genomic databases (Bernstein and Cameron 2006).

For the purposes of the compression algorithm introduced in this chapter, the assumption is made that a database is already sorted, and exhibits a useful degree of local redundancy.

Improving the compression of sorted, partially-redundant databases, such as the Human UniGene (nucleic acid) database, is particularly relevant to the problem of storing large nucleic acid databases such as GenBank, since much of the data in GenBank is at least partially redundant, e.g., the Expressed Sequence Tags (ESTs) and shotgun sequencing data. In fact, by focusing on the transcriptome of only one organism, the redundancy of the whole of the GenBank database is somewhat under-estimated, as complete duplication of sequences, and inter-organism similarities are not considered. Indeed, as previously mentioned, GenBank contains four times as many bases from the human genome as there are bases in the human genome. Therefore a sorted version of the GenBank database should compress much more effectively than the results of this chapter suggest. However, while such a sorting exercise is now practical (Bernstein and Cameron 2006), it is beyond the scope of this thesis.

Aims

The intention of this chapter is to produce a prototypical algorithm, NP3 (Gardner-Stephen and Knowles 2006, and US Patent Application 60/787,028), that can harness the inter-sequence redundancy that is present now in databases that are already sorted, such as the UniGene builds. The NP3 algorithm is a prototype in that it: (a) as-

sumes the database is already sorted, and; (b) the ad hoc coding methods it uses are neither optimal nor general.

The two goals of the NP3 algorithm are: (a) to demonstrate superior compression of a sorted database, and; (b) to create a compression algorithm that meets the functional requirements necessary to test the thesis of this dissertation. To accomplish the second goal, a compression algorithm is required that can be used in the context of cooperative compression and searching of nucleic acid and protein databases and indices. Therefore it should explicitly encode *recurrence records*, i.e., references to recurrent strings to allow the index of the database to take advantage of the recurrent strings discovered during compression, and provide rapid random access to individual database records to make searching as fast as possible.

The remainder of this chapter: (a) explores the design considerations in creating the NP3 compression scheme; (b) defines the NP3 nucleotide compression algorithm; (c) presents the compression performance of NP3 with the sorted Human UniGene (nucleic acid) database, and; (d) presents the compression performance of NP3 with the de facto corpus of nucleotide sequences traditionally used to compare DNA compression algorithms. In Chapter 7, the NP3 algorithm is combined with the DASH algorithm and the NIX index format introduced there, in order to test the thesis of this dissertation in a more challenging context than that of Chapter 5 where duplicate and redundant database records were present.

6.1 Design Considerations

Assuming that a database contains sufficient redundancy to support effective compression, a coding scheme must be selected. In view of the aims of this chapter, there are a number of issues that must be addressed. These arise because the compressed data will be used in a sequence search and alignment algorithm, and in the context of cooperative compression

of databases and indices. Three prominent issues are: (a) compression and decompression speed; (b) explicit access to recurrence information, and; (c) random access to database records. In this regard, NP3 is similar to the CINO system described by Williams and Zobel (1997b), and attempts to make a compromise between what the authors describe as *vertical compression* (compression of clusters of similar records or sequences), and the principle that records be independently decodable.

6.1.1 Compression And Decompression Speed

NP3 is a *distribution algorithm*, in that NP3 compressed files are expected to be decompressed many times, but compressed rarely. Because compression is performed only rarely, it can be afforded the luxury of a large time budget. However, to support efficient searching, decompression must be rapid. In this regard, what is desired is an asymmetric compression algorithm that maximises the compression factor, without sacrificing decompression speed.

Statistics from four hundred queries performed using the DASH algorithm of Chapter 4 indicate that, for any given search, it is common to access between 1% and 25% of the records in the database. Therefore, for a DNA compression algorithm to be employed without significant time overhead, it must be capable of extracting between tens of thousands and millions of database records in the 1 – 10 seconds DASH requires to perform each search. This precludes compression algorithms based on Arithmetic Coding and Statistical Modelling, as they would decompress too slowly.

6.1.2 Opaque Block Compression Unsuitable

There are a number of fast compression algorithms that could be used to compress blocks of database records, e.g., GZIP (Gailly 1993), or LZO (Oberhumer 1997). Using existing algorithms has the attraction that it would be trivial to implement, due to the availability

of existing programming libraries. Using existing libraries would also make it simple to compare different compression libraries, without modifying the NP3 program. However, the approach of using an existing compression library has four difficulties: (a) the lack of explicit recurrence records; (b) the boundaries of clusters of similar database records may not be known; (c) the requirement for random access to database records, and; (d) the poor performance of general purpose compression algorithms on DNA sequences.

6.1.2.1 The Lack Of Explicit Recurrence Records

Regardless of the particular compression library used, recurrence records would be not encoded in a format that NP3 could quickly and easily parse. Either (a) additional time must be spent parsing this information from the compressed data stream, if it is possible at all; or (b) the recurrence information must be explicitly recorded along side the compressed data. Both options are undesirable as they introduce additional time and/or space costs.

6.1.2.2 The Boundaries Of Clusters Of Similar Database Records May Not Be Known

In order to obtain the best compression, each cluster of similar database records should be compressed in a single block. That is, compression blocks should be aligned to cluster boundaries. However, there is a difficulty: a sorted database does not necessarily contain information on cluster boundaries. Further, there may be gainful compression to be made between clusters. This is especially true if clusters of similar records are in turn sorted to form super-clusters. This makes it difficult to select the block size that maximises compression performance, while maintaining acceptable random access speed.

6.1.2.3 The Requirement For Random Access To Database Records

Effective use of block compression is further complicated by the need for fast random access. This suggests that synchronisation points should exist between each database record, so that single records can be quickly extracted. This is in direct opposition to the maximisation of compression performance, as most compression libraries require blocks much larger than the several hundred characters of a typical nucleic acid or protein sequence.

6.1.2.4 The Poor Performance Of General Purpose Compression Algorithms On DNA

General purpose compression algorithms are generally unable to compress DNA data at or below entropy, except where sequences are clustered.

6.1.3 Existing DNA Compression Schemes Unsuitable

At the time that this work was performed, no suitable DNA specific compression algorithm was available. Generally, the algorithms: (a) used statistical prediction methods (and thus lack explicit recurrence records), and; (b) were relatively slow to decompress. Since then, a DNA compression algorithm has emerged that partly addresses these failings. That algorithm is GeNML (Korodi and Tabus 2005), and its use in the context of cooperative compression and sequence search and alignment is considered in Chapter 7.

Finally, it would have been possible to use an existing explicit recurrence encoding algorithm that supports synchronisation points, such as XRAY (Cannane and Williams 2002), however that algorithm was not known to this author when the work was carried out. Also, XRAY requires a collection-wide memory-resident model. While this may not have posed a problem for the collections tested in this dissertation, it is not unreasonable to expect the model to have difficulties with very large collections, e.g., GenBank, due to either the

model growing too large to fit in memory, or not obtaining sufficient compression over the whole collection due to local variations in composition. Also, while it is possible that XRAY or another method could be hand tuned to offer equal or better compression than NP3, the NP3 algorithm presented here offers faster decompression than XRAY (faster than gzip, versus slower than gzip). This is significant, because as will be shown in Chapter 7, the decompression of sequence bodies is the most time consuming component of the search process described there.

6.1.4 DNA Specific LZ77 Compression Suitable

In contrast to block compression using a general purpose compression algorithm, a DNA specific LZ77 (Ziv and Lempel 1977) derived algorithm has the potential to answer the demands of NP3, namely that: (a) the recurrence records be explicitly encoded; (b) the boundaries of clusters need not be known, and; (c) random access to database records be fast.

6.1.4.1 Explicit Recurrence Records

The LZ77 algorithm explicitly records the location and extent of references to recurrent strings, i.e., recurrence records, making it easy to parse the recurrence records from the compressed data stream.

6.1.4.2 Boundaries Of Clusters Need Not Be Known

The LZ77 algorithm searches for recurrent strings in a sliding window. This means that it can discover any recurrence between the record being compressed and the records in the window. Thus, there is no requirement for cluster information to be explicit in the database. The tension between (1) fast random access and (2) improving compression by sourcing recurrences from other database records is discussed later.

6.1.4.3 Provision Of Fast Random Access To Database Records

Rapid LZ77 implementations have been devised in the past, e.g., by Williams (1991) and Oberhumer (1997). One technique that they use to accelerate the already rapid decompression of LZ77 algorithms is byte-aligned codes, which avoid the computational cost of decoding variable bit-length tokens.

A byte-aligned implementation of the LZ77 algorithm also offers byte-aligned synchronisation points. The only prerequisite is that every database record boundary corresponds with a fresh code. Therefore synchronisation points are obtained in return for little further compression leakage. However, there is one remaining complication with providing per-record synchronisation points; because recurrent strings may be sourced from another record, access to other records may be required. Therefore the encoding of recurrence records is now the subject of further discussion.

6.1.5 Encoding Recurrence Records.

In opposition to the need for random access, is the requirement to explicitly encode recurrences of identical strings, not only within, but also between records. Such inter-record references complicate the decompression of randomly selected records, reducing decompression speed. This is because one recurrence record may point to another recurrence record. That recurrence record may point to yet another, and so on. In this way *recurrence chains* may form. Long recurrence chains are a problem, because it is necessary to extract the records along the length of the chain in order to extract the record at the head of the chain. Linear decompression is not affected, because the records in the chain will have already been decompressed when each successive record is retrieved, however decompression of random records suffers.

Inefficient random-order decompression of records caused by recurrence chains can be partially addressed by making use of: (a) the realisation that record access in sequence search and alignment is not really random, and; (b) the division of the database into partitions.

Recurrence chains do not necessarily introduce any overhead when searching a sorted database. This is because the so-called random access that is required for sequence search and alignment is, in practice, the retrieval of records that are similar to a given query. Therefore, the ordering of the sorted database by record similarity increases the locality of access. That is, if one record from a cluster of similar sequences is retrieved in response to a query, then it is likely that the other records in that cluster will also be recalled. Therefore, the decompression overhead of allowing nearby records to reference one another should be relatively small, since if one is required, most of the others are likely to be required too.

However, if references occur between clusters, then unnecessary extractions may occur. This behaviour should be self limiting, however, as the greater the similarity between two records from neighbouring clusters, the greater the likelihood that both will be required to service any query that requires either of them. Conversely, dissimilar records are unlikely to reference one another.

Finally, partitioning the compressed database limits the maximum length of recurrence chains to the number of records in each partition. Such cleaving of the database into manageable sized portions has two other benefits: First, DASH assumes and requires that databases are partitioned, and, second, it becomes feasible to cache all extracted records from a given partition, thus avoiding the possibility of inefficiency due to repeated extraction of records, and any recurrence chains that are involved. Caching decompressed partitions in this way also allows the decompression time to be amortised among queries processed as a batch.

Thus, while the length of recurrence chains should be monitored to ensure satisfactory random access performance, the methodology of encoding each successive record using a

byte oriented LZ77 variant has the potential to be both effective and reasonably efficient. The NP3 algorithm is thus constructed as a byte oriented LZ77 variant, as described below.

6.2 The NP3 Algorithm

The NP3 algorithm converts a nucleic acid database (in FASTA format) into a number of concatenated database partitions, each consisting of three main parts. These three parts are:

1. Administrative information;
2. Compressed record descriptions; and
3. A series of code streams describing how to assemble each record of the database from new and previously seen strings, some of which may come from other records.

Figure 6.1 illustrates the general data flow of the NP3 algorithm. The solid and dashed lines describe the compression and decompression data flows, respectively. The asymmetric time relationship between compression and decompression is reflected in the relative complexity of their data flows.

The compression data flow shows how the FASTA format database is initially parsed, and separated into record description and record body streams that are handled separately. The remaining processes are elucidated in the corresponding text below, that describes how the compressed representations of the description and body data are generated, collated and partitioned to produce an NP3 file.

6.2.1 Administrative Information

An NP3 file begins with a header that identifies the file as a valid NP3 file, and contains summary information. This includes the total number of records and letters stored, and a

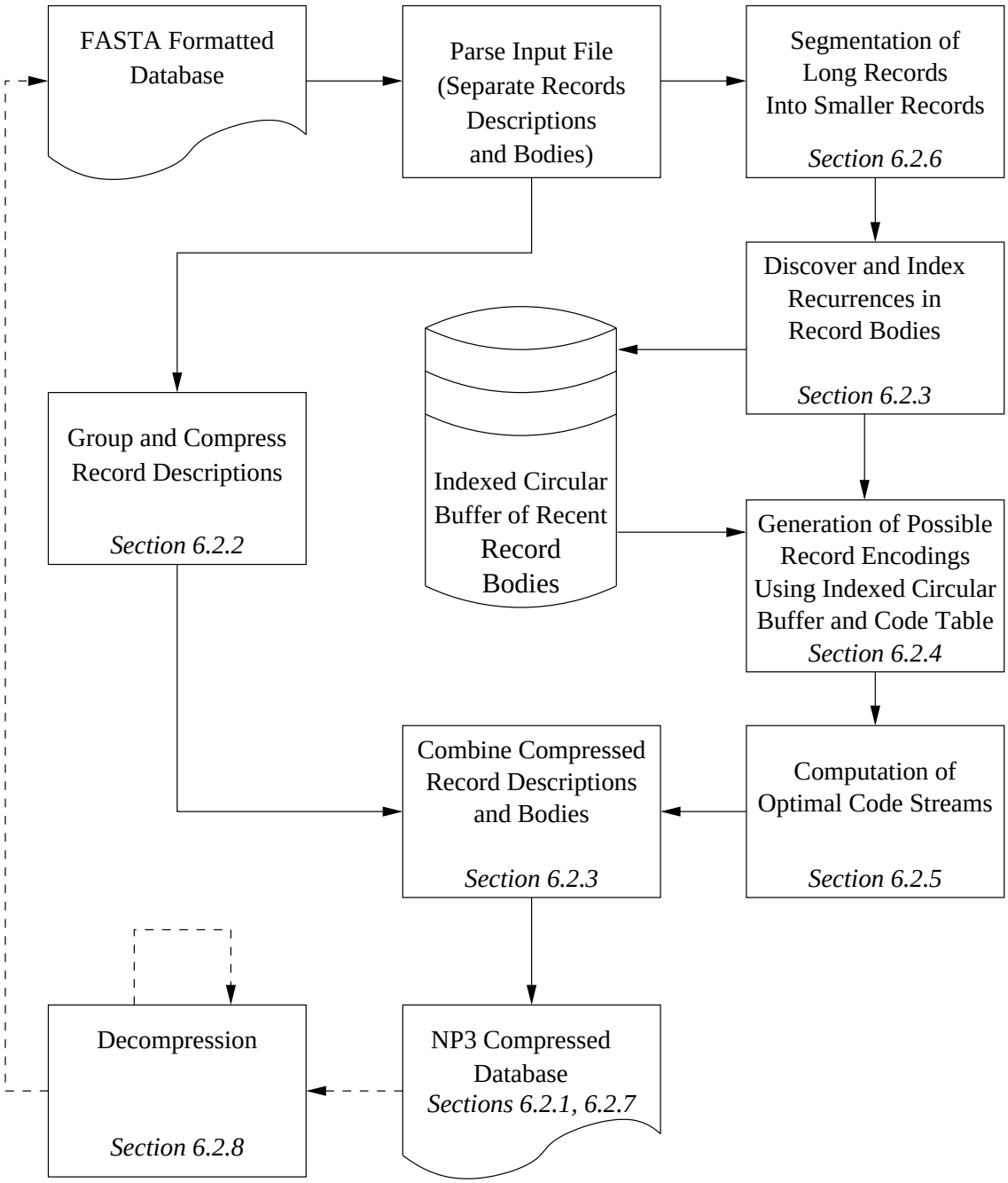

Figure 6.1: NP3 Flow Chart.

pointer to the first database partition. Each partition is also prefixed with a header containing the summary information for that section, as well as pointers to the various parts of that section, principally the code stream and record description look up tables that facilitate random access to the compressed records.

While it would have been be possible to compress the tables of addresses for the per-record synchronisation points, e.g., according to Bell et al. (1993), there was little motivation to compress them, as they comprised only 2% of the size of the compressed database. Therefore they were stored uncompressed.

Finally, all information in an NP3 file is stored in a consistent byte order to provide file compatibility between CPU architectures.

6.2.2 Compression Of Sequence Descriptions

The major arguments against opaque block compression are not compelling when compressing sequence descriptions, because: (1) recurrence records are not required, because the sequence descriptions will not be indexed (and therefore will not be cooperatively compressed); (2) sequence descriptions are retrieved for the limited number of results returned by a search, rather than for the possibly millions of records accessed during the processing of the same search, and; (3) general purpose compression algorithms work well on FASTA format sequence descriptions. Therefore, for NP3 an existing general purpose compression library was used when compressing sequence descriptions.

The tension in selecting a compression algorithm and block size for the sequence description data lies in balancing reduction of message length against decompression speed. Compression is performed only once for a database, so compression time is of lesser concern. However, since a goal of NP3 is to develop the next generation database and index format for DASH, the decompression of the sequence descriptions must not weigh heavily on the total run time.

As shown in Chapter 5, the DASH M2 algorithm typically takes around 1 second on a modern computer to search the Human UniGene (nucleic acid) database. Typical search parameters are to return alignments against 200 records. If we are to assume that the Human UniGene (nucleic acid) is a typical sized database, and if description retrieval is to contribute $\leq 50\%$ of run time, then the sequence descriptions must be extracted in ≤ 0.5 second. This allows 2.5 milli-seconds per sequence description.

The BZIP2 (Seward et al. 2001) algorithm was selected as it simultaneously offers among the best combined compression performance and decompression speed (Witten et al. 1999). Compressing the 470 MB of description data from the approximately 2 giga-base Human UniGene (nucleic acid) database with BZIP2 reduced the size to 76 MB, i.e., a compression factor of about 6. Decompression requires 50 CPU seconds on an AMD Opteron 244 (1.8 GHz) processor, equating to $\simeq 10MB/sec$. Therefore $0.5sec \times 10MB/sec = 5MB$ of descriptions can be decompressed in the 0.5 second time budget.

If $470MB$ of sequence description data per giga-base is representative of the typical volume of sequence description text in a biological sequence database, then no more than $5MB/470MB = 1.06\%$ of all description data can be decompressed in the time budget. This dictates a method of storing each sequence description either separately, or in relatively small blocks. The former is difficult when using existing compression libraries, while still aiming for optimal compression ratios, and therefore the blocking approach was chosen.

Taking the block based approach, and assuming that the search results consists of the 200 best matching records, and further assuming that the description of each of these records occurs in a different block, then 200 blocks must constitute no more than 1% of the total number of blocks. This gives a minimum of about 200 $\div$ 1% = 20,000 blocks, or a mean uncompressed block size of $470MB \div 20,000\,blocks = 23.5KB/block$.

However, the selection of a compression block size influences not only the decompression speed, but also the degree of compression that can be expected (Bell et al. 1993). Smaller blocks will provide less opportunity for compression. Indeed, 32 KB blocks were observed to yield compression factors of 4 to 5, versus the 6 observed for BZIP2's maximum block size of 900 KB.

It was observed that most sequence descriptions end with a `/len=`*n* tag. These tags added to the entropy of the description data, as the sequence length was not predictable from record to record. However, from a holistic perspective, the tags add almost zero entropy, since practically every record included such a suffix tag, and the precise content of the tag could be deterministically computed for each record. Therefore, the NP3 program trims these tags from descriptions before blocking and compressing them. The tags are recreated during decompression. Trimming these tags improved the compression factor of 32 KB blocks from the 4 – 5 range, to 5 – 5.5, and consequentially this method was adopted.

After BZIP2, the next most attractive algorithm for compressing sequence descriptions was GZIP, because it decompresses substantially faster than BZIP2, and with only moderately worse compression performance. The faster decompression of GZIP would allow larger block sizes, while remaining within the time budget, and so the overall margin of compression performance between GZIP and BZIP2 would be reduced, although not eliminated.

However, there is good reason to suggest that the time cost analysis for decompressing sequence descriptions using BZIP2 is pessimistic. The principal reason is that the database being compressed is sorted. Therefore, as mentioned previously, records are likely to be accessed in clusters. Thus, the total number of blocks required for extraction is likely to be much less than the total number of sequences included in the search results, as similar sequences are likely to be near one another in the database, and consequentially to have their descriptions stored in the same block. Therefore, the effective time cost for decompres-

sion of sequence descriptions is likely to be substantially reduced, making the improved compression of BZIP2 affordable in terms of time cost.

Finally, as will be shown later, even when compressed with BZIP2, the compressed sequence descriptions of the Human UniGene (nucleic acid) database were about one half the size of the compressed record bodies. Thus, reducing the size of the compressed sequence descriptions contributes significantly to the overall compactness of an NP3 compressed database. Thus using BZIP2 to maximise the compression factor of the sequence descriptions is worthwhile, despite any residual time cost it introduces.

6.2.3 Discovery Of Recurrences

6.2.3.1 Recurrence Search Algorithms

The problem of discovering the longest recent occurrence of the next few symbols of a string has been well explored. Bell and Kulp (1993) surveyed a number of approaches that were known at the time. Their conclusion was that, especially for larger window sizes, the `list2` method was the fastest for English text. It is contraindicated only when the text being compressed is highly skewed as, for example, in bi-level bitmap images. This presents no problem for nucleotide sequences, which have approximately equal representation of each base.

The `list2` method uses a circular buffer to store the sliding window. A double linked list is maintained for each of the q^2 bi-grams, where q is the alphabet size. That is, a list is maintained of the address of each occurrence of each bi-gram that occurs in the window.

Aside from the problems with skewed data, the only other reservation voiced by the authors about the `list2` method was the excessive memory requirements of $2q^2 + N$ pointers and integers, where N is the window size. However, memory has become relatively cheap and plentiful, and for nucleotide strings where $q = 4$, the memory cost of a window size of

$N = 10^6$, is inconsequential — requiring only $2 \times 4^2 + 10^6 \simeq 10^6$ pointers and integers. Advantage can be taken of the small nucleotide alphabet by using k-grams instead of bi-grams, to produce a method that can be called `list`k. Choosing $k = 10$ reduces the length of each list that must be searched by a factor of $\frac{q^k}{q^2} = \frac{4^{10}}{4^2} = 65,536$, while only modestly increasing memory requirements to $2 \times q^k + N = 2 \times 4^{10} + 10^6 = 2^{21} + 10^6 \simeq 3 \times 10^6$ pointers and integers.

A different approach that has become increasingly attractive since Bell and Kulp's 1993 survey, is the use of suffix structures, such as suffix-trees and suffix-arrays. This is both because memory is more plentiful, and also because the previously difficult problem of deletion of nodes in a suffix-tree has been solved by Larsson (1999), allowing suffix structures to be efficient in a sliding window context.

However, Sadakane and Imai (2000) discovered that, at least in the case of suffix arrays, suffix structures are unlikely to be faster than the `list`k approach at the window sizes that are envisaged for NP3, i.e., 10^5 – 10^6 letters. So, while there is increasing interest in using suffix structures to search DNA data, e.g., Cheng et al. (2003), Hunt et al. (2002), the `list`k approach is unlikely to be far from the state of the art in terms of speed, and certainly fast enough for the asymmetric compression required in the current application.

6.2.3.2 Discovery Of Recurrences

Recurrent strings are discovered by using the `list`k method (with $k = 10$) to search for matching regions in a window of sequence data spanning several hundred records or several hundred kilo bases, which ever is the lesser.

Consider Figure 6.2, where three records share common strings, as indicated by the striped regions. In this case, Record #2 contains a region that also occurs in Record #1, as indicated by arrow (a). Similarly, Record #3 contains a portion of that same region, which occurs in both Record #1 and Record #2 (arrows (b) and (c)). Where such multi-way recurrences

occur, they are linked to the nearest record. In this example, this means that the recurrence of Record #3 gets recorded against Record #2, and that the recurrence of Record #2 against Record #1, thus forming a short recurrence chain.

While minimising the distance of reference, this approach also maximises the number of links in the chain required to eventually find the original instance of the recurrence. This aids cooperative compression of the index by making the maximum number of recurrence records available. Nearer references also imply shorter average reference distances to encode, aiding compression performance. However, because a strictly byte-aligned compression scheme is used in this chapter, only limited advantage can be made of the shorter codes afforded by reducing the average reference distance.

However, maximising the number of links in recurrence chains works against decompression speed when extracting random records from the database. Fortunately, and as previously described, record access in a sorted database is not truly random, and many or all of the records along a recurrence chain may be required to answer any query that requires any one of the records in the chain. In that case, the overhead is at most marginal, regardless of the length of the recurrence chains, and there is little to be gained by minimising the chain length.

It is also possible to reduce the average recurrence chain length by avoiding the use of inter-record references when they offer no space advantage over alternative codes. However, this reduces the number of recurrence records that are available for cooperative compression. NP3 allows the user to specify whether to activate this heuristic.

This process of searching for recurrences takes time proportional to the amount of material seen to date, up until the window is fully loaded, and a constant time cost to update the memory structure for each base that is inserted into the window. Given that the window size is of the same order of magnitude as the index partition size, this results in an effect computational cost of $O(n^2)$ with respect to window or index partition size.

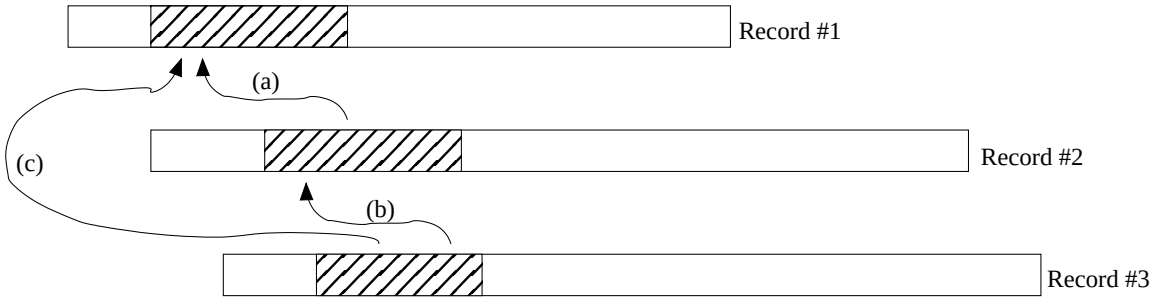

Figure 6.2: Example Of Three Records Each Containing A Common, i.e., Recurrent, Region. (The recurrent region is shaded).

6.2.4 Generation Of Possible Record Encodings

6.2.4.1 Ad Hoc Code Table

Selecting an optimal set of encoding methods for a broad class of inputs is not trivial. It is in no way claimed that this has been achieved in the current NP3 code table. The constraint of using byte-aligned tokens all but rules out the possibility of optimality, and by focusing on a single database, Human UniGene (nucleic acid), it is extremely unlikely that the resulting code will be robust in the face of varied input. Indeed, it gives the resulting algorithm an advantage over any generalised compression scheme. However, what has been created is an ad hoc code table that obtains effective compression on the Human UniGene (nucleic acid) database, is extremely fast to decompress, and encodes recurrences explicitly, thereby meeting the requirements imposed by cooperative compression, and thus allowing the testing of the thesis of this dissertation. As the ad hoc scheme has the required properties to allow the testing of the thesis of this dissertation, the construction of a more general or optimal code is left as a future exercise.

The full coding scheme used by NP3 is presented in Table 6.1 with both the various codes and the Recently Referenced Address Table (RRAT) being explained in the following paragraphs. Bit positions in Table 6.1 are marked as containing either a constant 1 or a 0, address (‘a’), relative address offset (‘r’), number of bases encoded (‘n’), extra bits for

number of bases encoded (‘x’), optional change of case (‘c’) where a 1 indicates toggling upper/lower case, or a base represented in either 2 bit (‘bb’) or 4 bit (‘base’) format.

Codes B, D and E store one, two or three bases, respectively, in a single byte.

Code F is used to encode longer strings, storing four bases per byte, but has a 2 byte overhead. This code is intended for coding long regions of non-compressible sequence, and in that context is relatively efficient, as the 2 byte overhead is amortised over many bases. To encourage rapid decompression, Code F can only be used at offsets within a record that are a multiple of four. This allows direct byte copying for 2 bit format extraction, and the efficient use of look up tables to extract in either four bit or ASCII formats.

Codes H and I are used to encode repeating letters and bi-grams, e.g., AAAA or GCGCG. These codes use the 4 bit IUPAC code to describe each repeating base, which allows them to also encode wild cards, such as N or R. The encoding of wild cards in this way obviates the need for a patch table as in the CINO method of Williams and Zobel (1997b), but otherwise serves the same purpose.

In order for Codes H and I to fit in two bytes each, they include relatively few bits (marked with ‘n’ in the table) to specify the number of bases encoded. To allow efficient encoding of longer lengths to be accommodated, Code J is provided to allow the addition of three extra address bits to the length field of these codes. Code J can also be used with Codes M through Q, which are described later.

Codes K and L are similar to Codes H and I, but indicate change of case, and are used to encode bases that appear in opposite case. These codes are only used if preservation of case is requested when constructing an NP3 file.

Code M encodes a reference to a string in a recently processed record, i.e., it encodes a recurrence record. The location of the string is encoded in a 22 bit value, as described in the table, allowing reference to the 512 most recently processed records.

Table 6.1: NP3 Binary Encoding Scheme For Nucleotide Sequence Data.

Code	Binary Format	Bits	Range	Description
B	100000bb	8	1	A single 2 bit encoded base.
D	1001bbbb	8	2	Two 2 bit encoded bases.
E	11bbbbbb	8	3	Three 2 bit encoded bases.
F	010nnnnn nnnnnnnn bb$\times n$	16+ 2n	8192	A series of 2 bit encoded bases. This is how long stretches of non-recurrent sequence are encoded with an efficiency approaching that of direct coding. n must be a multiple of 4.
H	011nnnnn basebase	16	32	A run of identical base pairs, e.g. GCGCG. (4 bit encoding allows the use of all 16 IUPAC codes for nucleotide bases).
I	10001001 basennnn	16	16	A run of up to 16 identical bases (4 bit encoding allows the use of all 16 IUPAC codes for nucleotide bases).
J	10100xxx	8	$\times 8$	The extend token is placed immediately after any other token to add three extra bits to the length fields, e.g., a maximum range of 255 becomes 2303.
K	1000101c 0000base	16	1	A single 4 bit encoded base with an implied case shift before it, and an optional case shift after (4 bit encoding allows the use of all 16 IUPAC codes for nucleotide bases).
L	100011cc basebase	16	2	Two 4 bit encoded bases with an implied case shift before the first, and optionally after each base.
M	00aaaaaa aaaaaaaa aaaaaaaa nnnnnnnn	32	256	Recurrence with address encoded as a 22 bit value. The lower 13 bits are the offset in a record, and the upper bits encode the record number of that record relative to the current record. Therefore it is possible to address up to 512 recent records of length 8,192 bases each.
N	100001rr nnnnnnnn	16	256	Recurrence with address encoded as a 2 bit value relative to the previously accessed address. Useful for encoding regions with short insertions and deletions (indels).
O	1011rrrr rrrrrrrr nnnnnnnn	24	256	Recurrence with address encoded by the last sequence address used (which might be in a different record), plus 12 bit offset.
P	101010aa nnnnnnnn	16	256	Recurrence with address encoded by the specified one of four entries from the recently referenced address table (RRAT), which might point to a different record.
Q	101011aa rrrrrrrr nnnnnnnn	24	256	Recurrence with address encoded by the specified one of four entries from the RRAT, which might point to a different record, plus an 8 bit relative offset (rrrrrrrr).

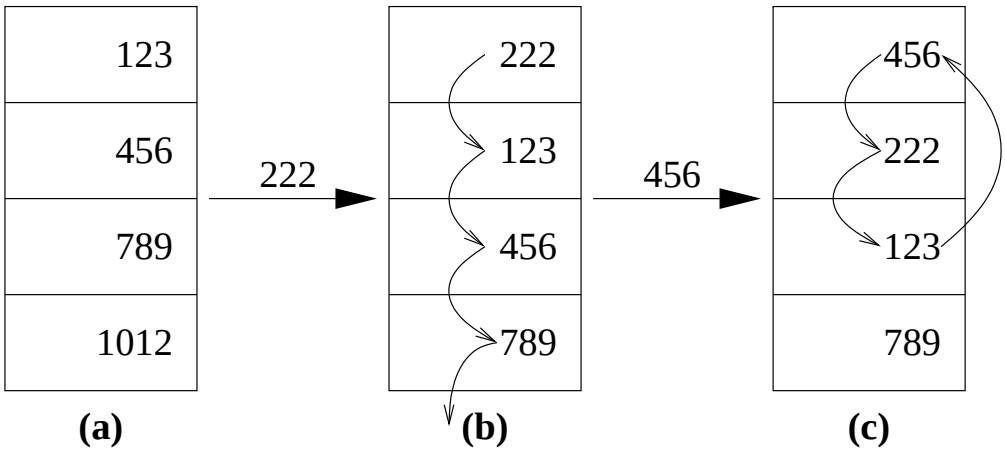

Figure 6.3: Recently Referenced Address Table (RRAT) Management.

6.2.4.2 Recently Referenced Address Table

Because of the clustering that is assumed to exist in the database, it is possible that when a record references multiple recurrent strings, that the references will be to strings in records that are near one another, and that multiple references may be made to a single record. To take advantage of this, a table is maintained of the addresses of recently referenced strings, the *Recently Referenced Address Table* (RRAT). This table contains the addresses of the four most recently referenced strings. Each time a string is referenced, the address of the string is inserted in the first entry of the table, and all other entries are moved down one position, causing the fourth entry to be discarded. However, if the address is already in the table, then it is promoted to the first entry, and the intervening entries are moved down.

Management of RRAT entries is illustrated by example in Figure 6.3. At (a), the table contains the four addresses 123, 456, 789 and 1012. Then the string at address 222 is referenced. Because it is not already in the table, the other entries are shuffled down, causing address 1012 to be discarded; address 222 is then stored in the first entry, as depicted in (b). Finally, in (c), address 456 is referenced. Because it is already in the table, only the entries above it are shuffled down to make room; no address is discarded.

Returning to Table 6.1, Codes N and O use the RRAT to encode references to recurrent strings, i.e., recurrence records, using less than four bytes. They gain their brevity by supplying only a relative address, which is combined with an address from the RRAT to obtain the complete address. Code N, with its two byte format, provides the lowest cost option for resuming recurrences that are interrupted by short indel events, i.e., insertion and deletions, while Code O's larger relative addressing range provides increased utility, at the cost of an additional byte. Codes N and O use only the first entry of the RRAT.

Codes P and Q are similar to Codes N and O, but can use any entry of the RRAT. The trade-off is their reduced relative addressing range. Indeed, Code P has no relative address component, and can therefore be used only when an RRAT entry contains the precise address required. Therefore, the RRAT is managed in order to correctly predict the address of future references as often as possible, and so allow more extensive use of the shorter Codes N through Q. This is best explained with an example.

Consider encoding AAAAAAAACCCGGGTTTTTTTTTT. The strings of A's, C's, G's and T's are used for clarity here, but in practice could be any sequence, and of any length. Assume that the address in the first entry of the RRAT points to a string of 14 consecutive A's followed by 11 consecutive T's, i.e., $\mathtt{A}^{14}\mathtt{T}^{11}$. In that case, the first 8 A's of the example sequence could be encoded in two bytes using either Code N or Code P. The six substitutions, CCCGGG versus AAAAAA, might then be coded using two instances of Code E to encode CCC and GGG, respectively.

Considering the contents of the RRAT, if the address of recurrent strings, once entered in the RRAT, are fixed, then after each code is output, the first RRAT entry will continue to point to the *beginning* of the $\mathtt{A}^{14}\mathtt{T}^{11}$ string, as depicted by arrow (a) in Figure 6.4. This is a problem when seeking to efficiently encode the remaining string of $\mathtt{T}^{11}$: The distance between the beginning of $\mathtt{A}^{14}\mathtt{T}^{11}$ and its $\mathtt{T}^{11}$ tail, which could be used to encode the remaining string as a recurrence, is 14 bases. Therefore, four bits are required to encode this relative

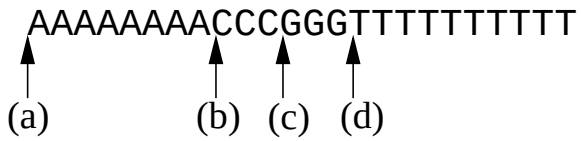

Figure 6.4: Comparison Of Different RRAT Advancement Strategies.

to the entry in the RRAT. This rules out the short Codes N and P, thereby necessitating the extra byte entailed by Code O or Q.

However, if the RRAT entries are set to the address of the symbol following the referenced string, it would point to the first C in the example sequence, indicated by arrow (b) in Figure 6.4, and the distance would be only six bases. Unfortunately, this does not redeem the current example, because the magnitude of the distance remains too large to be encoded using either of the shortest codes, Code N or Code P.

The current example can be saved by advancing RRAT entries for *every* residue that is encoded. Assuming the use of Code N to encode the first eight bases, followed by Code E $\times 2$ to encode CCCGGG, the entry would begin pointing to the start of the $\mathtt{A}^{14}\mathtt{T}^{11}$ string, indicated by arrow (a) in Figure 6.4, then be advanced by eight bases to the position indicated by arrow (b), then by a further three bases to the position indicated by arrow (c) when CCC is encoded, and then by three more bases to the position indicated by arrow (d) when GGG is encoded, bringing it to the first T of the $\mathtt{T}^{14}$ string: The address in the RRAT now points to exactly the right address, and Code N or Code P could now be used to encode $\mathtt{T}^{11}$ in two bytes.

By introducing this rule of advancing the RRAT entries, the distance has been reduced to zero — regardless of the length of the substitution — allowing the use of the shorter Codes N and P. This creates a form of approximate repeat matching, similar to the techniques used in other nucleotide compression algorithms. However, the efficiency is limited because the codes are byte-aligned.

6.2.4.3 Selection Of Codes During Compression

There are often multiple codes that can be used to encode a particular string. In the previous example, Code E was used to encode `CCC` and `GGG`. However, Codes B, D, F, H or I could have been used instead. The set of valid codes at each successive offset in the record is determined using the code table and indexed circular buffer of recently processed records.

Figure 6.5 illustrates an example of the set of possible codes at several offsets in a sample sequence. Assume for this example that the first entry in the RRAT contains the address of the string `TTAAAAAAAAG`.

At offset zero, it is possible to use either the long or short forms of direct coding, using Codes B, D, or E, to encode, respectively, `A`, `AC` or `ACA` in one byte. Because the offset is a multiple of four, Code F can be used to encode either four, eight, twelve or all sixteen bases in 3, 4, 5 or 6 bytes, respectively. It is also possible to use Run Length Encoding (Codes H and I).

At offset one, long direct encoding (Code F) is not possible because the offset is not aligned to a multiple of four. However, Codes B, D, E, H and I remain valid.

At offset two, Codes B through E and Codes H and I continue to be valid options. Moreover, Codes M through Q are now possible, because the string `AAAAAAAAG` can be encoded as a recurrence of a previously encountered string. Because the first entry of the RRAT points to the precise address of the recurrence (remember that addresses in the RRAT will have been advanced two bases, skipping the `TT` prefix), the shorter codes are all possible.

6.2.5 Computation Of Optimal Code Streams

The optimisation process operates by processing the set of encoding possibilities at each offset of a record, as described in the preceding text. The reach of these possibilities, i.e., how many bases can be encoded at each offset using the available mechanisms, are

Input Sequence	**ACAAAAAAAAGTCGTG**	
Offset Zero	**A**	Code B; 1 byte
	AC	Code D; 1 byte
	ACA	Code E; 1 byte
	ACAA	Code F; 3 bytes
	ACAAAAAA	Code F; 4 bytes
	ACAAAAAAAAGT	Code F; 5 bytes
	ACAAAAAAAAGTCGTG	Code F; 6 bytes
	A	Code H; 2 bytes
	ACA	Code I; 2 bytes
Offset One	**C**	Code B; 1 byte
	CA	Code D; 1 byte
	CAA	Code E; 1 byte
	C	Code H; 2 bytes
	CA	Code I; 2 bytes
Offset Two	**A**	Code B; 1 byte
	AA	Code D; 1 byte
	AAA	Code E; 1 byte
	AAAAAAAA	Code H; 2 bytes
	AAAAAAAA	Code I; 2 bytes
	AAAAAAAAG	Code M; 4 bytes
	AAAAAAAAG	Code N; 2 bytes
	AAAAAAAAG	Code O; 3 bytes
	AAAAAAAAG	Code P; 2 bytes
	AAAAAAAAG	Code Q; 3 bytes

Figure 6.5: Coding Options For Three Successive Offsets In An Example Sequence.

considered in conjunction with the byte cost of each. By way of example, consider the sequence ACAAAAAAAAGTCGTG.

6.2.5.1 Tabulation Of Coding Options

Using the coding possibilities of Table 6.1, the valid point to point transitions and their costs are tabulated in Tables 6.2 and 6.3, respectively. The y-axis represents the start offset in the record, and the x-axis the end offset, where the end offset refers to the first base immediately after the base(s) encoded by the Code indicated in Table 6.2. The blank spaces in the tables indicate record positions that cannot be reached with a single code from the respective initial record position.

The diagonal band of cost 1 byte arises from the direct coding method that can record up to three bases. Recall that the direct coding scheme for longer strings can only be employed between points that are aligned to four base boundaries, and so in most cases this is not an option. Indeed, at most offsets there are relatively few possibilities. In all situations, the shortest code that can make a given transition is always used in preference to all others. Thus Code E (1 byte) was recorded for the transition from offset 3 to offset 6, even though Code H (2 bytes) or Code I (2 bytes) were also possible.

6.2.5.2 Calculation Of Optimal Path

Upon calculation of the cost of each possible code, the cumulative cost of coding from the beginning to the end of the record can be computed, as in Table 6.4. Global dynamic programming is used to choose the path with optimal (minimum) cost. For example, for the route from position 0 to 4, and 4 to 5, with a cumulative cost of $3+1=4$ bytes, would be set aside in favour of the lower cost route from 0 to 3, and 3 to 5, at a cost of only $1+1=2$ bytes.

Table 6.2: Codes Corresponding To The Coding Costs In Table 6.3.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	B	D	E	F				F				F				F
1		B	D	E												
2			B	D	E	I	I	I	I	I						
3				B	D	E	I	I	I	I						
4					B	D	E	I	I	I		F				F
5						B	D	E	I	I						
6							B	D	E	I						
7								B	D	E						
8									B	D	E	F				F
9										B	D	E				
10											B	D	E			
11												B	D	E		
12													B	D	E	F
13														B	D	E
14															B	D
15																B
16																

Table 6.3: Matrix of Coding Costs (in bytes). The left hand column is the start offset, and the top row the final offset. Each row represents the cost of reaching each possible destination point using a single code from that offset.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	1	1	3				4				5				6
1		1	1	1												
2			1	1	1	2	2	2	2	2						
3				1	1	1	2	2	2	2						
4					1	1	1	2	2	2		4				5
5						1	1	1	2	2						
6							1	1	1	2						
7								1	1	1						
8									1	1	1	3				4
9										1	1	1				
10											1	1	1			
11												1	1	1		
12													1	1	1	3
13														1	1	1
14															1	1
15																1
16																

Table 6.4: Matrix of Cumulative Coding Costs (in bytes) from the beginning (offset=0) to any final offset, using the cheapest string of available codes.
The left hand column is the start offset, and the top row the final offset. The cheapest path from the start of the record to a given offset is found by determining which cell in the column corresponding to that offset has the lowest score. The underlined cells represent the cheapest encoding for the entire record, as determined by using this method from the end of the record and working backwards.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	1	1	3				4				5				6
1		2	2	2												
2			2	2	2											
3				2	2	2	3	3	3	3						
4					3	3	3	4	4	4		6				7
5						3	3	3	4	4						
6							3	3	3	4						
7								4	4	4						
8									4	4	4	6				7
9										4	4	4				
10											4	4	4			
11												5	5	5		
12													5	5	5	7
13														5	5	5
14															6	6
15																6
16																

The dynamic programming procedure finds an optimal encoding by back-tracking from the cheapest cost at the end of the record. In the case of this example, and referring to Table 6.4, back-tracking from the end of the record (column 16) would begin with the entry in row 13 (cost 5 bytes). By consulting Table 6.2, it is found that Code E was used. The starting point of this transition (offset 13) is now used as the end point for the next iteration: This process is repeated recursively, visiting offsets 10 and 3, before reaching the start of the record. This path is indicated by the underlined values in Table 6.4. This path is then reversed to give the forward path, 0 to 3 to 10 to 13 to 16, which can then be translated to find the codes used by the encoding scheme and the final representation of the record (Table 6.5).

Table 6.5: The Optimum Code For The Record In Figure 6.5.

Transition	0→3	3→10	10→13	13→16
Cost (bytes)	1	2	1	1
Codes Used (Tables 6.1, 6.2)	E	I	E	E
Binary Codes	11000100	10101001 00000111	11101101	11101110

6.2.5.3 Effect Of Extension Code (Code J)

The use of the length extension code, Code J, that varies the effective length of a code, depending on the number of bases being encoded, helps reduce the encoded record length. However, this complicates the task of the optimiser, because it effectively creates multiple versions of each of the original codes, with differing byte costs and maximum reaches. For each code, the optimiser must know where the break points are, i.e., the lengths where an extend coded is required. Each transition that spans any of these break points must be considered as a series of independent coding options in the dynamic programming grid, where each extend code creates another break point. For example, a run length encoding code requires 2 bytes for up to 16 bases, or when combined with an extend code, a run of between 17 and 127 bases can be encoded, but at the cost of an additional third byte. For example, a run of 40 identical bases would produce two run length encoding possibilities: one for 40 bases (with the extend code, and costing 3 bytes) and one for 16 bases (with no extend code, and costing only 2 bytes).

6.2.5.4 Effect Of RRAT

There is one other difficulty faced by the optimiser, and that is Codes N through Q that make use of the RRAT and relative addressing. Because dynamic programming optimisation works backwards from the end, the optimiser sees the addresses in reverse order.

While the magnitude of the distances between successive references are not modified by this inversion, the contents of the RRAT can only be guessed. As a result, the optimiser can make incorrect judgements as to which code is best in any given situation, and so the length of the final bit stream may differ from the predicted length. This non-optimal behaviour is reflected in the slightly different file sizes that result when the policy changes between preferring inter-record references and avoiding inter-record references, even though the situation should be cost neutral.

6.2.5.5 Computational Cost

Because the stream generation process utilises a dynamic programming optimisation process that can be represented as a two dimensional dynamic programming space, the computational cost is $O(n^2)$ with respect to the input record length. Partly because of this, the input record length is limited to a few KB as described in Sub-Section 6.2.6, thus the cost is effectively constrained, and is generally much smaller than the cost of the preceding recurrence discovery step.

However, provided that the database is much larger than the partition size, the compression time will be linear with respect to the database size due to the approximately constant cost of compressing each fixed-sized partition.

6.2.5.6 Summary

The optimisation framework just described makes it relatively simple to employ a wide variety of coding methods, and obtain the best results that it can afford. This is due to the decoupling of the optimiser from the specific codes and bit patterns; the optimiser is only aware of the cost of each option at a given point. This provides flexibility in later improving the coding scheme, without requiring significant change to the optimiser. The

ability to change the coding back end also makes the NP3 approach readily adaptable to other types of data, e.g., protein or English text.

6.2.6 Segmentation Of Long Records

Long sequences are common in genomic databases. To simplify the recurrence discovery, database partition processes, and to ensure the speed of compression, long sequences are cut into a number of shorter records of $\leq$8,192 bases.

In anticipation of the use of NP3 files in the context of sequence search and alignment, a short overlap is provided between each successive segment to simplify stitching of results that span multiple segments. The overlap can be used to translate alignments between neighbouring sequence segments. This overlap contributes only negligibly to the total file size, as it can always be encoded in 2 bytes using Code P, because the RRAT is initialised for each Record such that the first entry contains the address of the overlap.

When sequences are segmented, the description is appended with the byte 0x01, and stored with the first record of the sequence. The `/len=`*n* tag cannot be removed from the description in this situation, as during serial decompression the length of the reassembled sequence cannot be determined before hand.

Subsequent record descriptions consist of a 0x01 byte, followed by the record number of the first segment, and the offset of the current record within the sequence. The non-terminal records also have a 0x01 byte appended to the end of their descriptions, to indicate that more fragments follow. This system is used both within and between database partitions.

The complete procedure is illustrated in Table 6.6, where a list of record descriptions is presented for a hypothetical sequence that is divided into 4 records.

Table 6.6: Example Of Sequence Segmentation Tags In Record Descriptions.
The first record of a segmented sequence contains the true description of the sequence, followed by a 0x01 character to indicate that the sequence has subsequent records. The non-terminal records 2 and 3 also contain 0x01 characters at the end to indicate that the sequence continues. The non-head blocks 2, 3 and 4 are prefixed with 0x01 to indicate that they are fragments, and the description data identifies the head record and the offset of each block in the head record.

Record #	Stored Description
1	“> hypothetical protein foo”(0x01)
2	(0x01)”1:8160”(0x01)
3	(0x01)”1:16320”(0x01)
4	(0x01)“1:24672”

6.2.7 Database Partitioning

As indicated previously, an NP3 file is produced as a series of partitions. A new partition is commenced whenever the size of the previous partition exceeds a threshold. While this partitioning is a requirement for use with the DASH algorithm, it can also be leveraged to provide two additional benefits: (a) Parallel Construction of NP3 files, and; (b) Ease of appending and updating of NP3 files.

6.2.7.1 Parallel Compression Of NP3 Files

Because the partition boundaries can be rapidly determined at the beginning of the index process, it is possible to perform the compression process in parallel, as each partition is identified and isolated. Very high efficiency is possible during this parallel index construction, because there is no cooperation required between the threads or processes compressing each database partition. NP3 implements just such a parallel mode by using Sun Grid Engine to spawn multiple NP3 processes on one or more computers.

6.2.7.2 Ease Of Updating And Appending To NP3 Files

Partitioning also makes it trivial and rapid to patch an existing database to include new or updated records, since only a single partition need be modified. The current implementation of NP3 contains an append mode to allow the convenient addition of new material to an existing repository, although periodic rebuilding would still be required to sort the material.

6.2.8 Decompression

6.2.8.1 Sequential Record Access

To serially decompress an NP3 file, each database partition is considered in turn. Within each partition, the code stream for each record is interpreted in turn, to reconstruct the sequence. Any references to other records are implicitly met, as all preceding records have already been extracted. This results in a decompression time that is linearly proportional to the number of records decompressed.

6.2.8.2 Random Record Access

Random access decompression functions similarly: if the record makes no reference to other records, then decompression can occur just as for the serial case. Otherwise, the referenced records are decompressed recursively to obtain the necessary referenced strings. Thus the computational cost of random record retrieval is exponential with respect to the number of records in the index partition and the average number of other records referenced by each record.

However, if many records are to be retrieved by random access then the exponential component will be reduced as the probability of each referenced record having already been decompressed increases. Thus in the situation where all records are retrieved by random

access the computational cost of decompression will return to being linearly proportional to the number of records retrieved.

Moreover, it is suggested that random record retrieval in sequence search and alignment is not truly random, in that the objective is to retrieve all records that are similar to some query. Thus, since only records with similar content are referenced, it is reasonable to suggest that a fraction of those records will require retrieval anyway. Thus the effective cost is reduced. The question is whether this occurs to any meaningful extent.

6.3 Results

6.3.1 Compression Of The Human UniGene (Nucleic Acid) Database

To assess the performance of NP3 versus existing compressed formats, the sorted Human UniGene (nucleic acid) database was compressed using NP3. The same database was also compressed using a collection of existing compression methods. General purpose compression algorithms were included in the comparison, because while they are ineffective on typical (unsorted) nucleic acid databases, they can make use of the clustered recurrences in a sorted database — and actually compressed the Human UniGene (nucleic acid) database better than some dedicated DNA compression programs.

DNACompress (Chen et al. 2002b) is used as a representative of DNA compression algorithms, due to its being the best performer that was available freely available. Had GeNML (Korodi and Tabus 2005) and DNAPack (Behzadi and Le Fessant 2005) been freely available, they would have been included in the comparison. This issue has been partially resolved by implementing a variation of GeNML as a codec for the NP3 framework that is optimised for fast random-access sequence retrieval. See Chapter 7 for further details of the GeNML codec for NP3. Finally, where possible, the space used by each program to represent the sequence bodies only (as distinct from the one line sequence descriptions

Table 6.7: Comparison Of NP3 File Size With Other Formats For Human UniGene (Nucleic Acid) Database (1.96×10^9 bases).
Files sizes are listed in mega-bytes (MB) and bits per base (bits). NP3 produces a smaller file than any of the other methods, almost 10 times smaller than the original FASTA file version of the database. The two NP3 variations, "prefer inter-record references" and "prefer short chains" (i.e., avoid inter-record references) produce similar sized databases, with a slight size advantage when preferring to include inter-record references.

Format	Bodies Only		Bodies and Descriptions			Compress
	MB	bits	MB	bits	% NP3	Time (m:s)
FASTA Format (ASCII)	1,886	8.06	2,356	10.10	931%	-
BLAST (formatdb) (Altschul et al. 1997)	489	2.09	1,150	4.93	455%	6:09
DNACompress (Chen et al. 2002b)	459	1.96	512	2.19	202%	30:00
gzip	300	1.28	438	1.87	173%	6:08
bzip2 (Seward et al. 2001)	285	1.21	369	1.58	146%	13:23
GeNML Variant (Chapter 7) (16 way parallel build)	170	0.73	256	1.10	102%	1605:00
NP3 (prefer inter-record refs.)	**166**	**0.71**	**252**	**1.08**	**100%**	**630:40**
NP3 (prefer short chains)	166	0.71	253	1.08	100%	630:40
NP3 (16 way parallel build)	166	0.71	253	1.08	100%	41:54

included in the FASTA format database) is presented. These results are presented in Table 6.7.

These results show that NP3 is able to compress slightly better than the GeNML implementation that is operating in the same framework as the NP3 algorithm, and significantly better than the other algorithms surveyed, including the BZIP2 general purpose compression program (which is also able to make good use of the redundancy in this database), primarily through being able to compress the sequence bodies using only 0.71 bits per base. This compares exceptionally well with the 2.09 bits per base achieved by the BLAST (Altschul et al. 1997) `formatdb` program, which is the only other format surveyed that

allows fast random access to stored sequences. DNACompress performed substantially worse than the general purpose compression algorithms, suggesting that it is not tuned to take advantage of the many recurrences that exist between nearby records in this sorted database.

6.3.2 Compression Speed

As Table 6.7 shows, NP3 compresses slower than any of the other algorithms (except the GeNML Variant), requiring 630 minutes to compress 1.96×10^9 bases ($\approx$ 190 mega-bases per hour). The slow compression is mitigated by near-linear acceleration when the parallel build mode was activated to make use of 16 processors ($\approx$ 2.8 giga-bases per hour; 15.2 times faster), which also validates the previous assertions regarding linear compression time with respect to database size. Considering that NP3 is intended as a distribution algorithm, i.e., decompressed frequently, but compressed only rarely, these figures are tolerable.

6.3.3 Decompression Speed

To assess the decompression speed of NP3, tests were run on a 1.8 GHz AMD Opteron system running RedHat ES 3 (64bit), the results of which are presented in Tables 6.8 and 6.9.

As previously discussed, extracting sequence descriptions from the bzip2 compressed blocks is an expensive process. In light of this, statistics were generated for extracting sequence bodies alone, or along with their descriptions. The tests were run for three different contexts: (a) Global Random; (b) Local Random, and; (c) Sequential.

Table 6.8: NP3 Decompression Performance (UniGene Nucleic Acid Database). NP3 configured to prefer inter-record references if the cost is the same as the cheapest option, i.e., code selection ties are broken in favour of inter-record references.

	Without Descriptions		With Descriptions	
	Records/sec	Bases/sec	Records/sec	Bases/sec
Global Random	748	417,984	366	204,481
Local Random	71,160	44,642,536	644	409,617
Sequential	339,650	193,303,120	27,675	16,486,549

Table 6.9: NP3 Decompression Performance (UniGene Nucleic Acid Database). NP3 configured to avoid inter-record references, by using them only if they are the single cheapest option, i.e., code selection ties are broken in favour of not making inter-record references.

	Without Descriptions		With Descriptions	
	Records/sec	Bases/sec	Records/sec	Bases/sec
Global Random	950	533,276	378	212,214
Local Random	79,644	49,872,516	627	395,811
Sequential	341,103	194,272,864	27,493	16,370,482

6.3.3.1 Global Random Decompression

Global Random access refers to randomly selecting any one of the 3,480,838 records in the database and recalling it. The slight reduction in file size that the prefer-inter-record-references-when-cost-neutral heuristic offers comes at a substantial cost to decompression speed for random sequence access, both in this context and in the Local Random context.

It is not surprising that Global Random access is slow compared to the other contexts, as there is no spatial locality to exploit, and so each request takes a degree of effort to service. Notwithstanding this, it is still possible to randomly retrieve hundreds of sequences per second in this way, a feat not previously possible with such a compact representation of the data.

6.3.3.2 Local Random Decompression

Local Random access refers to randomly selecting any one of the 6.5×10^5 records in a given database partition. This is useful as an indication of the retrieval speed that can be expected if NP3 were used in the DASH sequence search and alignment program, where this is exactly the kind of activity that NP3 will be subjected to. This is because DASH consults each partition in turn. Here, the spatial locality, and NP3's caching of previously extracted records within a database partition provides several orders of magnitude improvement, provided that sequence descriptions are not required. This was sufficient for the decompression rate to reach well into the target range required for integration with the DASH algorithm (tens of thousands to millions of records per second).

Decompression is slightly faster when the NP3 file is constructed with fewer inter-record references. This is in line with expectation since resolving inter-record references necessitates decompressing extra records. Nonetheless, the difference in decompression speed is relatively small.

Retrieval of descriptions could be improved by caching decompressed description blocks similar to the way that extracted sequence bodies are cached now. Indeed, for Sequential decompression, retrieval of sequences with descriptions is over 100 times faster, because the present caching of a single block of sequence descriptions is sufficient in that context.

6.3.3.3 Sequential Decompression

Finally, *Sequential* access refers to asking for each successive record in the database. This provides a good measure of the true decompression speed of NP3, as each record is decompressed exactly once. The entire database could be sequentially decompressed in 130 seconds on the test hardware. If sequence descriptions were not required, then decompression could be accomplished in just 10.6 seconds, equating to 185 million bases per second, as summarised in Table 6.10. This is slightly faster than gzip, and much faster than

Table 6.10: Nucleotide Decompression Speed (No Descriptions) Of GZIP, NP3, And The GeNML Implementation Of Chapter 7 For The De Facto DNA Corpus, And The DNA Databases Used In This Dissertation.

Database	Program	Seconds	Bases per Second
Human UniGene	GZIP*	13.23	148,100,786
	NP3	10.58	185,195,973
	GeNML Variant**	91.95	21,309,137

* To make the comparison between GZIP and the NP3 and GeNML Variant algorithms, GZIP was given an input file consisting only of the DNA letters, i.e., striped of all descriptions and white space.
** These GeNML results were produced using the implementation of the GeNML described in Chapter 7.

the GeNML Variant. The relative slowness of GeNML is not surprising, as it is the only algorithm of the three to make heavy use of Arithmetic Coding. However, NP3 must be much slower than CINO (Williams and Zobel 1997a), since that algorithm is claimed to be several times faster than gzip — but CINO cannot compress DNA to below 2 bits per base.

6.3.4 Compression Of De Facto Corpus

However the performance of NP3 is less impressive where there is little or no opportunity for exploiting the inter-record redundancy of a well sorted database, such as the small de facto corpus (for an example, see Chen et al. (2002b)) used to evaluate existing DNA compression algorithms that focus on intra-record redundancy. In a complete inversion, NP3 compresses those sequences at an average of 2.03 bits/base, versus 1.73 bits/base using DNACompress, and 1.69 bits/base for GeNML (Korodi and Tabus 2005). This reflects the fact that NP3 focuses on large scale inter-record redundancies rather than on the intra-record redundancy and other DNA structures that DNACompress leverages: each is operating on a different scale of structure. GeNML appears to combine the best of both worlds, in that it obtains the best compression of the de facto corpus, and also performs well on the Human UniGene (nucleic acid) database. However, as previously mentioned, GeNML uses Arithmetic Coding, and hence decompresses much more slower than NP3.

6.4 Conclusions

The results demonstrate that, given a database that is well sorted, the NP3 algorithm is capable of producing loss-less representations of nucleic acid databases that are at least as small as the best known algorithm (GeNML), but unlike GeNML, preserving the desirable feature of fast random access to sequences. NP3 offers decompression of random records within a partition, and with sufficient speed to make it a potential database format for sequence search and alignment algorithms. This makes NP3 uniquely suited to providing a database representation for nucleic acid search and alignment applications that is several times smaller than is current used. However, NP3 does not compress as well as GeNML if the database does not contain much large scale redundancy. Thus NP3 and GeNML represent a trade-off between compressed size and decompression speed. A similar trade-off exists between NP3 and CINO, with CINO being even faster to decompress than NP3, but with a lower ratio of compression.

It is true that other compression schemes such as SEQUITUR (Nevill-Manning and Witten 1997) could be expected to obtain similar or better compression than the NP3 byte-aligned codec, but without the somewhat arbitrary coding decisions used in that codec. However, this does not detract from the utility of NP3 in supporting the hypotheses of this dissertation. Nor does it detract from the NP3 framework, as it is essentially codec independent. The consideration of an appropriate SEQUITUR variant as an NP3 codec with a particular emphasis on examining the speed and compression trade off would be a fruitful future exercise.

As mentioned, in the traditional context of nucleic acid databases where there is little exploitable local redundancy, NP3 has inferior compression performance compared to the existing algorithms. However, the recent advent of efficient database reordering algorithms and the immense internal redundancy of collections such as GenBank (recall that GenBank includes about 12×10^9 bases that originate from the human genome, yet the human

genome consists of only about 3×10^9 bases, indicating a four-fold redundancy for that portion of the database), combined with the attractive compression of sorted databases presented here, suggests that database reordering of nucleic acid databases is a powerful approach. As database sizes continue to balloon, and the internal redundancy increases as a natural result, this method has the potential to become increasingly potent as an effective means of containing the storage, processing and transmission costs of these databases.

In short, NP3's (1) effective compression; (2) database portability afforded by byte order independence, and; (3) asymmetric computational cost favouring decompression, all contribute to making it useful distribution and search format for large sorted DNA databases, provided that the databases contain sufficient local redundancy. While sorted databases may be uncommon today, it is almost inevitable that sorting techniques such as enabled by algorithms such as SPEX (Bernstein and Cameron 2006) will be applied to make the collections more manageable.

6.5 Future Directions

In the current implementation, a completely byte-aligned code format is used to facilitate high speed decompression in software. It is recognised that a bit aligned format would be more space efficient, but at the cost of decompression speed. Determining the precise cost of this decision would be constructive in objectively assessing this design decision. More generally, compression gains are all but certain through further exploration of coding method space. The complementary strengths of NP3 and DNACompress suggests that a combination of their methods has the potential to yield an algorithm with superior compression to both. This is considered in part in Chapter 7 by substituting the byte-aligned codec of this chapter with a variant of the GeNML algorithm.

It is also worth noting that the compression ratio of the sequence description data, at best 7:1 in our experiments with BZIP2, often lags behind that which NP3 achieves on the se-

quence bodies, to the point where the compressed sequence description data constitutes around 30% of the total NP3 compressed Human UniGene (nucleic acid) database. While the current approach is considered a satisfactory starting point, further work in the area of sequence description compression is warranted — particularly as databases and their inherent redundancies grow; the entropy of the sequence description will become the dominant factor in compressed file size. Given the specialised vocabulary common in biological sequence descriptions, compression priming techniques that take advantage of the global properties of the data being compressed (Bell et al. 1993) are an obvious starting point. The structure presented by the almost universal inclusion of accession numbers in sequences descriptions is also a logical target for improved compression. This could be attacked by utilising a dual-model compression scheme, with separate models for encoding the accession numbers and the textual part of the descriptions, respectively.

Work is also warranted on the creation and dissemination of efficient and effective tools for sorting and compressing the major biological sequence databases, and then searching the compressed databases that result. The combination of DASH and NP3 in Chapter 7 explores this goal by searching NP3 compressed sorted nucleic acid databases.

Chapter 7

NIX: Producing Compact Cooperatively Compressed Indices Of Biological Sequence Databases

Introduction

This chapter describes the NIX (short for NP3 Index) algorithm, which builds on the foundation laid by the FOLDDB algorithm introduced in Chapter 5. The method of cooperative compression presented in that chapter proved effective, provided that whole records were redundant. In contrast, NIX uses a fine grained method of cooperative compression that is effective, even when only parts of records are redundant. In order to reach this goal, this chapter also presents modifications to the NP3 algorithm, and adds NIX/NP3 searching capability to DASH.

The algorithms presented in this chapter are intended as a proof of concept, and thus they are tuned for compact index size rather than speed. Making these algorithms computationally efficient is left as a future avenue of research. Therefore, search speed is compared between the unfolded and fine-grain folded versions of the software described in this chapter, rather than being compared with other software, such as BLAST.

Results are presented showing that by performing cooperative compression using the algorithms described in this chapter, it is possible to reduce the size of already compressed indices by a further 40%. Moreover, it is shown that searching these compact indices yields improved sensitivity, thus supporting the thesis of this dissertation. However, for this proof of concept at least, these gains come at the cost of greatly inflated search times.

Finally, brief results are shown attempting to apply the fine grained cooperative compression algorithms of this chapter to a much less redundant nucleic acid database (the Human Genome), which demonstrates that, as things currently stand, the algorithms have limited applicability to unsorted databases.

7.1 The NIX Indexing Algorithm

The NIX indexing algorithm consists of constructing a conventional compressed inverted list that excludes redundant postings can be recomputed during decompression with the assistance of the recurrence records.

7.1.1 Omission Of Redundant Postings

Cooperative compression consists of posting in the index only one instance of each recurrent string. The question arises as to which instance should be posted. This problem can be simplified if we resolve sets of recurrent strings into pairs, thus reducing the question to: Should we post the first or the last occurrence? This question is answered by considering the context (sequence search and alignment) in which the cooperatively compressed index will be searched.

Sequence search and alignment involves extracting records from a database that are similar to some query sequence. Extracting a record from an NP3 compressed database requires the resolution of any recurrence chains that the record references. Therefore, when some

record, R_n, is extracted, the possibly empty set of records $A(R_n)$, on which it depends, must also be extracted. Therefore, it makes sense to post the last instance of a recurrent string, as the postings can be cloned into the context of each record in $A(R_n)$ as they are extracted from the database (recall that recurrence chains in NP3 files link toward the start of a database). Specifically, no additional data structure is required to facilitate the efficient reproduction of the omitted postings from the cooperatively compressed index. This is *Reverse Indexing*.

However, if multiple sequences point to a common source, then only one instance (the one that is in the common source) can be omitted from the cooperatively compressed index. This can be mitigated to some extent by constructing the NP3 database so that common sources are avoided whenever possible. However, this is at the cost of producing longer recurrence chains.

In contrast to Reverse Indexing, it is possible to instead index only the first instance of each pair of recurrent strings. This is *Forward Indexing*. The advantage of forward indexing is that when a recurrent string occurs many times, only one instance requires posting in the index (the common source). Thus, this method requires, at most, an equal number of postings to Reverse Indexing, but requires fewer postings than Reverse Indexing when common sources exist. However, the disadvantage of forward indexing is that because it is the first and not last repetition of a string that is posted, the recurrence chains in the NP3 file link forwards, which is the wrong direction to provide a path to locate the other repetitions of the string; an additional data structure is required to provide this information. Nonetheless, if this data structure is smaller than the space saved by omitting more postings from the index, then the result will be more compact than can be achieved with Reverse Indexing.

Figure 7.1, demonstrates the difference between Forward Indexing and Reverse Indexing with a simple example consisting of four postings, Posting 1, Posting 2, Posting 3 and

Posting 4, each of which correspond to instances of a recurrent string. The set of four postings are decomposed into pairs (indicated by arrows in the figure) where each posting is linked to the next nearest posting towards the start of the database partition. In our example, the recurrent string is not exactly repeated between Postings 3 and 4, thus requiring Posting 4 to point to Posting 2 rather than Posting 3.

With Reverse Indexing, considering each linked pair in turn, we need only store the last instance. Therefore we cancel the storage of the first instance. Thus the pair (Posting 1, Posting 2) allows the cancellation of storage of Posting 1. Similarly either pair (Posting 2, Posting 3) or (Posting 2, Posting 4) allows us to cancel the storage of Posting 2. However, no instances point to Postings 3 or 4, so they must be stored. Thus, with Reverse Indexing, two postings must be stored.

With Forward Indexing, the process is similar, but we cancel the storage of the latter posting in each pair. Thus (Posting 1, Posting 2) allows the cancellation of storage of Posting 2; (Posting 2, Posting 3) the cancellation of storage of Posting 3, and; (Posting 2, Posting 4) the cancellation of storage of Posting 4. Forward indexing has the advantage that where multiple postings point to the same source, as with (Posting 2, Posting 3) and (Posting 2, Posting 4), we need only store one posting, the common source, where as with Reverse Indexing we must store one posting per pair. Thus, with Forward Indexing, only one posting must be stored. However, as previously mentioned, with Forward Indexing, we must maintain a list of which database records contain references to which other database records, thus reducing the space advantage.

What is common in both cases and forms the core of the NIX indexing algorithm is that postings are excluded from the index where they can be shown to be redundant by virtue of their residing within a recurrent string.

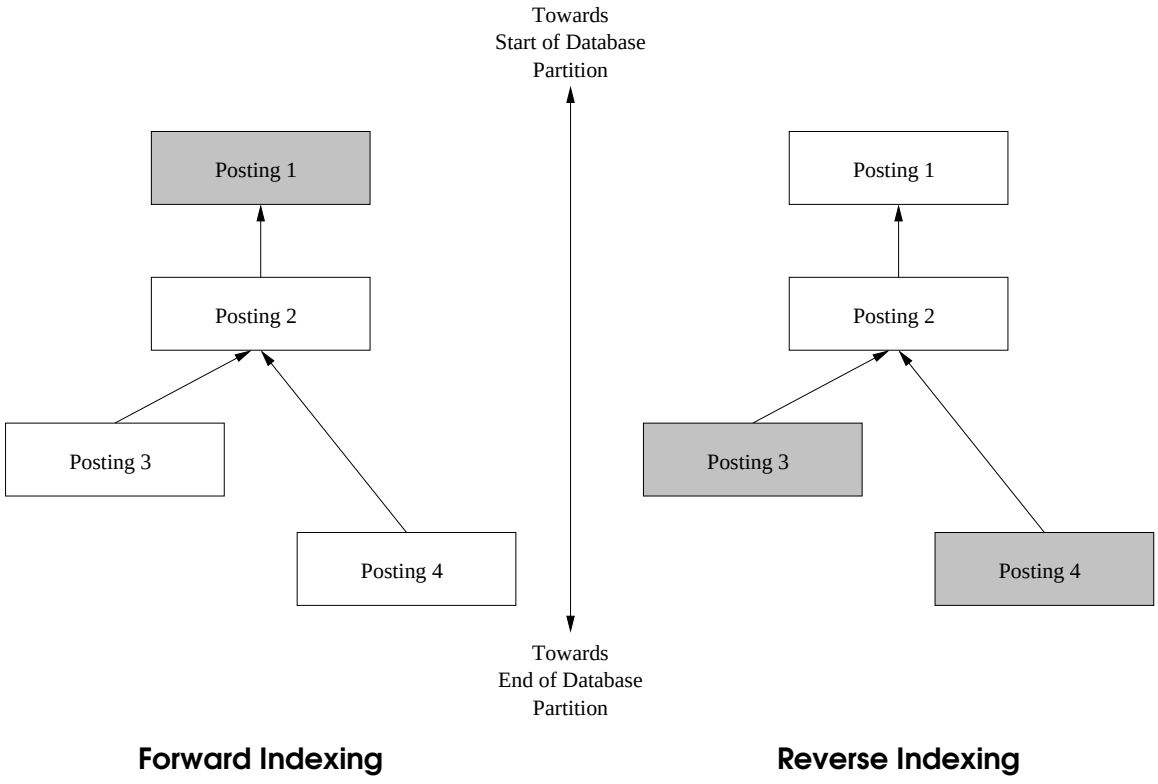

Figure 7.1: Forward And Reverse Indexing Of Chains Of Recurrences. Here four postings are depicted, each corresponding to an instance of a recurrent string. Shaded postings indicate postings that must be stored. Forward Indexing stores the head of each recurrence chain, while Reverse Indexing stores the tail (or leaf node) of each recurrence chain. Reverse Indexing may store more postings because, as in this example, branching may occur, meaning that there may be multiple leaf nodes for a given recurrence chain.

7.1.2 Reconstruction Of Omitted Index Postings

The method of reconstruction of omitted index postings differs for Forward and Reverse direction indices.

For Reverse Indexing, the method is straightforward and efficient: Whenever an HSP is discovered: (a) check the recurrence records that correspond to that record (recall that these are extracted implicitly when decompressing the NP3 file), and; (b) translate the relevant portion of the HSP into each context indicated by the recurrence records. This process is performed recursively until no new contexts are discovered.

For Forward Indexing, the situation is complicated by the fact that the recurrence records cannot be used to determine the existence of instances of recurrent strings further forward in the index (recall that the recurrence records in an NP3 file always point to records earlier in the database). Therefore an additional data structure must be consulted that indicates for each record, R, the forward records, R_f, that have recurrence records that point to R. This list is then traversed, and the HSP translated into each context identified by a recurrence record in some R_f that intersects with the HSP as discovered in R. This process is performed recursively until no new contexts are discovered.

This process is not as efficient as the reverse indexing case, because the data structure may indicate records that depend on the record under consideration, but are not similar in the region of the HSP being translated. Therefore, unlike in reverse indexed searching, records may be unnecessarily extracted and examined.

Note that a Forward direction index can be subjected to both the reverse and forward direction searches, with the potential of increased sensitivity, but at the cost of computational burden. The additional sensitivity arises because, while the forward search will exhaustively discover all omitted postings (and so guarantee no loss of sensitivity compared to a conventional index), by searching forwards and backwards for non-exact HSPs (recall that

HSPs are permitted to contain substitutions), it is possible to discover approximate matches that would otherwise remain undetected.

An issue that is not seriously considered in this dissertation is whether this trade-off in predictability of execution time is overly excessive, or if it is sufficiently offset by improved sensitivity. It would be a productive future exercise to explore this issue in more detail, perhaps with the objective of discovering techniques that mitigate or manage this issue.

7.1.3 Re-Use And Minimisation Of HSP Discovery Effort

Compared with searching an ordinary index, searching a cooperatively compressed index necessitates an additional and potentially costly translation step whenever an HSP is discovered. This step involves taking some HSP, h, and determining if the record segment it covers corresponds to any recurrent strings (as recorded in the NP3 file). If it does, the recurrent regions are translated into the context of each instance of the recurrent string, to produce translated HSPs, h'_1 ... h'_n.

If the HSP being translated lies entirely within a recurrent string, then the translated HSP does not require extension, because the translated location has the same immediate context as the original HSP. In other words, the details of the context that defined the end points of the original HSP are the same in the context of the translated HSP. However, if the HSP being translated includes the boundary of one or more recurrent strings, then the translated HSP requires extension on the boundary that the HSP crosses, as that end of the translated HSP has a different context to the original HSP. Figure 7.2 illustrates the process, showing: how the translation occurs; how the previous HSP extension work is reused; and, how additional HSP extension is minimised.

HSP 1 in fragment (a) lies completely within the recurrent string, and so its immediate context does not change, i.e., it is still bounded by `GTCAT` on the left, and `CGTTT` on the

Figure 7.2: HSP Translation Scenarios. Each record fragment (a), (b) and (c) contains a recurrent string, as represented by the dashed boxes. Three HSPs are found as represented by the solid boxes, and identified by the corresponding super-scripts.

right. Therefore, when it is translated into fragments (b) and (c), it does not require further extension.

In contrast, HSP 2 completely contains the recurrent string. In this case, only the portion of the HSP that is shared with the recurrent string can be translated into fragments (a) and (c). Further, because the immediate context of the translated portion of the HSP has changed (`GTACTGC` for record (a), or `CCGTACG` for record (c), instead of `GTTGCAA` on the left, and `GGATACGTC` for record (a), or `ATATTAGAC` for record (c), instead of `TAGTATCGC` on the right), this translated HSP must be extended at both ends.

Finally, HSP 3 presents a combination of the conditions of HSP 1 and HSP 2. Only the portion of the HSP that intersects the recurrent string (`GTCGTTT`) can be translated into fragments (a) and (b). Because the context on the left hand side does not change, the translated HSPs do not require extension on that side. However, the context does change on the right hand side, and so the translated HSPs must be extended on that side.

In each case, translating each of the three HSPs preserved the maximum extent of each HSP when translated. Moreover, the translated HSPs do not always need to be extended on both ends in order to ensure that they remain maximally extended. Together, these factors reduce the computational burden compared to discovering each of the nine final HSPs independently. The assumptions in this process are these: (1) that the list of all

instances of each recurrent string can be efficiently identified, and; (2) that HSP translation can be performed more quickly than HSP discovery and extension.

A final consideration in this process is that by requiring the DASH algorithm to now follow chains of recurrences during the early part of the search process, rather than merely cloning results at the end, detracts from the attractive time complexity properties of the DASH algorithm. Specifically, while the NIX algorithm is linear with respect to the number of stored postings, it inherits NP3's exponential upper bound when reconstructing and exploring the cooperatively compressed parts of the index. While the NP3 decompression cost can reduce to linear when all records are retrieved, the same is not true of the HSP translation cost when considering the cooperatively compressed postings. This is because the recurrence records must be examined even if the database record they correspond to has already been retrieved.

7.2 NIX Index Format

The NIX index format is an inverted file, similar to the FOLDDB index format. However, there are two important distinctions between NIX and FOLDDB.

The first distinction is that, where as the FOLDDB index contains all the data structures required to search the database it represents, including the sequence bodies, the NIX format contains only the index structures; the storage of the sequence bodies is handled by the NP3 file format. Thus, while only a single file is required to search a FOLDDB indexed database, two files, an NP3 file and a NIX file, are required to search a NIX indexed database.

The second distinction is that the representation of the inverted lists is made to be space efficient rather than fast to decompress. Rather than using an ad hoc compression scheme, recent advances in inverted list compression are used to minimise the space requirements.

7.2.1 Why Pointers To *k*-mer Indices Were Not Compressed

The lexicon size of a fixed-width index can be calculated from the alphabet size and the index width. Further, for typical index widths, this lexicon size is very small. Any useful representation of an inverted file must store a pointer to the posting list of each *k*-mer in the lexicon. Therefore, while the pointers to the compressed inverted list corresponding to each *k*-mer could be compressed, for example according to the method of Bell et al. (1993), because the lexicon is small compared to the size of the database, the savings would be small (only a few percent for a typical NIX file). Therefore, these pointers were stored uncompressed in a NIX file. This is the one exception to the general rule that NIX structures are optimised for compactness rather than access speed.

7.2.2 Compressing The Inverted Lists

In place of the fixed and ad hoc inverted list compression of the FOLDDB algorithm, and similar to NP3, NIX uses a plug-in system so that it can support a variety of inverted list compression coding and decoding schemes (*codecs*). This separates the cooperative compression functions of NIX from the inverted list coding, and allows greater freedom in the selection of the optimal codec for a given application, particularly as new inverted list compression schemes are developed.

The selection of a suitable codec for inclusion in NIX required some thought into the effect that cooperative compression would have on the distribution of values in an inverted list. The greatest impact that cooperative compression has on this distribution is that it can thin out any clusters in an inverted list. This occurs because cooperative compression merges the index postings of nearby recurring strings. Therefore clusters of postings may become sparser, and in the extreme, each cluster may be reduced to a single posting.

However, it is unlikely that clustering will be eradicated altogether, especially since similar, but not identical, sequences are common. Since the sequences are not identical, the NP3 al-

gorithm cannot merge their storage, and because NIX uses the recurrence information from the NP3 file, such clusters will not be affected. Therefore, the inverted list compression algorithms that can make good use of clusters remain attractive. Two recent cluster-sensitive inverted list coding schemes were considered: Interpolative Coding (Moffat and Stuiver 2000, 1996), and Selector Coding (Anh and Moffat 2005, 2004).

Interpolative Coding is slower to decompress than Selector Coding (Trotman 2003), because Interpolative Coding involves recursion. However, compression is much faster, because there is no searching phase in the compression algorithm. In contrast to Interpolative Coding, Selector Coding must search for the best selector at every iteration. Even if this is performed heuristically, empirical evidence suggests that compression is many times slower than for Interpolative Coding, and more than an order of magnitude slower if optimal compactness is to be achieved. Turning from speed to effectiveness, Interpolative coding generally compresses inverted lists better than Selector Coding, and indeed better than any of the other published inverted list compression schemes published at the time of writing (Anh et al. 2001, Moffat and Stuiver 2000). In summary, Interpolative Coding is faster to compress, and compresses better, while Selector Coding offers faster decompression.

Improved compression effectiveness, i.e., smaller representations, do not always translate into greater query throughput. This is, first, because decompression always takes time, and, second, because of fixed delays, such as disk seek and transfer times (Trotman 2003). However, in the context of this dissertation it is assumed that an index will be resident in RAM, and so the retrieval time will be negligible. Moreover, since the index must fit in to RAM, compactness takes priority, even though this may carry a speed penalty: The penalty of not fitting into RAM would be much greater.

In light of this need to minimise the compressed index size, Interpolative Coding was chosen as the inverted list codec for NIX. To mitigate the decompression speed penalty, a fast Interpolative Coding routine was written that is iterative rather than recursive.

7.2.2.1 Creating A Fast Interpolative Coder

Interpolative Coding compresses a list of ascending values by recursively dividing the interval and using only the minimum number of bits to place each value within its interval. This is best illustrated with an example. The example from Moffat and Stuiver (2000) is reproduced.

Consider the following inverted list $\langle 7; 3, 8, 9, 11, 12, 13, 17\rangle$, i.e., an inverted list containing the seven values 3, 8, 9, 11, 12, 13 and 17. Further, assume that the upper limit is known to be $N = 20$. In encoding the first value, 3, assume that the second value is already known to be 8. In that case, the first value must lie in the range 1..7: The first value can be thus be encoded in three bits. Moving to the third value, 9, assume that in addition to knowing the second value, that the fourth value, 11, is also known. Therefore, the third value must lie in the range 9..10, and can be encoded in a single bit. For the fifth value, assume that the forth and sixth values are known to be 11 and 13. In this case, the fifth value must be 12: Therefore no code is required to place this value. Finally, to encode the seventh value, we know that the upper limit, N, is 20. Therefore the value must lie in the range 14..20, and can be encoded in three bits.

This is all very well, but it has been assumed that the second, fourth and sixth values were already known. This can be dealt with if that list of values, $\langle 3; 8, 11, 13\rangle$, is encoded first. Following the same method as for the original list, assuming that the second value, 11, is known when encoding the first value, then the first value must lie in the range 1..10, and can be encoded in four bits. However, we know that there is a value on either side of it. Therefore, it cannot be 1, nor can it be 10. This allows the range to be narrowed to 2..9,

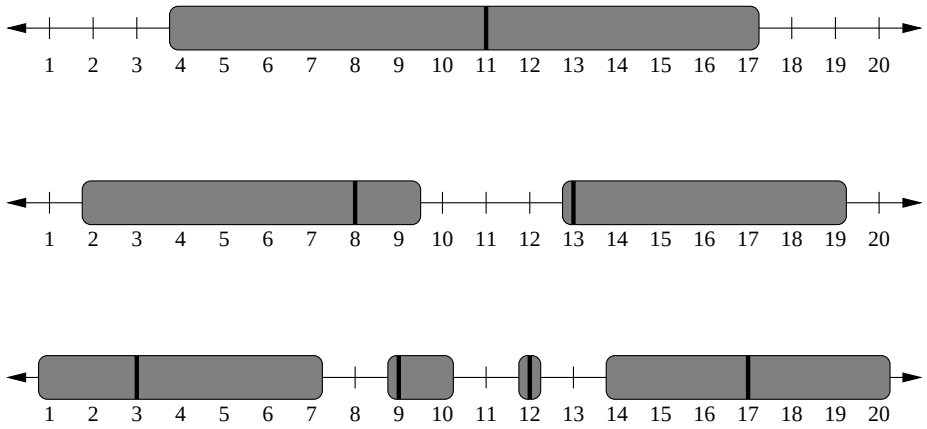

Figure 7.3: Example Interpolative Coding Intervals For The Inverted List, ⟨7; 3, 8, 9, 11, 12, 13, 17⟩, drawn from Moffat and Stuiver (2000).

and hence one bit can be shaved from the code. The third value, likewise, must lie in the range 12..20, which using the same logic as before, can be narrowed to 13..19, also saving a bit.

Finally, the second value of this list must be available before these two values can be encoded. Thus the list $\langle 1;11\rangle$ must be encoded. The single value, 11, must lie in the range 1..20. Now it is known that three values must lie on each side, so the range can be narrowed to 4..17, and the value can be encoded in four bits. However, by using a centred minimal binary code (as described by Howard and Vitter (1992)), the $2^4-(17-4+1)=16-14=2$ spare values in the four bit code can be traded away in favour of making two of the remaining codes one bit shorter. If these are placed in the centre of the range, then in this case a further bit can be saved.

Figure 7.3 depicts the recursive Interpolative Coding example described above. The shaded regions indicate the interpolated range of each value, and each layer of recursion is depicted on a separate number line. The number of layers of recursion is $\lceil log_2(n)\rceil$, where n is the length of the list.

In order to increase decoding speed, the fast interpolative coder in NIX replaces recursion with a blocked iterative strategy. This is possible, because, for a fixed length inverted list,

the elements of that list are visited in a deterministic order, independent of the values of the elements. By pre-computing tables for: (a) the order of visitation, and; (b) the nearest elements on either side that are known. These tables can then be traversed, allowing Interpolative Coding to be performed without recursion, and without maintaining an explicit stack.

This is all very well, provided the table matches the length of the inverted list being encoded or decoded. However, it readily generalises to allow pseudo-iterative Interpolative Coding of both shorter and longer lists.

The generalisation for shorter lists is to add monotonically increasing values to the end of the inverted list until it is the same length as the table. Because the values are monotonic, the Interpolative Coder does not require any bits to encode this perfect cluster of values. Further refinement is possible: By maintaining additional pre-computed tables that indicate the first and last positions in the table that require evaluation for a given length list, the number of elements can be reduced to be approximately equal to the length of the list being encoded or decoded.

The generalisation for longer lists is to use a hierarchical method. This introduces a limited amount of recursion, in return for raising the upper bound on the length of the inverted lists that can be processed. For NIX, the maximum required inverted list length is 2^{24}. Using a table size of $2^8 - 1 = 255$, only three levels of recursion are required, which can be efficiently implemented as a nested loop. The table size is 255, and not 256, because every 256^{th} value is placed by the middle of the three levels of recursion, and every 256^{th} of those values is placed by the top of the three levels of recursion. The block size of 255 also obeys the advice of Moffat and Stuiver (2000) in selecting trees to be of size $2^k - 1$, so that sub-trees will be of equal size. Also in obedience to Moffat and Stuiver (2000), the centred minimal binary code is used for all but leaf nodes, where the shorter codes are instead placed on the outside of the range.

The use of tables to accelerate Interpolative Coding is similar to the methods of Cheng et al. (2004). However, the methods presented in this dissertation use only Interpolative Coding, and unlike Cheng et al. (2004), do not introduce any compression leakage.

A further optimisation to Interpolative Coding that has not been implemented, is to use knowledge of the *k*-mer that an inverted list corresponds to. Most *k*-mers, cannot occur at successive positions, because the overlapping $(k-1)$ bases must be in agreement. For example, consider the 8-mer `ACGTAGTA`. This 8-mer can only occur every seven bases, because only the A's at each end will match. However, `ACGTACGT` could occur every four bases. This minimum distance that a *k*-mer can follow itself can be easily computed, and could be used when narrowing the interpolated range between the previous and next known entries in an inverted list. Given the frequency of repeats in most genomes and transcriptomes, this may deliver substantial space savings.

7.3 Modifications To NP3 To Optimise For Forward And Reverse Indexing

As it was described in Chapter 6, the NP3 algorithm was not optimised for either the Forward Indexing or Reverse Indexing forms of cooperative database and index compression. This is addressed by the introduction of three optimisations: two general optimisations that apply to both Forward Indexing and Reverse Indexing; and one optimisation specific to Reverse Indexing.

7.3.1 Optimisation One: Preferring Inter-Record References

By default, the NP3 algorithm attempts to minimise the length of recurrence chains in order to maximise decompression speed. Specifically, if there are two options to encode a given record fragment which result in identical bit lengths, the one that does not require

an inter-record reference will be chosen in preference to the one that does require an inter-record reference. The trade-off is that fewer postings are omitted from the cooperatively compressed index.

The first optimisation, then, is to reverse this policy and prefer inter-record references, using them whenever they require no more bits than the best alternative.

7.3.2 Optimisation Two: Per-Posting Rebate

The first optimisation can be taken further by estimating the reduction in index size that results from any given inter-record reference. This cost estimate can then be taken into account by the NP3 optimiser. Rather than minimising the NP3 file alone, the optimiser would then be optimising the combined NP3 and NIX file size.

While this method of minimising the combined file size is simple in principle, it is difficult to accurately predict the reduction in index size that results from any given inter-record reference. This is because the index is compressed; it is more difficult to correctly predict the savings in compressed index size than it is to predict the savings in uncompressed index size. Nonetheless, a simple approximation of a fixed number of bits per posting was tried.

The second optimisation, then, is to offset the cost of each inter-record reference by a fixed number of bits for each posting that it removes from the index.

7.3.3 Optimisation Three: Maximising Inter-Record Reference Target Coverage

The third modification is specific to Reverse Indexing. It is based on the fact that Reverse Indexing posts in the index only the last instance of a recurrent string pair, i.e., it excludes the posting for the first (source) instance of the string. Therefore, when using Reverse Indexing, the more unique source material included in inter-record references, the more

postings that can be omitted, and the greater the savings. To this end, the NP3 algorithm is supplemented with a coverage bitmap that indicates whether each residue in the current window has been used as a source string or not. When choosing the best source material for a recurrent string from among several options, only the number of previously unused residues in the source are counted when assessing the savings. While this may result in a larger NP3 file, it should result in a smaller combined NP3 and NIX file size.

The third optimisation, then, is to pair recurrent strings such that the pairs preferentially reference unique source material, thus maximising the number of postings that can be omitted from the index using Reverse Indexing.

7.4 Searching NP3/NIX Ensembles With DASH

The DASH algorithm required relatively few modifications in order to search NP3/NIX ensembles. The most obvious change was adding the code to read from NP3 and NIX files. Beyond that, the only other additions were: (a) the code required to reconstruct the postings omitted from the index, and; (b) the code to take advantage of the presence of recurrent strings in order to enhance the sensitivity of the HSP discovery stage of the DASH algorithm.

7.5 Comparison of NP3 and GeNML

The NP3 byte-aligned compression algorithm introduced in Chapter 6 is effective for databases containing many long repeated strings. However, NP3 is not effective when applied to typical nucleic acid databases that do not contain as many long repeated strings, such as genomes of single organisms. Indeed, NP3 is outperformed in these circumstances by algorithms such as DNACompress (Chen et al. 2002a) and GeNML (Korodi and Tabus

2005). This is unfortunate, because NP3 does offer the attractive characteristics of: (a) faster decompression, and; (b) the explicit coding of inter-record references, allowing the construction of compact indices. This trade-off is explored by adding a GeNML codec to NP3 to allow the comparison of the two algorithms in the context of a sequence search and alignment system.

A new implementation of GeNML was created as the source code for GeNML is not freely available. In order to fit within the framework of the DASH sequence search and alignment system, this implementation differs in several ways from the canonical GeNML algorithm published by Korodi and Tabus (2005). First, database partitioning reduces the average window size. Secondly, synchronisation points are added for each database record, with each record forming a single GeNML macro-block. Both of these modifications slightly degrade compression performance, but were unavoidable in adapting GeNML for use. Moreover, exactly the same concessions apply to the NP3 byte-aligned compression algorithm, thus making the comparison fair. Finally, the GeNML implementation used here boasts a hybrid arithmetic and direct coding scheme that uses arithmetic coding only when required, thus accelerating decompression speed.

7.6 Presentation Of Duplicated Results

The explicit representation of recurrence records as described in this chapter make it possible to efficiently determine for a pair of alignments whether they are sourced from identical or near identical material. When that is the case, it can be argued that such results should be presented to the user in such a way that these relationships are clearly visible. Indeed, when alignments are identical, only one of the alignments need be displayed, as the others can be fully described by indicating only the source sequence of each. That is, because we have "knowledge of content-equivalence relationships within a collection" (Bernstein and Zobel 2005), more effective user interfaces can be envisaged. The attraction of such representa-

tions increases as the redundancy within biological sequence databases increases over time. The attractiveness of merging results for easier reading is affirmed by the fact that BLAST has formatting options that partially achieve this goal. However, the implementation of such a system is outside of the scope of this dissertation. Therefore, the presentation of results in the current experiment remains unchanged from that used in previous chapters, i.e., a BLAST-like report.

7.7 Method

In order to assess the effectiveness of the NIX cooperatively compressed index format, NP3 and NIX ensembles were constructed of the Human UniGene (nucleic acid) database with a variety of configurations. For each configuration, DASH was run using the two canonical parameter sets described in Chapter 3, and the indices were constructed using frequent *k*-mer (i.e., stop *k*-mer) exclusion thresholds of 1.5, 2.5, 5.0 and 10 times random expectation. The list of configurations below describe the cooperative compression optimisations to NP3 that were enabled for each test. The section and table numbers refer, respectively, to the corresponding text from the previous section, and the tables in Appendix A that contain the command line parameters passed to NP3, NIX and DASH. The configurations used were:

1. Cooperative compression disabled, and stop *k*-mers excluded (as a negative control) (Table A.11);
2. Forward Indexing, and stop *k*-mers excluded (Table A.12);
3. As for (2), but preferring inter-record references (Section 7.3.1) (Table A.13);
4. As for (3), but with per-posting rebates (Sections 7.3.1 and 7.3.2) (Table A.14);
5. As for (4), but with stop *k*-mers included (Sections 7.3.1 and 7.3.2) (Table A.15);
6. Reverse Indexing, and stop *k*-mers excluded (Table A.16);
7. As for (6), but preferring inter-record references (Sections 7.3.1) (Table A.17);
8. As for (7), but with per-posting rebates (Sections 7.3.1 and 7.3.2) (Table A.18);
9. As for (8), but maximising distinct source material (Sections 7.3.1, 7.3.2 and 7.3.3) (Table A.19); and
10. As for (9), but with stop *k*-mers included (Sections 7.3.1, 7.3.2 and 7.3.3) (Table A.20).

To assess the effectiveness of NP3/NIX compression on a database with a more typical level of redundancy, the Human Genome (nucleic acid) database was also compressed and indexed, using configurations 1 and 3.

Using the methods described in Chapter 3, the following measurements were made for each of the experiments described above:

1. Increase in search sensitivity relative to operating without cooperative compression (i.e., the negative control), and the peer group of algorithms;

2. Reduction in NP3 and NIX index sizes (and the time taken to construct them) relative to operating without cooperative compression (i.e., the negative control); and

3. Reduction in search speed relative to operating without cooperative compression (i.e., the negative control), and also the results of Chapter 5.

7.8 Results

7.8.1 Improved Search Sensitivity

Tables 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7 and 7.8 present the sensitivity information for DASH Mode 2 and DASH Mode 4 for the four posting exclusion thresholds: 1.5× random expectation, 2.5× random expectation, 5.0× random expectation, and 10× random expectation.

The sensitivity scores listed in these tables reflect two trends that were expected: (a) That increasing the posting exclusion frequency threshold, E, results in increased sensitivity. That is, the more comprehensive the index, the more sensitive the search. (b) That DASH Mode 4 (M4) is much more sensitive than DASH Mode 2 (M2), which was first established in Chapter 5. This is the single largest influence on search sensitivity.

Table 7.1: Sensitivity Results For Each Of The Ten Configurations Of DASH Mode 2 + NP3/NIX, And Posting Frequency Exclusion Threshold = $1.5\times$ Random Expectation.

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
DASH M2 Config 1, E=1.5	70.06	70.04	69.86	69.83	69.72
DASH M2 Config 2, E=1.5	70.62	70.61	70.42	70.40	70.29
DASH M2 Config 3, E=1.5	70.90	70.89	70.70	70.68	70.57
DASH M2 Config 4, E=1.5	71.42	71.41	71.22	71.20	71.09
DASH M2 Config 5, E=∞	83.83	83.82	83.62	83.56	83.47
DASH M2 Config 6, E=1.5	71.08	71.07	70.89	70.87	70.76
DASH M2 Config 7, E=1.5	71.25	71.23	71.05	71.03	70.92
DASH M2 Config 8, E=1.5	72.20	72.19	72.01	71.99	71.86
DASH M2 Config 9, E=1.5	72.20	72.19	72.01	71.99	71.86
DASH M2 Config 10, E=∞	83.80	83.78	83.58	83.49	83.39
Smith-Waterman	100	100	100	100	100
BLAST (Default)	82.04	81.64	78.69	74.97	66.58
BLAST (No Filter)	88.43	88.38	87.96	87.73	86.86
BLAST (Report Everything)	92.78	92.36	89.41	85.62	76.96
FASTA	66.41	65.59	65.36	65.28	65.25
BLAT	51.69	51.41	50.88	50.04	42.06
PatternHunter	46.22	45.58	44.59	43.67	40.05
CAFE	26.69	26.68	25.96	24.20	17.32

Table 7.2: Sensitivity Results For Each Of The Ten Configurations Of DASH Mode 2 + NP3/NIX, And Posting Frequency Exclusion Threshold = $2.5\times$ Random Expectation.

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
DASH M2 Config 1, E=2.5	75.69	75.69	75.50	75.47	75.39
DASH M2 Config 2, E=2.5	75.69	75.69	75.50	75.47	75.39
DASH M2 Config 3, E=2.5	75.97	75.97	75.78	75.75	75.67
DASH M2 Config 4, E=2.5	76.53	76.53	76.34	76.31	76.22
DASH M2 Config 5, E=∞	83.83	83.82	83.62	83.56	83.47
DASH M2 Config 6, E=2.5	75.77	75.75	75.56	75.53	75.43
DASH M2 Config 7, E=2.5	75.94	75.92	75.73	75.70	75.61
DASH M2 Config 8, E=2.5	76.83	76.82	76.62	76.60	76.50
DASH M2 Config 9, E=2.5	76.83	76.82	76.62	76.60	76.50
DASH M2 Config 10, E=∞	83.80	83.78	83.58	83.49	83.39
Smith-Waterman	100	100	100	100	100
BLAST (Default)	82.04	81.64	78.69	74.97	66.58
BLAST (No Filter)	88.43	88.38	87.96	87.73	86.86
BLAST (Report Everything)	92.78	92.36	89.41	85.62	76.96
FASTA	66.41	65.59	65.36	65.28	65.25
BLAT	51.69	51.41	50.88	50.04	42.06
PatternHunter	46.22	45.58	44.59	43.67	40.05
CAFE	26.69	26.68	25.96	24.20	17.32

Table 7.3: Sensitivity Results For Each Of The Ten Configurations Of DASH Mode 2 + NP3/NIX, And Posting Frequency Exclusion Threshold = $5.0\times$ Random Expectation.

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
DASH M2 Config 1, E=5	79.51	79.50	79.30	79.22	79.14
DASH M2 Config 2, E=5	79.55	79.55	79.34	79.27	79.19
DASH M2 Config 3, E=5	79.6	79.59	79.39	79.32	79.24
DASH M2 Config 4, E=5	80.13	80.12	79.92	79.88	79.8
DASH M2 Config 5, E=∞	83.83	83.82	83.62	83.56	83.47
DASH M2 Config 6, E=5	79.76	79.75	79.54	79.47	79.38
DASH M2 Config 7, E=5	79.95	79.94	79.73	79.67	79.57
DASH M2 Config 8, E=5	80.41	80.39	80.18	80.11	80.02
DASH M2 Config 9, E=5	80.41	80.39	80.18	80.11	80.02
DASH M2 Config 10, E=∞	83.80	83.78	83.58	83.49	83.39
Smith-Waterman	100	100	100	100	100
BLAST (Default)	82.04	81.64	78.69	74.97	66.58
BLAST (No Filter)	88.43	88.38	87.96	87.73	86.86
BLAST (Report Everything)	92.78	92.36	89.41	85.62	76.96
FASTA	66.41	65.59	65.36	65.28	65.25
BLAT	51.69	51.41	50.88	50.04	42.06
PatternHunter	46.22	45.58	44.59	43.67	40.05
CAFE	26.69	26.68	25.96	24.20	17.32

Table 7.4: Sensitivity Results For Each Of The Ten Configurations Of DASH Mode 2 + NP3/NIX, And Posting Frequency Exclusion Threshold = $10\times$ Random Expectation.

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
DASH M2 Config 1, E=10	81.70	81.70	81.50	81.41	81.33
DASH M2 Config 2, E=10	81.69	81.69	81.49	81.40	81.32
DASH M2 Config 3, E=10	81.74	81.73	81.53	81.44	81.36
DASH M2 Config 4, E=10	82.02	82.01	81.81	81.75	81.67
DASH M2 Config 5, E=∞	83.83	83.82	83.62	83.56	83.47
DASH M2 Config 6, E=10	82.12	82.11	81.91	81.82	81.72
DASH M2 Config 7, E=10	82.18	82.17	81.97	81.88	81.78
DASH M2 Config 8, E=10	82.39	82.38	82.17	82.08	81.99
DASH M2 Config 9, E=10	82.39	82.38	82.17	82.08	81.99
DASH M2 Config 10, E=∞	83.80	83.78	83.58	83.49	83.39
Smith-Waterman	100	100	100	100	100
BLAST (Default)	82.04	81.64	78.69	74.97	66.58
BLAST (No Filter)	88.43	88.38	87.96	87.73	86.86
BLAST (Report Everything)	92.78	92.36	89.41	85.62	76.96
FASTA	66.41	65.59	65.36	65.28	65.25
BLAT	51.69	51.41	50.88	50.04	42.06
PatternHunter	46.22	45.58	44.59	43.67	40.05
CAFE	26.69	26.68	25.96	24.20	17.32

Table 7.5: Sensitivity Results For Each Of The Ten Configurations Of DASH Mode 4 + NP3/NIX, And Posting Frequency Exclusion Threshold = $1.5\times$ Random Expectation.

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
DASH M4 Config 1, E=1.5	78.35	78.27	77.88	77.78	77.57
DASH M4 Config 2, E=1.5	79.22	79.14	78.75	78.65	78.35
DASH M4 Config 3, E=1.5	79.39	79.31	78.92	78.82	78.50
DASH M4 Config 4, E=1.5	79.67	79.59	79.20	79.11	78.88
DASH M4 Config 5, E=∞	91.94	91.88	91.53	91.38	90.91
DASH M4 Config 6, E=1.5	79.95	79.87	79.48	79.39	79.17
DASH M4 Config 7, E=1.5	80.12	80.03	79.64	79.55	79.34
DASH M4 Config 8, E=1.5	81.00	80.92	80.51	80.41	80.20
DASH M4 Config 9, E=1.5	81.00	80.92	80.51	80.41	80.20
DASH M4 Config 10, E=∞	91.94	91.87	91.52	91.37	90.92
Smith-Waterman	100	100	100	100	100
BLAST (Default)	82.04	81.64	78.69	74.97	66.58
BLAST (No Filter)	88.43	88.38	87.96	87.73	86.86
BLAST (Report Everything)	92.78	92.36	89.41	85.62	76.96
FASTA	66.41	65.59	65.36	65.28	65.25
BLAT	51.69	51.41	50.88	50.04	42.06
PatternHunter	46.22	45.58	44.59	43.67	40.05
CAFE	26.69	26.68	25.96	24.20	17.32

Table 7.6: Sensitivity Results For Each Of The Ten Configurations Of DASH Mode 4 + NP3/NIX, And Posting Frequency Exclusion Threshold = 2.5× Random Expectation.

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
DASH M4 Config 1, E=2.5	85.25	85.17	84.69	84.55	84.17
DASH M4 Config 2, E=2.5	84.65	84.58	84.21	84.08	83.75
DASH M4 Config 3, E=2.5	84.72	84.64	84.28	84.16	83.83
DASH M4 Config 4, E=2.5	85.12	85.05	84.69	84.56	84.23
DASH M4 Config 5, E=∞	91.94	91.88	91.53	91.38	90.91
DASH M4 Config 6, E=2.5	86.61	86.53	86.06	85.92	85.53
DASH M4 Config 7, E=2.5	86.69	86.61	86.14	86.00	85.61
DASH M4 Config 8, E=2.5	87.12	87.05	86.56	86.42	86.04
DASH M4 Config 9, E=2.5	87.12	87.05	86.56	86.42	86.04
DASH M4 Config 10, E=∞	91.94	91.87	91.52	91.37	90.92
Smith-Waterman	100	100	100	100	100
BLAST (Default)	82.04	81.64	78.69	74.97	66.58
BLAST (No Filter)	88.43	88.38	87.96	87.73	86.86
BLAST (Report Everything)	92.78	92.36	89.41	85.62	76.96
FASTA	66.41	65.59	65.36	65.28	65.25
BLAT	51.69	51.41	50.88	50.04	42.06
PatternHunter	46.22	45.58	44.59	43.67	40.05
CAFE	26.69	26.68	25.96	24.20	17.32

Table 7.7: Sensitivity Results For Each Of The Ten Configurations Of DASH Mode 4 + NP3/NIX, And Posting Frequency Exclusion Threshold = $5.0\times$ Random Expectation.

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
DASH M4 Config 1, E=5	88.94	88.86	88.51	88.38	88.00
DASH M4 Config 2, E=5	89.17	89.08	88.74	88.59	88.21
DASH M4 Config 3, E=5	89.20	89.13	88.79	88.64	88.27
DASH M4 Config 4, E=5	89.32	89.25	88.89	88.74	88.30
DASH M4 Config 5, E=∞	91.94	91.88	91.53	91.38	90.91
DASH M4 Config 6, E=5	89.83	89.75	89.39	89.25	88.86
DASH M4 Config 7, E=5	89.87	89.79	89.44	89.30	88.91
DASH M4 Config 8, E=5	89.99	89.91	89.54	89.41	89.03
DASH M4 Config 9, E=5	89.99	89.91	89.54	89.41	89.03
DASH M4 Config 10, E=∞	91.94	91.87	91.52	91.37	90.92
Smith-Waterman	100	100	100	100	100
BLAST (Default)	82.04	81.64	78.69	74.97	66.58
BLAST (No Filter)	88.43	88.38	87.96	87.73	86.86
BLAST (Report Everything)	92.78	92.36	89.41	85.62	76.96
FASTA	66.41	65.59	65.36	65.28	65.25
BLAT	51.69	51.41	50.88	50.04	42.06
PatternHunter	46.22	45.58	44.59	43.67	40.05
CAFE	26.69	26.68	25.96	24.20	17.32

Table 7.8: Sensitivity Results For Each Of The Ten Configurations Of DASH Mode 4 + NP3/NIX, And Posting Frequency Exclusion Threshold = $10\times$ Random Expectation.

Format	PatternHunter Variant Score				
	at 50%	at 75%	at 90%	at 95%	at 100%
DASH M4 Config 1, E=10	90.89	90.82	90.47	90.33	89.86
DASH M4 Config 2, E=10	90.95	90.89	90.53	90.39	89.92
DASH M4 Config 3, E=10	90.97	90.91	90.56	90.42	89.96
DASH M4 Config 4, E=10	90.94	90.87	90.53	90.37	89.89
DASH M4 Config 5, E=∞	91.94	91.88	91.53	91.38	90.91
DASH M4 Config 6, E=10	91.18	91.11	90.77	90.62	90.16
DASH M4 Config 7, E=10	91.19	91.13	90.79	90.65	90.19
DASH M4 Config 8, E=10	91.41	91.34	91.00	90.85	90.40
DASH M4 Config 9, E=10	91.41	91.34	91.00	90.85	90.40
DASH M4 Config 10, E=∞	91.94	91.87	91.52	91.37	90.92
Smith-Waterman	100	100	100	100	100
BLAST (Default)	82.04	81.64	78.69	74.97	66.58
BLAST (No Filter)	88.43	88.38	87.96	87.73	86.86
BLAST (Report Everything)	92.78	92.36	89.41	85.62	76.96
FASTA	66.41	65.59	65.36	65.28	65.25
BLAT	51.69	51.41	50.88	50.04	42.06
PatternHunter	46.22	45.58	44.59	43.67	40.05
CAFE	26.69	26.68	25.96	24.20	17.32

In addition to these expected trends, the search sensitivity scores improve as cooperative compression is more aggressively applied (Wilcoxon sign test yields $p \leq 0.0008$ for E=1.5 index with DASH M2). For example, Configurations 2, 3 and 4 which employ forward chained cooperative compression differ only in the increasing degree to which cooperative compression was applied. However, the statistical confidence that sensitivity was increased diminishes and eventually disappears as the indices become more thorough (i.e. as E increases). This is not surprising, as the more thorough the search, the less results will be missed, and therefore the smaller the potential sensitivity gain due to searching using a cooperative compressed index. However, as will be explained below, because cooperative compression results in a smaller index, it is possible to gain substantial sensitivity by using a more thorough index, while still requiring a smaller index than would be necessary for a less thorough search using an ordinary, i.e., non-cooperatively compressed, index.

Overall, reverse chained cooperative compression results in better sensitivity than forward chained cooperative compression — except when E=∞, where forward chained indexing gains a slight edge. Regardless of the preference for either forward or reverse chained cooperatively compressed indices, what is certain is that cooperative compression affords a modest gain in sensitivity. What has not been attempted here, but would be possible, is to use both forward and reverse chaining in the same search to obtain further sensitivity gains.

It will be noticed that the scores obtained with Configuration 1 (the negative control which does not use cooperative compression) with an exclusion threshold of E=1.5× random are slightly better than the equivalent results presented in Chapter 5 (see Table 5.5). Moreover, the comparison also shows that the results presented in this chapter have a much higher ratio of sensitivity scores when the PatternHunter Variant Score threshold is to require 100% of an alignment (right most column in the tables) versus when the threshold is to require only 50% of an alignment (left most column of results in the tables). Both of these are the result of minor changes made to the DASH algorithm between when the experiments in these chapters were performed. Also substantial sensitivity gains are realised by using

more thorough indices (E={2.5, 5.0, 10.0}$\times$ random expectation) in the experiments in this chapter in addition to the less thorough index (E=1.5$\times$random expectation) which was used in the experiments of Chapter 5.

Compared with the peer group of algorithms, DASH with NP3/NIX cooperative compression is now able to match or beat the sensitivity of every other algorithm when 90% or more of an alignment is required, and only falls slightly short of this achievement when the scoring threshold is relaxed to 50% of an alignment. This is reflected in Figure 7.4 where a selection of the DASH+NP3/NIX results are compared with the results of the peer group of algorithms.

7.8.2 Reduced Index Sizes

Tables 7.9, 7.10, 7.11 and 7.12 present the index sizes for the ten configurations, with each table using a different *k*-mer frequency threshold for posting exclusion in the index. Those thresholds are 1.5$\times$ random (Table 7.9), 2.5$\times$ random (Table 7.10), 5.0$\times$ random (Table 7.11) and 10$\times$ random expectation (Table 7.12). The time required to construct each index varied between 24 and 97 minutes using a RedHat Linux ES3 system with a 1.8 GHz AMD Opteron and 8 GB of RAM.

Not surprisingly, the general rule governing index size is, the lower the threshold for posting exclusion, the smaller the index. For the smallest index (Configuration 3, E=1.5), the index is less than 20% larger than the only lightly indexed format used by BLAST (6.05 bits per base for DASH+NP3/NIX versus 5.09 bits per base for BLAST). In all cases, the index is smaller than the 3,606 MB (15.43 bits per base) required for the FOLDDB index presented in Chapter 5. Thus, for a fixed index size budget, we see not only the direct sensitivity gains of cooperative compression, but also the opportunity for significant secondary gains by using a more thorough index.

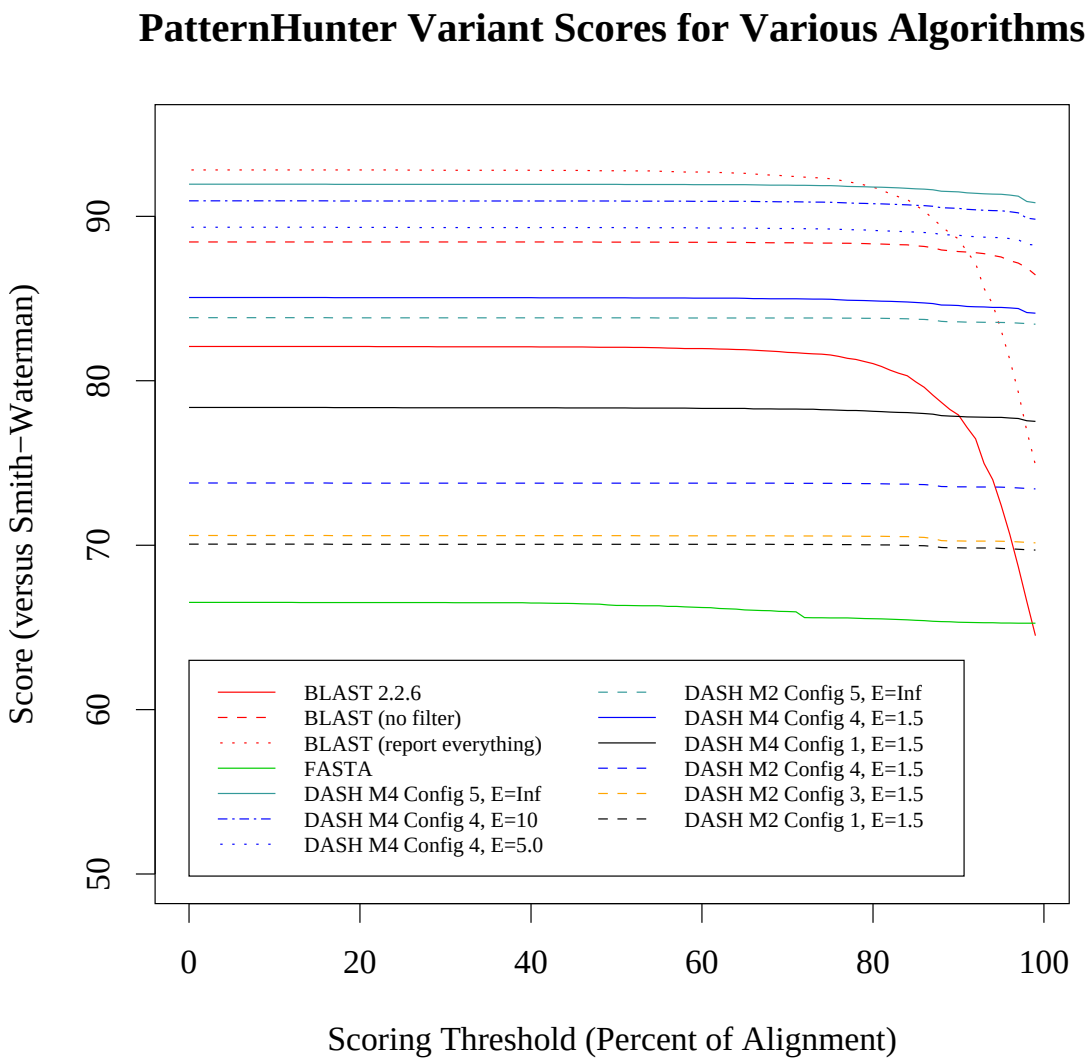

Figure 7.4: Pattern Hunter Variant Scores For Nucleic Acid Queries (Using The Human UniGene (Nucleic Acid) Database). CAFE, BLAT and PatternHunter all score lower than the results presented here, and are omitted from this graph for clarity. DASH obtains excellent sensitivity compared with the other algorithms, especially when the scoring threshold approaches 100% of the alignment.

Looking within each table now, we can observe the difference that the ten configurations make on index size. Recall that Configuration 1 is the negative control, where cooperative compression is not performed; Configurations 2 through 5 use forward-indexed cooperative compression and; Configurations 6 through 10 use reverse-indexed cooperative compression. Recall also, that Configurations 5 and 10 disable the cut-off threshold for posting exclusion, i.e., do not exclude stop words from the index. Thus, the threshold value is not used, and the indices generated for Configuration 5 and Configuration 10 do not differ with the value of E.

Table 7.13 shows the relative size of the smallest index size achieved for each value of E, versus the size of the negative control index (produced using Configuration 1). For all values of E, Configuration 3 (forward indexing cooperative compression, plus prefer inter-record references) produced the smallest index. That forward indexing performed better than either the negative control or reverse indexing is in line with expectation; forward indexing should always compress at least as well as reverse indexing. What comes as a surprise, is that Configuration 4 produced a larger index than Configuration 3, since Configuration 4 should have benefited from the larger number of inter-record references in the NP3 files produced using that configuration. This anomaly probably indicates that using a fixed sized per-posting rebate is not appropriate, especially when combined with a posting compression algorithm that is sensitive to clusters. The same phenomenon also occurs with Configurations 8, 9 and 10 (which use the fixed size per-posting rebate) versus Configuration 7 (which produces the smallest index that uses reverse direction cooperative compression).

The final observation is that the benefit of cooperative compression of the indices increased with the posting exclusion threshold. This makes sense, since it seems reasonable that very frequent k-mers should be more likely to form part of recurrent strings, precisely because they are more frequent. In other words, a more thorough index will index more redundant strings, and thus the resulting index should compress better. This may also explain why

the difference in compression between forward and reverse indexing widens as the posting exclusion threshold increases.

7.8.3 Increased Search Time

Tables 7.14, 7.15, 7.16, 7.17, 7.18, 7.19, 7.20 and 7.21 show the search times for DASH using NP3/NIX files. The search times of the peer group of algorithms are included to provide context and perspective.

Two issues are immediately apparent: (a) The search times are very slow compared with the FOLDDB results; and (b) The search times become slower, not faster, as cooperative compression is enabled. To shed light on these matters, Table 7.22 presents a break down of the time DASH+NP3/NIX spent performing various tasks for several configuration combinations. The time spent in each activity is unscaled, in order to show the absolute difference in time spent between negative control and cooperative compression. However, note that these times were necessarily obtained with DASH compiled to include profiling instrumentation. Thus, the search times presented here differ by some (hopefully constant) factor from the search speed results presented earlier, which were produced using a version of DASH compiled with compiler optimisations enabled.

7.8.3.1 NP3 And NIX Decompression Costs

Table 7.22 reveals that NP3 decompression is the single largest contributor to search time. Indeed, even if all other activities could somehow be avoided the NP3 decompression time would cause DASH to search more slowly than BLAST — this is despite the effort invested in making NP3 very fast to decompress. Note that the NP3 extraction time for Configuration 4 is much worse than for Configurations 1 through 3. As the only difference between Configurations 3 and 4 is the more aggressive use of inter-record redundancy records in the NP3 file, we must conclude that the assumption made in Chapter 6 that searches will

Table 7.9: Nucleic Acid Database And Index Sizes In Megabytes (MB) And Bits Per Base (B/B) (NIX E-value = 1.5).

Format	Bodies Only		Descriptions		Index		Total	
	MB	B/B	MB	B/B	MB	B/B	MB	B/B
Smith-Waterman* (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
BLAST (formatdb)	489	2.09	660	2.83	40	0.17	1,189	5.09
BLAT* (faToTwoBit)	630	2.70	-	-	1,088	4.66	1,718	7.36
PatternHunter** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
FASTA** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
NP3+NIX per Table A.11 (negative control)	166	0.71	87	0.37	1,771	7.58	2,023	8.66
NP3+NIX per Table A.12 (forward indexed cooperative compression)	166	0.71	87	0.37	1,181	5.06	1,434	6.14
NP3+NIX per Table A.13 (as above, but prefer inter-record references when not more expensive)	165	0.71	87	0.37	1,162	4.98	1,414	6.05
NP3+NIX per Table A.14 (as above, but optimising combined NP3+NIX size)	354	1.52	87	0.37	1,322	5.66	1,763	7.55
NP3+NIX per Table A.15 (as above, but no stop-word filtering)	354	1.52	87	0.37	2,420	10.36	2,860	12.25
NP3+NIX per Table A.16 (reverse indexed cooperative compression)	166	0.71	87	0.37	1,196	5.12	1,448	6.20
NP3+NIX per Table A.17 (as above, but prefer inter-record references when not more expensive)	166	0.71	87	0.37	1,183	5.07	1,435	6.14
NP3+NIX per Table A.18 (as above, but optimising combined NP3+NIX size)	354	1.52	87	0.37	1,281	5.49	1,722	7.37
NP3+NIX per Table A.19 (as above, but maximise distinct source material referenced by inter-record references)	354	1.52	87	0.37	1,281	5.49	1,722	7.37
NP3+NIX per Table A.20 (as above, but no stop-word filtering)	354	1.52	87	0.37	3,086	13.21	3,527	15.10
CAFE*** (CAFE Index)	496	2.12	102	0.44	6,961	29.79	7,634	32.67

* Indicates that program maintains an index in RAM, and that the database format contains both sequence bodies and descriptions (BLAT).

** Indicates that algorithm indexes during searching (PatternHunter and FASTA), or does not use an index (Smith-Waterman).

*** Indicates that multiple small indices were used instead of one large index, due to technical difficulties (CAFE).

Table 7.10: Nucleic Acid Database And Index Sizes In Megabytes (MB) And Bits Per Base (B/B) (NIX E-value = 2.5).

Format	Bodies Only		Descriptions		Index		Total	
	MB	B/B	MB	B/B	MB	B/B	MB	B/B
Smith-Waterman* (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
BLAST (formatdb)	489	2.09	660	2.83	40	0.17	1,189	5.09
BLAT* (faToTwoBit)	630	2.70	-	-	1,088	4.66	1,718	7.36
PatternHunter** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
FASTA** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
NP3+NIX per Table A.11 (negative control)	166	0.71	87	0.37	2,538	10.87	2,791	11.95
NP3+NIX per Table A.12 (forward indexed cooperative compression)	166	0.71	87	0.37	1,602	6.86	1,855	7.94
NP3+NIX per Table A.13 (as above, but prefer inter-record references when not more expensive)	165	0.71	87	0.37	1,572	6.73	1,824	7.81
NP3+NIX per Table A.14 (as above, but optimising combined NP3+NIX size)	354	1.52	87	0.37	1,828.7	7.83	2,270	9.72
NP3+NIX per Table A.15 (as above, but no stop-word filtering)	354	1.52	87	0.37	2,420	10.36	2,860	12.25
NP3+NIX per Table A.16 (reverse indexed cooperative compression)	166	0.71	87	0.37	1,847	7.91	2,099	8.99
NP3+NIX per Table A.17 (as above, but prefer inter-record references when not more expensive)	166	0.71	87	0.37	1,830	7.84	2,082	8.91
NP3+NIX per Table A.18 (as above, but optimising combined NP3+NIX size)	354	1.52	87	0.37	1,955	8.37	2,396	10.26
NP3+NIX per Table A.19 (as above, but maximise distinct source material referenced by inter-record references)	354	1.52	87	0.37	1,955	8.37	2,396	10.26
NP3+NIX per Table A.20 (as above, but no stop-word filtering)	354	1.52	87	0.37	3,086	13.21	3,527	15.10
CAFE*** (CAFE Index)	496	2.12	102	0.44	6,961	29.79	7,634	32.67

* Indicates that program maintains an index in RAM, and that the database format contains both sequence bodies and descriptions (BLAT).

** Indicates that algorithm indexes during searching (PatternHunter and FASTA), or does not use an index (Smith-Waterman).

*** Indicates that multiple small indices were used instead of one large index, due to technical difficulties (CAFE).

Table 7.11: Nucleic Acid Database And Index Sizes In Megabytes (MB) And Bits Per Base (B/B) (NIX E-value = 5.0).

Format	Bodies Only		Descriptions		Index		Total	
	MB	B/B	MB	B/B	MB	B/B	MB	B/B
Smith-Waterman* (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
BLAST (formatdb)	489	2.09	660	2.83	40	0.17	1,189	5.09
BLAT* (faToTwoBit)	630	2.70	-	-	1,088	4.66	1,718	7.36
PatternHunter** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
FASTA** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
NP3+NIX per Table A.11 (negative control)	166	0.71	87	0.37	3,138	13.44	3,391	14.52
NP3+NIX per Table A.12 (forward indexed cooperative compression)	166	0.71	87	0.37	1,898	8.13	2,151	9.21
NP3+NIX per Table A.13 (as above, but prefer inter-record references when not more expensive)	165	0.71	87	0.37	1,860	7.96	2,111	9.04
NP3+NIX per Table A.14 (as above, but optimising combined NP3+NIX size)	354	1.52	87	0.37	2,210	9.50	2,650	11.35
NP3+NIX per Table A.15 (as above, but no stop-word filtering)	354	1.52	87	0.37	2,420	10.36	2,860	12.25
NP3+NIX per Table A.16 (reverse indexed cooperative compression)	166	0.71	87	0.37	2,576	11.03	2,829	12.11
NP3+NIX per Table A.17 (as above, but prefer inter-record references when not more expensive)	166	0.71	87	0.37	2,561	10.97	2,813	12.04
NP3+NIX per Table A.18 (as above, but optimising combined NP3+NIX size)	354	1.52	87	0.37	2,658	11.38	3,098	13.27
NP3+NIX per Table A.19 (as above, but maximise distinct source material referenced by inter-record references)	354	1.52	87	0.37	2,658	11.38	3,098	13.27
NP3+NIX per Table A.20 (as above, but no stop-word filtering)	354	1.52	87	0.37	3,086	13.21	3,527	15.10
CAFE*** (CAFE Index)	496	2.12	102	0.44	6,961	29.79	7,634	32.67

* Indicates that program maintains an index in RAM, and that the database format contains both sequence bodies and descriptions (BLAT).

** Indicates that algorithm indexes during searching (PatternHunter and FASTA), or does not use an index (Smith-Waterman).

*** Indicates that multiple small indices were used instead of one large index, due to technical difficulties (CAFE).

Table 7.12: Nucleic Acid Database And Index Sizes In Megabytes (MB) And Bits Per Base (B/B) (NIX E-value = 10.0).

Format	Bodies Only		Descriptions		Index		Total	
	MB	B/B	MB	B/B	MB	B/B	MB	B/B
Smith-Waterman* (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
BLAST (formatdb)	489	2.09	660	2.83	40	0.17	1,189	5.09
BLAT* (faToTwoBit)	630	2.70	-	-	1,088	4.66	1,718	7.36
PatternHunter** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
FASTA** (FASTA ASCII)	1,886	8.06	470	2.04	-	-	2,356	10.10
NP3+NIX per Table A.11 (negative control)	166	0.71	87	0.37	3,349	14.34	3,602	15.42
NP3+NIX per Table A.12 (forward indexed cooperative compression)	166	0.71	87	0.37	1,985	8.50	2,237	9.58
NP3+NIX per Table A.13 (as above, but prefer inter-record references when not more expensive)	165	0.71	87	0.37	1,942	8.32	2,194	9.39
NP3+NIX per Table A.14 (as above, but optimising combined NP3+NIX size)	354	1.52	87	0.37	2,337	10.01	2,778	11.89
NP3+NIX per Table A.15 (as above, but no stop-word filtering)	354	1.52	87	0.37	2,420	10.36	2,860	12.25
NP3+NIX per Table A.16 (reverse indexed cooperative compression)	166	0.71	87	0.37	2,875	12.31	3,127	13.39
NP3+NIX per Table A.17 (as above, but prefer inter-record references when not more expensive)	166	0.71	87	0.37	2,862	12.25	3,114	13.33
NP3+NIX per Table A.18 (as above, but optimising combined NP3+NIX size)	354	1.52	87	0.37	2,927	12.53	3,368	14.42
NP3+NIX per Table A.19 (as above, but maximise distinct source material referenced by inter-record references)	354	1.52	87	0.37	2,927	12.53	3,368	14.42
NP3+NIX per Table A.20 (as above, but no stop-word filtering)	354	1.52	87	0.37	3,086	13.21	3,527	15.10
CAFE*** (CAFE Index)	496	2.12	102	0.44	6,961	29.79	7,634	32.67

* Indicates that program maintains an index in RAM, and that the database format contains both sequence bodies and descriptions (BLAT).

** Indicates that algorithm indexes during searching (PatternHunter and FASTA), or does not use an index (Smith-Waterman).

*** Indicates that multiple small indices were used instead of one large index, due to technical difficulties (CAFE).

Table 7.13: Relative Size Of Most Compact Index Versus Negative Control.

Threshold	Negative Control Size	Smallest Size	Configuration #	Reduction
1.5×	1,771 MB	1,162 MB	3	34%
2.5×	2,539 MB	1,572 MB	3	38%
5.0×	3,138 MB	1,860 MB	3	41%
10×	3,349 MB	1,942 MB	3	42%

either involve all or none of a set of records that are chained in an NP3 file is flawed, or at least has limits. In comparison, the handling of NIX structures turns out to be relatively computationally light, with the time cost being directly proportional to the number of postings being handled, thus affirming the decision to use the Fast Interpolative Coder. The best hope for reducing the NP3 decompression time would seem to be to amortise it over multiple queries, by performing searches in batches of perhaps 10 to 100.

7.8.3.2 Time Spent Performing Dynamic Programming, Discovering HSPs, And Translating HSPs

An unexpected result of using cooperative compression is that the time spent performing dynamic programming actually decreased compared to the negative control (Configuration 2 versus Configuration 1). What was expected, was that since cooperative compression resulted in the discovery of more HSPs, that more dynamic programming would be required to process them. Thus, even the modest reduction in dynamic programming effort that was observed came as a complete surprise. The most likely explanation of this effect is that many of extra HSPs that were discovered due to HSP translation were in regions that would have otherwise been subjected to dynamic programming. By finding an HSP in such a region, DASH's alignment assembly algorithm would have gained an additional piece of information, and the dynamic programming region would be divided into two smaller regions at each end of the extra HSP, as illustrated in Figure 7.5. Since dynamic programming has quadratic time cost with respect to the length of the region, such a divi-

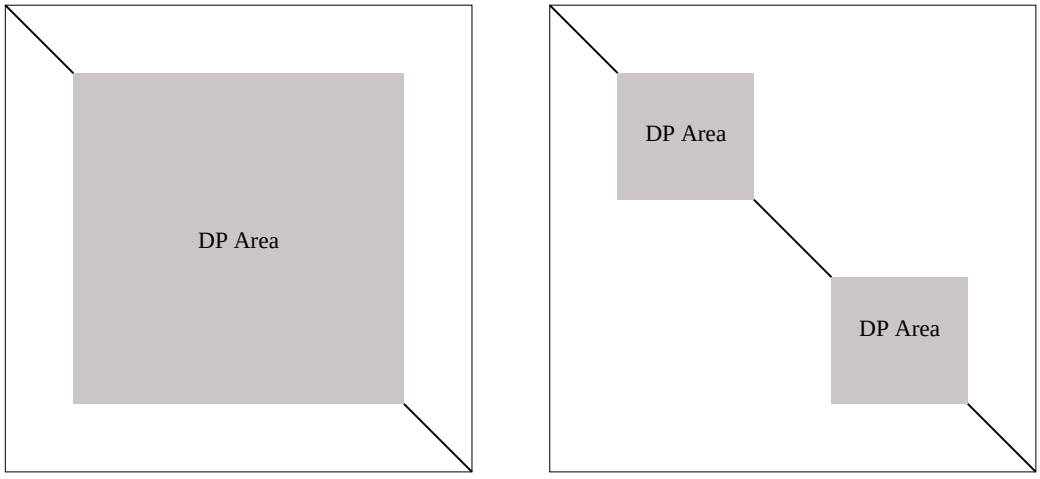

Figure 7.5: How Finding Extra HSPs Can Reduce Dynamic Programming Time. In (a) the DP region covers a large area, and so will require a long time to search. In (b) an extra HSP is introduced that allows the DP area to be divided into two much smaller regions, which together can be searched in much less time than the single large area.

sion would yield substantial time savings. Eventually, as cooperative compression is more aggressively applied to discover more HSPs, this effect is swamped by the additional dynamic programming required to service those HSPs that do correspond to new alignments.

Turning to HSP discovery time (including the time spent translating HSPs in Configurations 2 through 4 where cooperative compression is enabled), it is observed that cooperative compression results in significantly increased HSP discovery times, in contrast to the expectation that HSP discovery time would reduce. It must be concluded that the time overhead incurred in attempting to translate HSPs, on average, greatly out weighs the time saved by not independently discovering each member of a set of identical HSPs. A contributing factor is that whereas in the original DASH algorithm an O(1) algorithm could be employed to prevent the repeated discovery of already located HSPs, no such algorithm was found that could be applied to HSP translation. Thus a costly tree search was required in order to prevent the repeated discovery of translated HSPs. Moreover, the tree search could not be avoided, because to do so would result in the repeated discovery of HSPs —

which is exactly we were trying to avoid. Thus, it is this HSP translation overhead, together with that of NP3 record decompression, that cause such slow searching.

One opportunity that has not been explored here is the re-use of dynamic programming effort in the case where HSPs are translated. Provided that the overhead of determining when such re-use can occur is not excessive, then it may be possible to reduce the total dynamic programming effort incurred. However, as the experience with attempting to re-use HSP discovery effort has shown, it is by no means trivial to obtain the efficiency required to reduce search times. None the less, the relatively high cost of performing dynamic programming compared with HSP discovery suggests that it should be easier to reduce dynamic programming time than to reduce HSP discovery time.

7.8.4 Cooperative Compression Of A Less Redundant Database

Finally, Table 7.23 shows the effect of cooperative compression on the database and index of a database with less redundancy, the Human Genome. Results for both the unfolded (Configuration 1) and folded (Configuration 3) are shown, using a posting frequency exclusion threshold of 1.5× random expectation. In this case, the index size is reduced by only 3%, much less than the 30% – 40% obtained when compressing the more redundant Human UniGene (nucleic acid) database. This is due to the limited ability of NP3 to find recurrent strings in unsorted databases and those with low redundancy. Thus, while NP3 already decompresses too slowly to facilitate competitive search times, it is worthwhile to attempt cooperative compression of the Human Genome database using GeNML+NIX. This is because it allows us to determine if the current best performing DNA compression algorithm can discover sufficient redundant strings to make cooperative compression feasible on an typical and unsorted genomic database, such as the Human Genome.

7.8.5 Comparison Of GeNML And NP3

Table 7.24 summarises the decompression speed results for the original NP3 byte-aligned codec and the GeNML codec developed in this chapter. These are compared against the general purpose compressor GZIP. Unsurprisingly, the byte-aligned codec is much faster than the GeNML codec, reflecting the computational cost of the Arithmetic Coding required for the GeNML codec. The NP3 byte-aligned codec is also slightly faster than GZIP. However, as discovered in Chapter 7, this is not fast enough to support fast searching of nucleic acid databases, and therefore we must conclude that, given that the GeNML codec is about ten times slower, that it is much too slow to support fast searching of nucleic acid databases.

7.8.6 Compressed Database And Index Sizes

Turning to compression effectiveness, Table 7.25 presents the compressed database sizes of the NP3 byte-aligned and GeNML codecs versus the canonical GeNML algorithm. For the two NP3 codecs, results are presented first with cooperative compression disabled (marked "(a)"), and with cooperative compression enabled (marked "(b)"). For the NP3 codecs, the size of the NIX index structures is also listed, and when cooperative compression is enabled, the percentage of postings that were removed from the index by cooperative compression are also listed. Performance of the canonical GeNML algorithm is listed only for those databases where results were published by Korodi and Tabus (2005). Consequentially, no canonical GeNML results are given for the Human UniGene database.

By comparing the performance of the NP3 GeNML codec to the canonical GeNML algorithm, it is apparent that the NP3 GeNML codec does not perform as well as the canonical GeNML algorithm. This is to be expected for the reasons detailed earlier in this chapter, particularly the insertion of regular synchronisation points, and the reduction in effective

window size imposed by database partitioning. However, it is noteworthy that only a small amount of compression is sacrificed.

For the de facto DNA Compression Corpus and the Human Genome databases, we observe that the GeNML codec compresses the sequence bodies more compactly than does the byte-aligned codec. This is in line with expectation — the trade off being greatly reduced decompression speed. However, for the Human UniGene database we discover that the NP3 byte-aligned codec performs slightly better than the NP3 GeNML codec. Two possible contributors come to mind. First, the NP3 byte-aligned codec was developed using the Human UniGene Database as its test input. Therefore, it is not surprising to discover that it performs particularly well when compressing that database. Thus, the byte-aligned codec results may be better than can normally be expected. Second, the GeNML algorithm was designed for optimal performance with "typical" DNA data. That is, the GeNML algorithm contains few specific features aimed at efficiently compressing the many long well preserved recurrent strings that occur in close proximity to one another in the Human UniGene database. In particular, the way that GeNML encodes data in relatively small blocks, each of which contains some addressing overhead, is probably sufficient to explain the difference — thus explaining the observation that the relative performance of the NP3 GeNML codec for the Human UniGene is worse (compared to the NP3 byte-aligned codec) than can be expected for "typical" DNA data.

Turning now to cooperative compression performance, we find the same trends occurring as for body compression: Because the GeNML algorithm is more able to efficiently compress approximate matches than the NP3 byte-aligned compression algorithm, the NP3 GeNML codec is able to remove several times more postings from the index than the NP3 byte-aligned codec. This translates into substantially smaller indices, with overall savings of about two bits per base for both the index of the de facto DNA Compression Corpus, and the index of the Human Genome database, in both cases yielding a reduction in size of about 15%. Note that this saving in index size (about two bits per base) is as large as the

best savings that can ever be made through compressing the sequence bodies, since they require only two bits per base to encode directly. Moreover, the 15% reduction in index size using the NP3 GeNML codec compares favourably to the 3% obtained with the NP3 byte-aligned codec. Thus, we find that cooperative compression can obtain useful reductions in index size when dealing with a typical nucleic acid database, when coupled with the current state of the art in DNA compression algorithms.

In the case of compressing the index of the Human UniGene database we discover that, as was the case for sequence body compression of the Human UniGene database, the NP3 byte-aligned codec out performs the NP3 GeNML codec. Here, the issue seems to be that the NP3 GeNML codec does not remove as many postings from the index (see the right hand column in Table 7.25), and thus is condemned to produce a larger index. This is because the NP3 GeNML codec seeks to optimise the size of the NP3 file, not the combined NP3 and NIX file pair. Thus, whenever it will save some bits, the GeNML algorithm will use Arithmetic Coding instead of referencing an approximate repeat to encode a block. However, it is only by referencing repeats that cooperative compression can reduce the index size. Since the index is several times larger than the compressed database, using Arithmetic Coding to obtain the greatest space savings for the database, may cause the index to be much larger, and so the GeNML algorithm may end up costing space overall. This could be addressed by implementing a similar scheme to that used in the NP3 byte-aligned codec, where the combined size of the NP3 and NIX file pair is optimised, although as discovered with the NP3 byte-aligned codec, this is not as simple as merely instituting a fixed rebate based on the average cost of storing an index posting.

7.8.7 Effect Of Query Length On Search Time

In order to assess the impact of query length on search time in the DASH+NP3/NIX environment it is helpful to break down the search time into separate components for dynamic

programming extension of alignments, HSP discovery, NIX index reconstruction and NP3 decompression. Figure 7.6 shows that dynamic programming increases and NIX index retrieval times increase more slowly with increasing query length compared to the NIX and NP3 retrieval activities. NP3 decompression is the most time consuming activity overall, while NIX index retrieval experiences the largest proportional growth.

It is the time spent in the NIX and NP3 activities that are of particular interest. The linear growth of NIX index retrieval time is in line with expectation, since the longer the query the more index columns must be extracted. The relatively slow proportional growth of NP3 decompression time is also expected, as it is only the posting of additional sequences that result in additional NP3 decompression time. Moreover, the time that can be required for NP3 decompression is bounded by the number of sequences in the database. Thus NP3 decompression time must eventually be logarithmic with respect to total query length as the proportion of decompressed sequences approaches 100%. Figure 7.6 shows some hints of this logarithmic behaviour in the way that the NP3 decompression time increases more quickly at the shorter end of query lengths than it does at the longer end.

Together, these factors indicate that the DASH search system could be used to service batches to effectively amortise the NP3 decompression time over many queries, with only a linear increase in search time due to the NIX index retrieval and other activities. In this way the space benefits of the NP3/NIX data structures can be realised without incurring a penalty in the form of excessive search time. Critically, the median time for each activity grows only linearly or logarithmically with respect to query length.

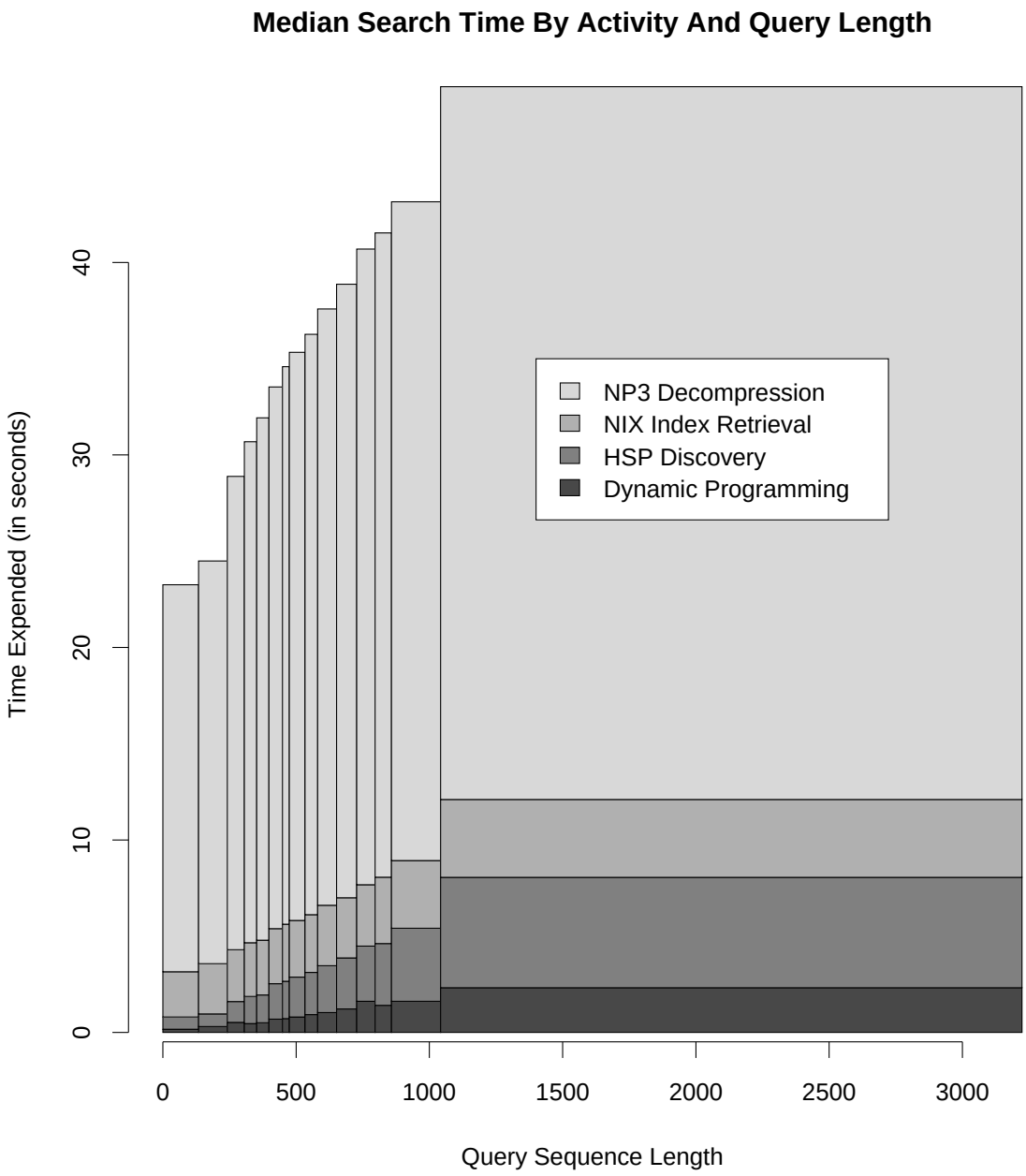

Figure 7.6: DASH+NP3/NIX Search Time Versus Query Length: Mode 2; Human UniGene Database; Configuration 3 (Forward Indexing).
Each bar represents 13 of the 200 queries. Median times are used in all cases.

7.9 Discussion

7.9.1 Analysis Of Performance With Disk Based Index

The previous material has assumed that the search process occurs with the database and index resident in RAM at all times. It is in that context that the excessive search time is detrimental. Let us briefly consider the performance of an NP3/NIX compressed database and index ensemble versus a FOLDDB uncompressed database and index ensemble. We will assume that both are stored on a disk system that delivers a sustained data stream of 100 MB per second. The resulting characteristics are then compared with those of BLAST.

7.9.1.1 Analysis Of DASH With FOLDDB And NP3/NIX

We know from Chapter 5 that a FOLDDB index and database ensemble for the Human UniGene nucleic acid database constructed using $E = 1.5\times$ random expectation will be 3,606 MB (Table 5.1). Further, we know that the average search time of that database using DASH M2 is 1.582 seconds. Thus, the total search time, with the FOLDDB index and database ensemble stored on disk, will be 3,606 MB $\div$ 100 MB per second = 36.06 seconds. The mean search time of 1.582 seconds (Table 5.3) is less than the time taken to read the data from disk, and so we can assume that the search occurs while the data stream is being read.

Considering now the NP3/NIX ensemble for the Human UniGene nucleic acid database constructed using $E = 1.5\times$ random expectation and using Reverse Indexing, we know from Table 7.9 the files will be 1,448 MB in size. We also know from Table 7.14 that the mean search time is 13.86 seconds. Thus, the total search time, with the NP3/NIX ensemble stored on disk, will be 1,448 MB $\div$ 100 MB per second = 14.5 seconds. Again, the search time is less than the time taken to read the data from disk, and so we can assume that the search occurs while the data stream is being read.

Thus we find that, even though searching an NP3/NIX compressed database and index ensemble is much slower than searching a FOLDDB formatted database and index ensemble, if the data must be stored on disk, then NP3/NIX is preferable, and that overall search time can be better than halved. This is particularly advantageous in, for example, the scenario where a workstation with relatively limited memory is used search a large collection, with the secondary benefit that the reduced index size may make it possible to fit the NP3/NIX compressed database into RAM.

7.9.1.2 The Beneficial Effect Of Partitioned Data

However, because the DASH, NP3 and NIX algorithms are designed around a partitioned database, it is possible to obtain better query throughput. This is by searching for batches of queries. Because the database and index are partitioned, with each partition small enough to fit in the memory of even a modest computer, each partition need be read only once for a whole batch of queries. This is a significant advantage over monolithic database structures, because random disk access is completely replaced by sequential disk access, which is much faster. Thus the disk delay can be divided by the number of queries in the batch.

Moreover, if each partition is extracted only once, the NP3 decompression time can be paid only once per batch instead of once per query. As Table 7.22 shows, the NP3 decompression time accounts for between 79% and 86% for DASH M2. This also effectively deals with the problem of unbound computational cost when evaluating recurrence chains, as each partition can be fully resident in RAM when being searched. Finally, because the database is formed into a number of approximately equal sized partitions that are searched independently, search time will scale linearly with database size.

7.9.1.3 Comparison Of Batched DASH Versus NCBI-BLAST

Thus, assuming that a batch of 20 queries were processed instead of a single query, the disk time would remain constant. The search time would increase by no more than the time taken for the NIX, HSP, DP and X stage of the 19 additional queries. This would be no more than $19 \times (100\% - 79\%) \times mean search time$ = $19 \times 21\% \times 13.86$ seconds = 55.30 seconds. Thus the total search time for all 20 queries would be 13.86 seconds + 55.30 seconds = 69.16 seconds, or an amortised 3.46 seconds per query. This is almost 3 times faster than NCBI-BLAST when using a memory resident database.

7.10 Conclusions

In conclusion, it has been shown that by using cooperative compression, it is possible to reduce the compressed index of a sorted moderately redundant nucleic acid index by 34% – 42%, without requiring that the database contain wholly redundant records. This compression result is accompanied by an increase in sensitivity.

These achievements of cooperative compression come at a significant computational cost, primarily as a result of: (a) the time taken to decompress sequences from an NP3 compressed database (which might be possible to amortise over multiple searches); and (b) that in the current implementation it takes more time to discover an HSP via translation using recurrence records, than it does to discover it normally. The GeNML algorithm, although compressing slightly better in some cases, is around ten times slower to decompress than the NP3 byte-aligned coding scheme, precluding it from being useful in the context of sequence search and alignment. Thus the decision to create the fast but slightly space inefficient NP3 byte-aligned coding scheme has been vindicated.

Where the database must be stored on disk the NP3/NIX cooperatively compressed index and database ensemble can reduce the data size to the point where the savings in data trans-

fer time more than compensate for the additional time spent decompressing and searching, resulting in a faster search overall. Moreover, if batches of queries are processed, then the NP3/NIX decompression time would be amortised to the point where DASH+NP3/NIX operating on an on-disk database could search faster than NCBI-BLAST with a memory resident database, with DASH requiring search time proportional to the product of the database size and sum of the query lengths.

In summary, it has been shown that cooperative compression can work, and can simultaneously deliver improved search sensitivity, reduced data sizes and reduced overall search times. Together, these results provide evidence for the thesis of this dissertation. This evidence is in addition to that found in Chapter 5, where cooperative compression was shown to be effective for databases where whole records were duplicated.

Nonetheless, there are significant efficiency hurdles that must be overcome for it to be useful with unsorted databases that do not contain large numbers of wholly redundant records, or when performing few queries on a memory based database. Indeed, whereas compressed indices could be shrunk by around 34% – 42% for sorted moderately redundant nucleic acid databases, reductions of only 3% – 15% were achieved on the unsorted Human Genome database. Nonetheless, index space savings of at least two bits per base were possible for each database considered, thus confirming that cooperative compression can save more space than DNA compression, which can never save more than two bits per base. Moreover, this problem of poor performance when compressing unsorted databases has been substantially addressed by the recent development of an algorithm that allows for sorting a genomic database in O(n) time (Bernstein and Cameron 2006).

7.11 Future Directions

7.11.1 Sorting Databases

The recent publication of a variant of the SPEX algorithm that allows for sorting genomic databases in approximately O(n) time presents a significant opportunity to generalise the attractive compression performance that NIX can obtain with a sorted database to any database with sufficient intrinsic internal redundancy, by making the sorting step practicable for all collection sizes.

7.11.2 Improving Search Efficiency

The logical next course of action is to address the inefficiency of the HSP translation, so that, on average, HSP translation is at least as efficient as HSP discovery. Specifically, there is an opportunity to produce and implement an O(1) algorithm for determining whether a given translation would point to an already discovered HSP, and thus can be skipped. If that can be achieved, then in order to further improve sensitivity, it may be worth exploring attempting translation in both the forward and reverse directions when searching a forward chained cooperatively compressed index. Also, attention should be given to reusing dynamic programming effort expended for an HSP for any translations of that HSP.

7.11.3 Avoiding NP3 Decompression Time

It may also be possible to increase the value of search time by side-stepping the decompression of the NP3 compressed database by replacing the NP3 file with a direct 2-bit or similar representation (such as CINO (Williams and Zobel 1997a)), and storing the recurrence records in a dedicated data structure. If this can be done such that the recurrence records and the cooperatively compressed index are smaller than the equivalent non-cooperatively

compressed index, then it may be possible to complete with existing nucleic acid search algorithms.

7.11.4 Presenting Relationships Among Search Results

It would be a valuable exercise to consider how to use the recurrence record information made available by cooperative compression to accurately and succinctly communicate the relationships and equivalences in the results of a given search. Of course, such techniques could be applied to the results of any search algorithm, independent of whether it uses the methods described here.

7.11.5 Improving Compression Performance By Using Dissimilar Regressors

An opportunity exists for improving the compression of the canonical GeNML algorithm by using dissimilar regressors. That is, to model one record by comparing it against another that shares no bases in common with it. This counter intuitive scheme works because if you know for every base in some DNA sequence you are encoding, that the base cannot be one particular one of the four nucleotides (since the string you are encoding and the regressor are known to not coincide at any position, and you know the base at any given position of the dissimilar regressor), then that base must be one of the remaining three nucleotides. For example, if you are encoding `C`, and know that it is not `A`, then you need only encode a choice between `C`, `G` and `T`. In other words, by knowing what the sequence you are encoding is not, its entropy has been reduced by 25%. If the dissimilar regressor has only a few bases in common with the sequence being encoded, i.e., are mostly dissimilar, then the entropy may still be reduced, but by a lesser degree.

The challenges in implementing this scheme are several. First, it must be possible to efficiently search for dissimilar DNA sequences, a subject which has been partially explored, but only for the restricted domain of designing oligonucleotides that do not bind to a given piece of DNA (Abbasi and Sengupta 1997). Second, given that the maximum compression possible is 25%, if space is to be saved, then the address of the dissimilar block must be able to be encoded in relatively few bits, an issue that is analysed in the following paragraphs.

For block length n, the probability of a completely dissimilar regressor block, that is one in which none of the bases correspond (for now, we ignore the contribution of mostly dissimilar regressors), is $(\frac{3}{4})^n$. For $n = 32$, this equates to $p = 1.004524 \times 10^{-4}$, for $n = 56$, $p = 1.008 \times 10^{-7}$, while for $n = 120$ it is $p = 1.017 \times 10^{-15}$. Given that for a window length of m, and provided that $m \gg n$, there are approximately $c = 2 \times m$ regressors (m forward ones, and m palindromic ones). Thus the probability of finding at least one dissimilar regressor in a window of length n is $p_d = 2np$.

Thus for window sizes in the mega-base range, it is not unreasonable to expect to find dissimilar regressors for modest block lengths. The difficulty is whether the address field to reference the dissimilar regressor exceeds the saving in encoding length: The space saving will be $n \times (-\log_2 \frac{1}{4} + \log_2 \frac{1}{3}) = n \times (2 - 1.585) = 0.415n$ bits. However, the address takes $\lceil 1 + \log_2 m \rceil$ bits to encode the direction and position of the regressor. Thus, a net saving requires $0.415n > \lceil 1 + \log_2 m \rceil$, and thus $n > 2.41 \lceil 1 + \log_2 m \rceil$.

In short, the block length must be about 2.5 times longer than the address length necessitated by the window size. Assuming a window size of $10^6 \approx 2^{20}$ bases, this would require a block length of at least $2.41 \times 21 = 50.61$ bases. Since block sizes must be a multiple of eight, the minimum block size is 56 bases, which would give a net saving of 2.24 bits for each block. The probability of finding a completely dissimilar regressor of this length was previously found to be $p_d = p \times 2 \times n = 1.008 \times 10^{-7} \times 2 \times 2^{20} = 0.211$. Thus, ap-

proximately one in five 56 bit blocks could be expected to be compressed in this way. The overall affect would be to reduce the worst case compression by $0.211 \times 2.24 = 0.473$ bits.

This method has the advantage that it has the potential to compress those blocks that cannot be profitably compressed in any other way. However, the substantial cost of finding the dissimilar regressor blocks (the most similar algorithm in the literature (Abbasi and Sengupta 1997) requires at least O($n \log n$) time), combined with the marginal returns, suggests that it would be of questionable value; Korodi and Tabus almost certainly made the correct decision to focus their efforts on the simultaneously more efficient and effective Constrained Normalisation. Nonetheless, the opportunity exists to slightly improve the compression of the GeNML algorithm by using dissimilar regressors. However, this falls outside of the scope of this dissertation.

Table 7.14: Comparison Of Nucleic Acid Search Speed (Using The Human UniGene (Nucleic Acid) Database), DASH M2, E=1.5.

Format	Search Time (seconds)			Search Time
	mean	median	total	(×BLAST)
DASH M2 Config 1 (unfolded), E=1.5	13.52	13.13	2,703.27	1.38
DASH M2 Config 2, E=1.5	19.74	19.48	3,948.20	2.02
DASH M2 Config 3, E=1.5	21.40	21.10	4,280.86	2.19
DASH M2 Config 4, E=1.5	67.39	59.89	13,478.57	6.88
DASH M2 Config 5, E=1.5	71.81	62.82	14,362.75	7.33
DASH M2 Config 6, E=1.5	13.86	13.43	2,772.85	1.42
DASH M2 Config 7, E=1.5	14.94	14.41	2,988.20	1.53
DASH M2 Config 8, E=1.5	45.36	44.34	9,071.25	4.63
DASH M2 Config 9, E=1.5	45.35	44.44	9,069.62	4.63
DASH M2 Config 10, E=1.5	60.21	45.78	12,041.44	6.15
Smith-Waterman	16,259.14	14,069.69	3,251,827.13	1,660.56
NCBI-BLAST 2.2.6 (Default)	9.79	9.40	1,958.27	1.00
NCBI-BLAST 2.2.6 (No Filter)	21.37	10.16	4,274.56	2.18
NCBI-BLAST 2.2.6 (Report Everything)	49.65	11.27	9,929.06	5.07
BLAT*	2.10	2.07	420.75	0.21
PatternHunter**	78.37	78.61	15,673.57	8.00
FASTA	500.24	506.14	100,048.54	51.09
CAFE***	1,673.37	1,537.97	334,673.07	170.90

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 7.15: Comparison Of Nucleic Acid Search Speed (Using The Human UniGene (Nucleic Acid) Database), DASH M2, E=2.5.

Format	Search Time (seconds)			Search Time (×BLAST)
	mean	median	total	
DASH M2 Config 1 (unfolded), E=2.5	14.37	14.09	2,874.52	1.47
DASH M2 Config 2, E=2.5	19.63	19.27	3,925.80	2.00
DASH M2 Config 3, E=2.5	21.09	20.59	4,218.33	2.15
DASH M2 Config 4, E=2.5	68.40	60.64	13,679.90	6.99
DASH M2 Config 5, E=2.5	71.88	62.60	14,377.00	7.34
DASH M2 Config 6, E=2.5	14.94	14.59	2,973.14	1.52
DASH M2 Config 7, E=2.5	16.08	15.71	3,215.47	1.64
DASH M2 Config 8, E=2.5	46.30	45.41	9,213.32	4.70
DASH M2 Config 9, E=2.5	46.36	45.52	9,225.35	4.71
DASH M2 Config 10, E=2.5	60.24	45.75	12,048.93	6.15
Smith-Waterman	16,259.14	14,069.69	3,251,827.13	1,660.56
NCBI-BLAST 2.2.6 (Default)	9.79	9.40	1,958.27	1.00
NCBI-BLAST 2.2.6 (No Filter)	21.37	10.16	4,274.56	2.18
NCBI-BLAST 2.2.6 (Report Everything)	49.65	11.27	9,929.06	5.07
BLAT*	2.10	2.07	420.75	0.21
PatternHunter**	78.37	78.61	15,673.57	8.00
FASTA	500.24	506.14	100,048.54	51.09
CAFE***	1,673.37	1,537.97	334,673.07	170.90

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 7.16: Comparison Of Nucleic Acid Search Speed (Using The Human UniGene (Nucleic Acid) Database), DASH M2, E=5.

Format	Search Time (seconds)			Search Time (×BLAST)
	mean	median	total	
DASH M2 Config 1 (unfolded), E=5	14.76	14.43	2,951.58	1.51
DASH M2 Config 2, E=5	21.26	20.94	4,251.74	2.17
DASH M2 Config 3, E=5	22.80	22.50	4,559.29	2.33
DASH M2 Config 4, E=5	70.30	61.89	14,060.70	7.18
DASH M2 Config 5, E=5	71.81	62.45	14,362.42	7.33
DASH M2 Config 6, E=5	15.38	15.10	3,060.80	1.56
DASH M2 Config 7, E=5	16.51	16.13	3,301.64	1.69
DASH M2 Config 8, E=5	59.11	45.66	11,821.70	6.04
DASH M2 Config 9, E=5	59.09	45.59	11,817.18	6.03
DASH M2 Config 10, E=5	60.27	45.83	12,054.41	6.16
Smith-Waterman	16,259.14	14,069.69	3,251,827.13	1,660.56
NCBI-BLAST 2.2.6 (Default)	9.79	9.40	1,958.27	1.00
NCBI-BLAST 2.2.6 (No Filter)	21.37	10.16	4,274.56	2.18
NCBI-BLAST 2.2.6 (Report Everything)	49.65	11.27	9,929.06	5.07
BLAT*	2.10	2.07	420.75	0.21
PatternHunter**	78.37	78.61	15,673.57	8.00
FASTA	500.24	506.14	100,048.54	51.09
CAFE***	1,673.37	1,537.97	334,673.07	170.90

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 7.17: Comparison Of Nucleic Acid Search Speed (Using The Human UniGene (Nucleic Acid) Database), DASH M2, E=10.

Format	Search Time (seconds)			Search Time (×BLAST)
	mean	median	total	
DASH M2 Config 1 (unfolded), E=10	15.07	14.50	3,014.77	1.54
DASH M2 Config 2, E=10	21.60	20.99	4,319.24	2.21
DASH M2 Config 3, E=10	23.02	22.35	4,603.76	2.35
DASH M2 Config 4, E=10	70.93	62.46	14,186.79	7.24
DASH M2 Config 5, E=10	71.78	62.42	14,355.01	7.33
DASH M2 Config 6, E=10	15.77	15.20	3,154.32	1.61
DASH M2 Config 7, E=10	16.87	16.02	3,374.47	1.72
DASH M2 Config 8, E=10	59.61	45.58	11,922.71	6.09
DASH M2 Config 9, E=10	59.51	45.55	11,902.85	6.08
DASH M2 Config 10, E=10	60.22	45.74	12,044.17	6.15
Smith-Waterman	16,259.14	14,069.69	3,251,827.13	1,660.56
NCBI-BLAST 2.2.6 (Default)	9.79	9.40	1,958.27	1.00
NCBI-BLAST 2.2.6 (No Filter)	21.37	10.16	4,274.56	2.18
NCBI-BLAST 2.2.6 (Report Everything)	49.65	11.27	9,929.06	5.07
BLAT*	2.10	2.07	420.75	0.21
PatternHunter**	78.37	78.61	15,673.57	8.00
FASTA	500.24	506.14	100,048.54	51.09
CAFE***	1,673.37	1,537.97	334,673.07	170.90

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 7.18: Comparison Of Nucleic Acid Search Speed (Using The Human UniGene (Nucleic Acid) Database), DASH M4, E=1.5.

Format	Search Time (seconds)			Search Time
	mean	median	total	(×BLAST)
DASH M4 Config 1 (unfolded), E=1.5	45.96	42.08	9,191.08	4.69
DASH M4 Config 2, E=1.5	58.36	53.10	11,672.56	5.96
DASH M4 Config 3, E=1.5	60.43	54.53	12,086.86	6.17
DASH M4 Config 4, E=1.5	157.42	140.26	31,484.02	16.08
DASH M4 Config 5, E=1.5	228.19	202.94	45,637.93	23.31
DASH M4 Config 6, E=1.5	49.96	45.11	9,991.64	5.10
DASH M4 Config 7, E=1.5	51.06	45.81	10,212.24	5.21
DASH M4 Config 8, E=1.5	89.35	81.39	17,869.50	9.13
DASH M4 Config 9, E=1.5	89.34	81.88	17,867.26	9.12
DASH M4 Config 10, E=1.5	131.25	114.04	26,250.75	13.41
Smith-Waterman	16,259.14	14,069.69	3,251,827.13	1,660.56
NCBI-BLAST 2.2.6 (Default)	9.79	9.40	1,958.27	1.00
NCBI-BLAST 2.2.6 (No Filter)	21.37	10.16	4,274.56	2.18
NCBI-BLAST 2.2.6 (Report Everything)	49.65	11.27	9,929.06	5.07
BLAT*	2.10	2.07	420.75	0.21
PatternHunter**	78.37	78.61	15,673.57	8.00
FASTA	500.24	506.14	100,048.54	51.09
CAFE***	1,673.37	1,537.97	334,673.07	170.90

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 7.19: Comparison Of Nucleic Acid Search Speed (Using The Human UniGene (Nucleic Acid) Database), DASH M4, E=2.5.

Format	Search Time (seconds)			Search Time
	mean	median	total	(×BLAST)
DASH M4 Config 1 (unfolded), E=2.5	54.86	51.30	10,971.98	5.60
DASH M4 Config 2, E=2.5	66.56	60.50	13,312.95	6.80
DASH M4 Config 3, E=2.5	68.75	62.30	13,750.53	7.02
DASH M4 Config 4, E=2.5	175.42	158.83	35,084.76	17.92
DASH M4 Config 5, E=2.5	228.30	202.31	45,659.76	23.32
DASH M4 Config 6, E=2.5	60.71	56.38	12,141.75	6.20
DASH M4 Config 7, E=2.5	61.92	57.38	12,384.72	6.32
DASH M4 Config 8, E=2.5	101.40	93.67	20,279.94	10.36
DASH M4 Config 9, E=2.5	101.38	93.63	20,275.09	10.35
DASH M4 Config 10, E=2.5	131.33	113.87	26,266.05	13.41
Smith-Waterman	16,259.14	14,069.69	3,251,827.13	1,660.56
NCBI-BLAST 2.2.6 (Default)	9.79	9.40	1,958.27	1.00
NCBI-BLAST 2.2.6 (No Filter)	21.37	10.16	4,274.56	2.18
NCBI-BLAST 2.2.6 (Report Everything)	49.65	11.27	9,929.06	5.07
BLAT*	2.10	2.07	420.75	0.21
PatternHunter**	78.37	78.61	15,673.57	8.00
FASTA	500.24	506.14	100,048.54	51.09
CAFE***	1,673.37	1,537.97	334,673.07	170.90

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 7.20: Comparison Of Nucleic Acid Search Speed (Using The Human UniGene (Nucleic Acid) Database), DASH M4, E=5.

Format	Search Time (seconds)			Search Time (×BLAST)
	mean	median	total	
DASH M4 Config 1 (unfolded), E=5	67.03	61.25	13,406.81	6.85
DASH M4 Config 2, E=5	82.62	75.76	16,523.68	8.44
DASH M4 Config 3, E=5	85.76	79.04	17,151.31	8.76
DASH M4 Config 4, E=5	207.93	187.86	41,585.04	21.24
DASH M4 Config 5, E=5	228.14	202.86	45,628.39	23.30
DASH M4 Config 6, E=5	75.06	68.69	15,012.03	7.67
DASH M4 Config 7, E=5	76.43	69.80	15,286.00	7.81
DASH M4 Config 8, E=5	118.83	107.69	23,766.83	12.14
DASH M4 Config 9, E=5	118.85	107.72	23,770.43	12.14
DASH M4 Config 10, E=5	131.34	113.58	26,268.15	13.41
Smith-Waterman	16,259.14	14,069.69	3,251,827.13	1,660.56
NCBI-BLAST 2.2.6 (Default)	9.79	9.40	1,958.27	1.00
NCBI-BLAST 2.2.6 (No Filter)	21.37	10.16	4,274.56	2.18
NCBI-BLAST 2.2.6 (Report Everything)	49.65	11.27	9,929.06	5.07
BLAT*	2.10	2.07	420.75	0.21
PatternHunter**	78.37	78.61	15,673.57	8.00
FASTA	500.24	506.14	100,048.54	51.09
CAFE***	1,673.37	1,537.97	334,673.07	170.90

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 7.21: Comparison Of Nucleic Acid Search Speed (Using The Human UniGene (Nucleic Acid) Database), DASH M4, E=10.

Format	Search Time (seconds)			Search Time (×BLAST)
	mean	median	total	
DASH M4 Config 1 (unfolded), E=10	73.28	65.48	14,656.82	7.48
DASH M4 Config 2, E=10	90.22	80.72	18,044.92	9.21
DASH M4 Config 3, E=10	92.92	83.09	18,584.88	9.49
DASH M4 Config 4, E=10	222.54	199.86	44,507.16	22.73
DASH M4 Config 5, E=10	228.04	202.27	45,608.33	23.29
DASH M4 Config 6, E=10	82.35	74.36	16,470.22	8.41
DASH M4 Config 7, E=10	83.73	74.41	16,746.83	8.55
DASH M4 Config 8, E=10	128.36	112.52	25,672.03	13.11
DASH M4 Config 9, E=10	128.38	112.41	25,676.75	13.11
DASH M4 Config 10, E=10	131.32	113.56	26,264.41	13.41
Smith-Waterman	16,259.14	14,069.69	3,251,827.13	1,660.56
NCBI-BLAST 2.2.6 (Default)	9.79	9.40	1,958.27	1.00
NCBI-BLAST 2.2.6 (No Filter)	21.37	10.16	4,274.56	2.18
NCBI-BLAST 2.2.6 (Report Everything)	49.65	11.27	9,929.06	5.07
BLAT*	2.10	2.07	420.75	0.21
PatternHunter**	78.37	78.61	15,673.57	8.00
FASTA	500.24	506.14	100,048.54	51.09
CAFE***	1,673.37	1,537.97	334,673.07	170.90

* Search times include time spent by server shared among all queries (BLAT).

** Minimum search time subtracted from all other queries to exclude cost of indexing for algorithms that index while searching (PatternHunter).

*** Search time is divided by number of index partitions (CAFE).

Table 7.22: Break Down Of DASH+NP3/NIX Search Time: Total seconds expected on various activities for the 200 standard queries for configurations (C) 1 – 4, and posting exclusion thresholds (E) 1.5, 2.5, 5.0 and 10 times random expectation. Configuration 1 is the control which does not employ cooperative compression.

		DASH M2					DASH M4				
		Activity					Activity				
C	E	NP3	NIX	HSP	DP	X	NP3	NIX	HSP	DP	X
1	1.5	4,664	225	181	264	82	7,852	1,329	2,173	5,570	501
2	1.5	5,639	575	468	237	82	8,576	1,354	3,687	4,804	443
3	1.5	6,004	624	497	238	82	9,452	1,476	3,978	6,185	518
4	1.5	18,674	1,227	2,376	296	91	22,189	2,164	15,939	7,078	585
1	2.5	4,928	241	197	314	88	8,922	1,627	1,943	8,139	640
2	2.5	5,885	578	510	276	85	9,385	1,441	3,829	6,620	547
3	2.5	6,340	619	529	277	86	9,613	1,473	3,948	6,514	541
4	2.5	18,732	1,234	2,471	323	94	22,259	2,191	16,168	7,469	605
1	5.0	4,992	247	206	373	95	9,510	1,950	2,313	11,402	796
2	5.0	5,945	579	526	326	89	9,872	1,560	4,495	8,947	664
3	5.0	6,382	628	552	322	92	10,082	1,579	4,613	8,772	656
4	5.0	18,710	1,240	2,552	381	99	22,785	2,375	18,964	10,289	753
1	10	5,013	248	210	458	104	9,638	2,054	2,515	13,106	894
2	10	5,954	581	537	393	99	9,958	1,592	4,740	10,210	736
3	10	6,394	628	559	389	99	10,169	1,607	4,837	10,003	724
4	10	18,692	1,244	2,581	462	107	22,968	2,428	19,778	19,778	841

NP3 counts time spent decompressing NP3 compressed data.

NIX counts time spent decompressing compressed postings lists from NIX files.

HSP counts time spent discovering and extending HSPs, including translating HSPs when this is done.

DP counts time spent performing dynamic programming.

X counts time spent doing all other activities, including output and house keeping.

Table 7.23: Human Genome Nucleic Acid Database And Index Sizes For Surveyed Algorithms In Megabytes (MB) And Bits Per Base (B/B).

Format	Bodies Only		Descriptions^		Index		Total	
	MB	B/B	MB	B/B	MB	B/B	MB	B/B
DASH+NP3/NIX (No folding)	654	1.78	1	0.01	3,442	9.36	4,098	11.15
DASH+NP3/NIX (With folding)	654	1.78	1	0.01	3,352	9.12	4,007	10.91
GeNML (Korodi and Tabus 2005)	563	1.54	-	-	-	-	-	-
Smith-Waterman* (FASTA ASCII)	3,004	8.21	289	0.79	-	-	3,293	9.00
BLAST (formatdb)	736	2.01	433	1.18	32	0.09	1,201	3.28
BLAT* (faToTwoBit)	950	2.60	-	-	1,867	5.10	2,817	7.70
PatternHunter** (FASTA ASCII)	3,004	8.21	289	0.79	-	-	3,293	9.00
FASTA** (FASTA ASCII)	3,004	8.21	289	0.79	-	-	3,293	9.00
CAFE*** (CAFE Index)	987	2.70	9	0.02	9,950	27.18	10,945	29.90

* Indicates that program maintains an index in RAM, and that the database format contains both sequence bodies and descriptions (BLAT).

** Indicates that algorithm indexes during searching (PatternHunter and FASTA), or does not use an index (Smith-Waterman).

*** Indicates that multiple small indices were used instead of one large index, due to technical difficulties (CAFE).

^ The great variability in the size of the compressed descriptions arises because the descriptions in this database are very compressible, but only CAFE and DASH make a serious attempt at compressing them.

Table 7.24: Nucleotide Decompression Speed (No Descriptions) Of GZIP, NP3, And The GeNML Implementation Of Chapter 7 For The De Facto DNA Corpus, And The DNA Databases Used In This Dissertation.

Database	Program	Seconds	Bases per Second
De Facto Corpus*	GZIP**	0.09	14,212,789
	NP3 (byte-aligned)	0.06	21,319,183
	NP3 (GeNML variant)	0.82	1,559,940
Human UniGene	GZIP**	13.23	148,100,786
	NP3 (byte-aligned)	11.93	164,239,178
	NP3 (GeNML variant)	91.95	21,309,137
Human Genome	GZIP**	25.52	120,790,046
	NP3 (byte-aligned)	22.21	138,791,624
	NP3 (GeNML variant)	215.68	13,294,078

* The small size of this corpus made it difficult to obtain accurate timing on the 1.8GHZ Opteron systems running Linux with its 10ms granularity. A 400 MHz Ultra-SPARC-II based system running Solaris 10 with micro-second granularity was used instead.

** To make the comparison between GZIP and the NP3 hosted algorithms, GZIP was given an input file consisting only of the DNA letters, i.e., striped of all descriptions and white space.

Table 7.25: For Each Of: The De Facto DNA Compression Corpus; The Human UniGene Database; And The Human Genome Database: Compressed Database And Index Sizes, With Cooperative Compression Disabled (a), and Enabled (b), And Postings Omitted Through Cooperative Compression.

Program	Compressed Size (bits per base) Bodies	Index	Both	Postings Omitted
De Facto Corpus				
GeNML*	1.73	-	-	-
NP3 (byte-aligned) (a)	1.93	20.44	22.37	-
NP3 (byte-aligned) (b)	1.93	19.82	21.75	4.31%
NP3 (GeNML variant) (a)	1.79	20.44	22.23	-
NP3 (GeNML variant) (b)	1.79	17.86	19.65	16.41%
Human UniGene				
GeNML*	-	-	-	-
NP3 (byte-aligned) (a)	0.71	7.58	8.29	-
NP3 (byte-aligned) (b)	0.71	4.98	5.69	19.90%
NP3 (GeNML variant) (a)	0.73	7.59	8.32	-
NP3 (GeNML variant) (b)	0.73	5.43	6.16	16.13%
Human Genome				
GeNML*	1.54	-	-	-
NP3 (byte-aligned) (a)	1.78	9.37	11.15	-
NP3 (byte-aligned) (b)	1.78	9.21	10.99	1.61%
NP3 (GeNML variant) (a)	1.67	8.29	9.96	-
NP3 (GeNML variant) (b)	1.67	7.11	8.78	7.24%

* GeNML results are those reported by Korodi and Tabus (2005). No results have been reported by them for the Human UniGene database.

Part IV

Summary Of Results And Conclusions

Chapter 8

Conclusions

In Part II it was demonstrated that, for a database where whole records are redundant, that using cooperative compression could reduce sequence search and alignment time requirements by 6% – 13%, and space requirements by 18%, while obtaining a small increase in sensitivity.

The results of Part II were generalised in Part III, where the cooperative compression techniques were refined and applied to a sorted database containing less redundancy (the Human UniGene nucleotide database), and finally to an unsorted database of "typical" composition and redundancy, namely the Human Genome Database. Substantial space savings were again made, of 40% (for the Human UniGene database) and 15% (for the Human Genome database), and were also accompanied by significantly increased sensitivity.

The partitioned data model allows DASH to processes batches of queries from an on-disk database several times faster than NCBI-BLAST could process them using a memory resident database. This is in spite of increased search times for single queries due to the high computational cost of decompressing database and index records.

It is this computational complexity that presents the greatest difficulty in making the methods of this dissertation generally successful. The need to decompress additional documents in order to reconstruct the index postings omitted by the cooperative compression of the postings list results in an unavoidable time cost.

The partitioned index structure means each database partition and its index can be fully decompressed into RAM, thus allowing much of the extra search time cost to be amortised by processing queries in batches. However, this is not a perfect solution: single queries are still slow and, critically, the time cost of translating HSPs from one context to another remains. Therefore the end result is a search process that while having the potential to be faster by avoiding the repeated searching of identical material, often ended up being slower due to higher constant overheads, as reflected in the fact that HSP translation turned out to be slower than discovering HSPs in the ordinary way.

Work is warranted in exploring how to make the translation of HSPs faster. This may be possible by creating a more efficient test as to whether any given translation can be skipped. The current test requires a cache unfriendly tree traversal, which results in the test being slower than either the translation step, or the discovery of the alignment in the ordinary way.

Nonetheless, it was shown that it is possible to explicitly re-use information about recurrent strings obtained during the database compression phase in the compression of the index, and also the searching of the database, which efficiency issues aside, was successful and did result in a small gain in sensitivity. In this way the methods of this dissertation have contributed a new approach to increasing the sensitivity of an index based search. This creates a significant new avenue of exploration in attempting to devise a new generation of genomic search algorithms that combine the seemingly conflicting goals of improved speed, sensitivity and space characteristics.

Overall, it was shown that the considerable space savings that cooperative compression delivers can result in reduced search times overall, especially if the data are stored on disk. Thus, the possibility of creating DNA compression, indexing, and search algorithms that reduce the space and time requirements, while simultaneously increasing the search sensitivity was again demonstrated in principle, if not in practice.

8.1 Conclusions

In conclusion then, I can say that evidence was found in support of the thesis of this dissertation that was proposed in Chapter 1. However, this was only achieved when sufficient redundancy existed within the database being considered, and that the redundancy was in the form of whole repeated records, when searches were batched together, or when the database was stored on disk.

When the redundancy consisted of smaller units, such as recurrent strings within records, then space savings and sensitivity increases can be made, but at the cost of a disproportionate increase in search time. However, this increase in search time could be reversed by batching queries, and would allow a DASH based system to process queries against a disk-resident database several time faster (in aggregate) than NCBI-BLAST, even if BLAST were able to operate from a memory-resident database.

The increased search time for individual queries was found to be primarily due to the fact that: (a) Decompressing sequence bodies, even using the fast NP3 byte-aligned codec, was slow if not amortised over multiple searches, and; (b) the fine grained approach of copying or translating alignments discovered for each instance of a recurrent string into each other instance of that string was more time consuming than simply discovering each alignment independently. That HSP translation turned out to be slower than re-discovery was a surprising result. Indeed, if this situation could be reversed, then the algorithms presented in this dissertation have the potential to be significantly faster than existing genomic sequence search and alignment algorithms. Indeed, the benefits may even flow on to other search and comparison systems, such as internet search engines.

In order to obtain further space savings, a DNA compression algorithm such as GeNML is required. However, this would further increase the decompression time, and therefore, the search time. Although the variation of the GeNML algorithm presented in this dissertation already reduces the amount of Arithmetic Coding performed, it is still too slow. The

greatest hope for making it faster would seem to lie in using the latest advances in fast Arithmetic Coding algorithms.

In summary, cooperative compression has been shown to obtain worthwhile time and space savings, accompanied by minor sensitivity gains, thus supporting the thesis of this dissertation. However, the current state of the art limits its applicability to biological sequence databases that contain sufficient redundancy. This is currently the exception rather than the rule, but as the quantity and redundancy of sequenced data continues to grow, cooperative compression will increase in relevance, particularly with the recent advent of algorithms that allow for the efficient sorting of large genomic databases.

Part V

Appendix

Appendix A

Invocation Commands For Search Algorithms

This appendix contains the invocation commands for: (a) each of the peer group of algorithms; and (b) the various invocations of the DASH search algorithm introduced in this thesis.

Table A.1: SSEARCH 3.4t25 (Smith-Waterman) Configuration.

	Command Template
Nucleic Acid Queries	`ssearch34 -n -a -b 500 -d 500 -E 1000000 -r +1/-3 -f -5 -g -2 -m 0 -Q $QUERY -O $OUTFILE ${database}`
Protein Queries	`ssearch34 -p -a -b 500 -d 500 -E 1000000 -f -11 -g -1 -m 0 -Q $QUERY -O $OUTFILE ${database}`

Table A.2: BLAT Configuration.

	Command Template
Database/Index Construction	`grep -v '^$|^#' Hs.seq.all > hs.fa ; faToTwoBit hs.fa hs.seq.all.2bit ; gfServer localhost 12345 -tileSize=8 -maxDnaHits=500 -repMatch=65536 hs.seq.all.2bit`
Nucleic Acid Queries	`gfClient -out=blast localhost 12345 / $QUERY $OUTFILE`
Protein Queries	`~/bin/i386/blat -out=blast ~/data/genpept.fsa $QUERY -prot -tileSize=3 -ooc=3.ooc $OUTFILE`

Table A.3: NCBI-BLAST 2.2.6 Default Configuration.

	Command Template
Database/Index Construction	`formatdb -p F -i Hs.seq.all ; formatdb -p T -i genpept.fsa`
Nucleic Acid Queries	`blastall -p blastn -G 5 -E 2 -d $BLAST_DB_FILE -i $QUERY -o $OUTFILE`
Protein Queries	`blastall -p blastp -d $BLASTP_DB_FILE -i $QUERY -o $OUTFILE`

Table A.4: NCBI-BLAST 2.2.6 No Filtering Configuration.

	Command Template
Database/Index Construction	`formatdb -p F -i Hs.seq.all ; formatdb -p T -i genpept.fsa`
Nucleic Acid Queries	`blastall -p blastn -G 5 -E 2 -F F -d $BLAST_DB_FILE -i $QUERY -o $OUTFILE`
Protein Queries	`blastpgp -s T -F F -d $BLASTP_DB_FILE -i $QUERY -o $OUTFILE`

Table A.5: NCBI-BLAST 2.2.6 Report Everything Configuration.

	Command Template
Database/Index Construction	`formatdb -p F -i Hs.seq.all ; formatdb -p T -i genpept.fsa`
Nucleic Acid Queries	`blastall -p blastn -G 5 -E 2 -e 500 -v 100000 -b 100000 -d $BLAST_DB_FILE -i $QUERY -o $OUTFILE`
Protein Queries	`blastall -p blastp -e 500 -v 100000 -b 100000 -d $BLASTP_DB_FILE -i $QUERY -o $OUTFILE`

Table A.6: PatternHunter Configuration.

	Command Template
Nucleic Acid Queries	`phn -m 32 -db 3 -i $query -j $database -o $outputfile`

Table A.7: FASTA Configuration.

	Command Template
Nucleic Acid Queries	`fasta34 -n -a -b 500 -d 500 -E 1000000 -f 5 -g 2 -m 0 -Q $QUERY -O $OUTFILE Hs.seq.all 6`
Protein Queries (a)	`fasta34 -p -a -b 500 -d 500 -E 1000000 -f 11 -g 1 -m 0 -Q $QUERY -O $OUTFILE genpept.fsa 1`
Protein Queries (b)	`fasta34 -p -a -b 500 -d 500 -E 1000000 -f 11 -g 1 -m 0 -Q $QUERY -O $OUTFILE genpept.fsa 2`

Table A.8: CAFE Configuration.

	Command Template
Database/Index Construction	`cafe -g 9 -f -v Hs.seq.all; cafe -a -g 3 -f -v genpept.fsa`
Nucleic Acid Queries	`cafe -n 500 -G 5 -E 2 -I $database < $query >$outputfile`
Protein Queries	`cafe -a -n 500 -G 11 -E 1 -I $database < $query >$outputfile`

Table A.9: DASH+FOLDDB M2 Configuration.

	Command Template
Database/Index Construction	`folddb -e 2.5 -r 50000000 -s 65000 -d Hs.seq.all -o index.Hs.seq.all; folddb -e 10 -r 50000000 -s 65000 -d genpept.fsa -o index.genpept.fsa; folddb -2 -e 10 -r 50000000 -s 65000 -d genpept.fsa -o index.genpept.fsa.nofold`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d index.Hs.seq.all -i $query -o $outputfile -G 5 -E 2`
Protein Queries (folded)	`dash -s mode2 -b 1000 -p dashp -d index.genpept.fsa -i $query -o $outputfile`
Protein Queries (normal)	`dash -s mode2 -b 1000 -p dashp -d index.genpept.fsa.nofold -i $query -o $outputfile`

Table A.10: DASH+FOLDDB M4 Configuration.

	Command Template
Database/Index Construction	`folddb -e 2.5 -r 50000000 -s 65000 -d Hs.seq.all -o index.Hs.seq.all; folddb -e 10 -r 50000000 -s 65000 -d genpept.fsa -o index.genpept.fsa ; folddb -2 -e 10 -r 50000000 -s 65000 -d genpept.fsa -o index.genpept.fsa.nofold`
Nucleic Acid Queries	`dash -s mode4 -b 1000 -p dashn -d index.Hs.seq.all -i $query -o $outputfile -G 5 -E 2`
Protein Queries (folded)	`dash -s mode4 -b 1000 -p dashp -d index.genpept.fsa -i $query -o $outputfile`
Protein Queries (normal)	`dash -s mode4 -b 1000 -p dashp -d index.genpept.fsa.nofold -i $query -o $outputfile`

Table A.11: DASH + NP3/NIX Configuration 1: No Cooperative Compression (Negative Control).

	Command Template
Database/Index Construction	`np3 -9 Hs.seq.all;` `nix -v -r -k -E 1.50 Hs.seq.all.np3`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;` `dash -s mode4 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;`

Table A.12: DASH + NP3/NIX Configuration 2: Forward Indexing.

	Command Template
Database/Index Construction	`np3 -9 Hs.seq.all;` `nix -v -E 1.50 Hs.seq.all.np3`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;` `dash -s mode4 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;`

Table A.13: DASH + NP3/NIX Configuration 3: Forward Indexing, Prefer Inter-Record References.

	Command Template
Database/Index Construction	`np3 -9 -n Hs.seq.all;` `nix -v -E 1.50 Hs.seq.all.np3`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;` `dash -s mode4 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;`

Table A.14: DASH + NP3/NIX Configuration 4: Forward Indexing, Prefer Inter-Record References, Rebate Estimated Savings Of Omitted Postings.

	Command Template
Database/Index Construction	`np3 -9 -n -R 13 Hs.seq.all;` `nix -v -E 1.50 Hs.seq.all.np3`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;` `dash -s mode4 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;`

Table A.15: DASH + NP3/NIX Configuration 5: Forward Indexing, Prefer Inter-Record References, Rebate Estimated Savings Of Omitted Postings, Do Not Exclude Stop k-mers.

	Command Template
Database/Index Construction	`np3 -9 -n -R 13 Hs.seq.all;` `nix -v Hs.seq.all.np3`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;` `dash -s mode4 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;`

Table A.16: DASH + NP3/NIX Configuration 6: Reverse Indexing.

	Command Template
Database/Index Construction	`np3 -9 Hs.seq.all;` `nix -v -r -E 1.50 Hs.seq.all.np3`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;` `dash -s mode4 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;`

Table A.17: DASH + NP3/NIX Configuration 7: Reverse Indexing, Prefer Inter-Record References.

	Command Template
Database/Index Construction	`np3 -9 -n Hs.seq.all;` `nix -v -r -E 1.50 Hs.seq.all.np3`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;` `dash -s mode4 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;`

Table A.18: DASH + NP3/NIX Configuration 8: Reverse Indexing, Prefer Inter-Record References, Rebate Estimated Savings Of Omitted Postings.

	Command Template
Database/Index Construction	`np3 -9 -n -R 13 Hs.seq.all;` `nix -v -r -E 1.50 Hs.seq.all.np3`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;` `dash -s mode4 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;`

Table A.19: DASH + NP3/NIX Configuration 9: Reverse Indexing, Prefer Inter-Record References, Rebate Estimated Savings Of Omitted Postings, Maximise Distinct Source Material.

	Command Template
Database/Index Construction	`np3 -9 -n -r -R 13 Hs.seq.all;` `nix -v -r -E 1.50 Hs.seq.all.np3`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;` `dash -s mode4 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;`

Table A.20: DASH + NP3/NIX Configuration 10: Reverse Indexing, Prefer Inter-Record References, Rebate Estimated Savings Of Omitted Postings, Maximise Distinct Source Material, Do Not Exclude Stop *k*-mers.

	Command Template
Database/Index Construction	`np3 -9 -n -r -R 13 Hs.seq.all;` `nix -v -r Hs.seq.all.np3`
Nucleic Acid Queries	`dash -s mode2 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;` `dash -s mode4 -b 1000 -p dashn -d Hs.seq.all.nix -i $query -o $outputfile -G 5 -E 2;`

Bibliography

Abbasi S. and Sengupta A. An O($n$log$n$) algorithm for finding dissimilar strings. *Inf. Process. Lett.*, 62(3):135–139, 1997.

Altschul S. Amino acid substitution matrices from an information theoretic perspective. *J.Mol. Biol.*, 219:555–565, 1991.

Altschul S. A protein alignment scoring system sensitive at all evolutionary distances. *J.Mol. Evol.*, 36:290–300, 1993.

Altschul S., Boguski M. S., Gish W., and Wootton J. C. Issues in searching molecular sequence databases. *Nature Genetics*, 6:119–129, 1994.

Altschul S. and Gish W. Local alignment statistics. *Methods Enzymol.*, 266:460–480, 1996a.

Altschul S., Gish W., Miller W., Myers E., and Lipman D. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

Altschul S. F. Evaluating the statistical significance of multiple distinct local alignments. In Suhai S., editor, *Theoretical and Computational Methods in Genome Research*, pages 1–14. Plenum Press, New York, 1997.

Altschul S. F. and Gish W. Local alignment statistics. *Methods Enzymol.*, 266:460–480, 1996b.

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

Anderson D. P., Cobb J., Korpela E., Lebofsky M., and Werthimer D. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

Anh V. N., de Kretser O., and Moffat A. Vector-space ranking with effective early termination. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 35–42, New York, NY, USA, 2001. ACM Press.

Anh V. N. and Moffat A. Index compression using fixed binary codewords. In Schewe K. D. and Williams H., editors, *Proc. 15th Australasian Database Conference*, Dunedin, New Zealand, January 2004.

Anh V. N. and Moffat A. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.

Arnold R. and Bell T. C. A corpus for the evaluation of lossless compression algorithms. In *Designs, Codes and Cryptography*, pages 201–210, 1997.

Balkenhol B., Kurtz S., and Shtarkov Y. M. Modifications of the burrows and wheeler data compression algorithm. In *Data Compression Conference*, pages 188–197, 1999.

Barker K., Youngblood R. F., Burdick D. W., Barker K. L., and Wessel W. W., editors. *New American Standard Bible*. The Lockman Foundation, La Habra, CA 90631, U.S.A., 1960 - 1995.

Barton G. *Protein Sequence Alignment and Database Scanning*, chapter 2, pages 31–64. IRL Press at Oxford University Press, 1996.

Batzoglou S., Jaffe D. B., Stanley K., Butler J., Gnerre S., Mauceli E., Berger B., Mesirov J. P., and Lander E. S. Arachne: A whole-genome shotgun assembler. *Genome Res.*, 12 (1):177–189, January 2002. URL `http://dx.doi.org/10.1101/gr.208902`.

Begleiter R., El-Yaniv R., and Yona G. On prediction using variable order markov models. *Journal of Artificial Intelligence Research (JAIR)*, 22:385–421, May 2004.

Behzadi B. and Le Fessant F. Dna compression challenge revisited: A dynamic programming approach. In *Combinatorial Pattern Matching*, volume 3537 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 2005.

Bell T. C. and Kulp D. Longest-match string searching for ziv-lempel compression. *Software - Practice and Experience*, 23(7):757–771, 1993.

Bell T. C., Moffat A., Nevill-Manning C. G., Witten I. H., and Zobel J. Data compression in full-text retrieval systems. *JASIS*, 44(9):508–531, 1993.

Benson D. A., Karsch-Mizrachi I., Lipman D. J., Ostell J., and Wheeler D. L. Genbank. *Nucleic Acids Research*, 34(1):D16–D20, 2006.

Bentley J. L. and Yao A. C.-C. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

Bernstein Y. and Cameron M. Fast discovery of similar sequences in large genomic collections. In Lalmas M., MacFarlane A., Rüger S. M., Tombros A., Tsikrika T., and Yavlinsky A., editors, *Advances in Information Retrieval, 28th European Conference on IR Research, ECIR 2006, London, UK, April 10-12, 2006, Proceedings*, volume 3936 of *Lecture Notes in Computer Science*, pages 432–443. Springer, 2006. ISBN 3-540-33347-9.

Bernstein Y. and Zobel J. Redundant documents and search effectiveness. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 736–743, New York, NY, USA, 2005. ACM Press.

Blanco R. and Barreiro A. Characterization of a simple case of the reassignment of document identifiers as a pattern sequencing problem. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 587–588, New York, NY, USA, 2005. ACM Press.

Blandford D. and Blelloch G. Index compression through document reordering. In Storer J. A. and Cohn M., editors, *Proc. 2002 IEEE Data Compression Conference*, pages 342–351, Los Alamitos, CA, U.S.A, April 2002. IEEE Computer Society Press.

Blandford D. K. and Blelloch G. E. Compact representations of ordered sets. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 11–19, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

Blandford D. K., Blelloch G. E., and Kash I. A. Compact representations of separable graphs. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 679–688, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

Bock A., Forchhammer K., Heider J., and Baron C. Selenoprotein synthesis: an expansion of the genetic code. *Trends Biochem Sci*, 16(12):463–467, Dec 1991.

Bookstein A., Klein S. T., and Raita T. Model based concordance compression. In *Proceedings of the Data Compression Conference DCC-92*, pages 82–91, Washington, D.C., 1992. IEEE Computer Society.

Bookstein A., Klein S. T., and Raita T. Is huffman coding dead? In *Research and Development in Information Retrieval*, pages 80–87, 1993.

Booth H. S., Maindonald J. H., Wilson S. R., and Gready J. E. An efficient z-score algorithm for assessing sequence alignments. *Journal of Computational Biology*, 11(4): 616–625, 2004.

Brudno M., Do C. B., Cooper G. M., Kim M. F., Davydov E., Green E. D., Sidow A., and Batzoglou S. Lagan and multi-lagan: Efficient tools for large-scale multiple alignment of genomic dna. *Genome Research*, pages 721–731, 2003.

Buckley C. and Lewit A. F. Optimization of inverted vector searches. In *SIGIR '85: Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 97–110, New York, NY, USA, 1985. ACM Press.

Bunton S. Semantically motivated improvements for PPM variants. *The Computer Journal*, 40(2/3):76–93, 1997.

Burrows M. and Wheeler D. J. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

Califano A. and Rigoutsos I. Flash: A fast look-up algorithm for string homology. In *In International Conference on Intelligent Systems for Molecular Biology, Bethesda, MD, USA*, pages 56–64, 1993.

Cannane A. and Williams H. E. A general-purpose compression scheme for large collections. *ACM Trans. Inf. Syst.*, 20(3):329–355, 2002. ISSN 1046-8188.

Chaisson M. J. and Pevzner P. A. Short read fragment assembly of bacterial genomes. *Genome Res.*, 18(2):324–330, February 2008. URL `http://dx.doi.org/10.1101/gr.7088808`.

Challoner B. R. *Holy Bible, Challoner Revision of Douay-Rheims Translation*. Christian Classics Ethereal Library, Grand Rapids, MI, U.S.A., christian classics ethereal library edition, 1752. URL `http://www.ccel.org/ccel/bible/douayr.html`.

Chen X., Kwong S., and Li M. A compression algorithm for dna sequences and its applications in genome comparison. *Genome Informatics (GIW'99), Tokyo, Japan*, pages 51–61, 1999.

Chen X., Kwong S., and Li M. A compression algorithm for dna sequences. *IEEE Engineering in Medicine and Biology Magazine*, 20:61,66, 2001.

Chen X., Li M., Ma B., and Tromp J. Dnacompress: fast and effective dna sequence compression. *Bioinformatics*, 18(12):1696–1698, 2002a.

Chen X., Li M., Ma B., and Tromp J. Dnacompress: fast and effective dna sequence compression. *Bioinformatics Application Note, Oxford University Press*, 18(12):1696–1698, 2002b.

Chen X. L. A framework for comparing genomic search techniques. Master's thesis, Royal Melbourne Institute of Technology, July 2004.

Cheng C.-S., Shann J. J.-J., and Chung C.-P. A unique-order interpolative code for fast querying and space-efficient indexing in information retrieval systems. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*, page 229, Washington, DC, USA, 2004. IEEE Computer Society.

Cheng L., Cheung D., and Yiu S. Approximate string matching in dna sequences. In *Proceeings of the Eighth International Conference on Database Systems for Advanced Applications*, pages 303–310, 2003.

Choueka Y., Fraenkel A. S., and Klein S. T. Compression of concordances in full-text retrieval systems. In *SIGIR '88: Proceedings of the 11th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 597–612, New York, NY, USA, 1988. ACM Press.

Claverie J.-M. and States D. Information enhancement methods in large scale sequence analysis. *Computers in Chemistry*, 17:191–201, 1993.

Cleary J. G. and Teahan W. J. Unbounded length contexts for PPM. *The Computer Journal*, 40(2/3):67–75, July 1997.

Cleary J. G. and Witten I. H. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4):396–402, April 1984.

Collins J. F., Coulson A. F. W., and Lyall A. The significance of protein sequence similarities. *Computer Applications in the Biosciences*, 4(1):67–71, 1988.

Cormack G. V. and Horspool R. N. S. Data compression using dynamic Markov modelling. *The Computer Journal*, 30(6):541–550, 1987.

Darby J. N. *Holy Bible, Darby Translation*. Christian Classics Ethereal Library, Grand Rapids, MI, U.S.A., christian classics ethereal library edition, 1890. URL `http://www.ccel.org/ccel/bible/darby.html`.

Dayhoff M. O., Schwartz R. M., and Orcutt B. C. Matrices for detecting distant relationships. *Altas of Protein Sequence and Structure, Natl. Biomed. Res. Found., Washington, D.C.*, 5(3):345–352, 1978.

Dembo A., Karlin S., and Zeitouni O. Limit distribution of maximal non-aligned two-sequence segmental score. *Ann. Prob.*, 22:2022–2039, 1994.

Deorowicz S. Improvements to Burrows-Wheeler compression algorithm. *Software – Practice and Experience*, 30(13):1465–1483, 2000.

Deorowicz S. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software – Practice and Experience*, 32(2):99–111, 2002.

Dwan C. Speedup at what cost? In *O'Reilly's Bioinformatics Technology Conference, Tucson , Arizona*, Jan. 28-31 2002.

Effros M. Universal lossless source coding with the burrows wheeler transform. In *Data Compression Conference*, pages 178–187, 1999.

Elias P. Universal codeword sets and representations of the integers. *IEEE Transactions in Information Theory*, 21(2):194–203, March 1975.

Erdos P. and Renyi A. On a new law of large numbers. *J. Analyse Math*, 22:103–111, 1970.

Erdos P. and Revesz P. On the length of the longest head-run. *Topics in Information Theory, Math. Soc. J. Bolyai*, 16:219–228, 1975.

Fenwick P., Titchener M., and Lorenz M. Burrows wheeler - alternatives to move to front. In *DCC '03: Proceedings of the Conference on Data Compression*, page 428, Washington, DC, USA, 2003. IEEE Computer Society.

Fenwick P. M. A new data structure for cumulative probability tables: an improved frequency-to-symbol. *Softw. Pract. Exper.*, 26(4):489–490, 1996.

Fox E. A., Harman D. K., Baeza-Yates R., and Lee W. C. *Inverted Files*, chapter 3, pages 28–43. Frakes and Baeza-Yates, 1992.

Gailly J. L. Gzip program and documentation., 1993. URL `ftp://prep.ai.mit.edu/pub/gnu/gzip-*.tar`.

Galisson F. The fasta and blast programs. Technical Report bioweb.pasteur.fr/seqanal/blast/blast_fasta-uk.ps, Pasteur Institute, 2000.

Gallager R. and Voorhis D. V. Optimal source codes for geometrically distributed integer alphabets. *IEEE Transactions on Information Theory*, IT-21:228–230, 1975.

Gallager R. G. Variations on a theme by huffman. *IEEE Transactions on Information Theory*, IT-24(6):668–674, November 1978.

Gardner-Stephen P. and Knowles G. DASH: A new high speed genomic sequence search and alignment tool. In *Proceedings of WSEAS MCBC-MCBE-ICAI-ICAMSL, Puerto De La Cruz, Tenerife, Spain*, pages 59–64, Dec. 19-21 2003.

Gardner-Stephen P. and Knowles G. DASH: Localising dynamic programming for order of magnitude faster, accurate sequence alignment. In *Proceedings of the 3rd IEEE Conference on Computational Systems Bioinformatics (CSB2004), Stanford, USA*, pages 732–735, August 2004.

Gardner-Stephen P. and Knowles G. Np3: Cooperative compression of clustered dna databases and their indexes. *WSEAS Transactions on Biology and Biomedicine*, 3(7): 565–, July 2006.

Gilbert E. N. and Moore E. F. Variable length binary encodings. *Bell Systems Technical Journal*, pages 933–967, July 1959.

Golomb S. W. Run-length encodings. *IEEE Transactions on Information Theory*, IT-12 (3):399–401, 1966.

Grumbach S. and Tahi F. A new challenge for compression algorithms: genetic sequences. *J. Inform. Process. Management*, 30:866–875, 1994.

Hancock J. M. and Armstrong J. S. Simple34: an improved and enhanced implementaiton for vax and sun computers of the simple algorithm for analysis of clustered repetitive motifs in nucleotide sequences. *Comput. Appl. Biosci.*, 10:67–70, 1994.

Hardy P. and Waterman M. The sequence alignment software library at USC. 1997. URL `http://www-hto.usc.edu/software/seqaln/`.

Harman D. K. and Candela G. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41 (8):581–589, August 1990.

Heinz S. and Zobel J. Efficient single-pass index construction for text databases. *J. Am. Soc. Inf. Sci. Technol.*, 54(8):713–729, 2003. ISSN 1532-2882.

Henikoff S. and Henikoff J. G. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89:10915–10919, 1992.

Howard P. G. and Vitter J. S. Analysis of arithmetic coding for data compression. *Information Processing and Management*, 28(6):749–764, 1992.

Howard P. G. and Vitter J. S. Arithmetic coding for data compression. *Proc. IEEE*, 82(6): 857–865, 1994.

Huffman D. A. A method for the construction of minimum redundancy codes. *In Proceedings of the IRE*, 40:1098–1101, 1952.

Hunt E., Atkinson M., and Irving R. Database indexing for large DNA and protein sequence collections. *The VLDB Journal*, 11:256–271, 2002.

International Bible Society . *The Holy Bible, New International Version*. Zondervan Publishing House, Grand Rapids, Michigan 49530, U.S.A., 1973 - 1984.

Jaffe D. B., Butler J., Gnerre S., Mauceli E., Lindblad-Toh K., Mesirov J. P., Zody M. C., and Lander E. S. Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome Res*, 13(1):91–96, January 2003. ISSN 1088-9051. URL `http://dx.doi.org/10.1101/gr.828403`.

Johnson M. P. *Holy Bible, World English Version*. Rainbow Missions, Inc, Mesa CO 81643-0275, U.S.A., 2003. URL `http://www.ebible.org`.

Joint Commission on Biochemical Nomenclature . Abbreviations and symbols for the description of conformations of polynucleotide chains. recommendations 1982. *Eur. J. Biochem.*, 131:9–15, 1983.

Karlin S. and Altschul S. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc. Natl. Acad. Sci. USA*, 87: 2264–2268, 1990.

Karlin S. and Altschul S. F. Applications and statistics for multiple high-scoring segments in molecular sequences. *Proc. Natl. Acad. Sci. USA*, 90:5873–5877, 1993.

Karlin S., Ghandour G., Ost F., Tavare S., and Korn L. J. New approaches for computer analysis of nucleic acid sequences. *PNAS (USA)*, 80:5660–5664, 1983.

Kent W. Blat–the blast-like alignment tool. *Genome Res.*, 12(4):656–664, April 2002.

Knowles G. and Gardner-Stephen P. DASH, DASH-H: A software and hardware for sequence alignment. *WSEAS Transactions on Biology and Biomedicine*, 3(1):37–42, Jan. 2006.

Knuth D. E., Morris J. H., and Pratt V. R. Fast pattern matching in strings. *SIAM Journal on Computing*, pages 323–350, June 1977.

Korodi G. and Tabus I. An efficient normalized maximum likelihood algorithm for dna sequence compression. *ACM Trans. Inf. Syst.*, 23(1):3–34, 2005.

Kurtz S., Phillippy A., Delcher A., Smoot M., Shumway M., Antonescu C., and Salzberg S. Versatile and open software for comparing large genomes. *Genome Biology*, 5:R12, 2004.

Larsson N. J. *Structures of String Matching and Data Compression*. PhD thesis, Department of Computer Science, Lund University, Sweden, Sept. 1999.

Lester N., Moffat A., and Zobel J. Fast on-line index construction by geometric partitioning. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 776–783, New York, NY, USA, 2005. ACM Press.

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

Li M., Badger J. H., Chen X., Kwong S., Kearney P., and Zhang H. An information based sequences distance and its application to whole motochondrial genome phylogeny. *Bioinformatics*, 17(2):149–154, 2001.

Li M., Ma B., Kisman D., and Tromp J. PatternHunter II: Highly Sensitive and Fast Homology Search. *Journal of Bioinformatics and Computational Biology*, 2(3):417–439, 2004.

Lucarella D. A document retrieval system based upon nearest neighbour searching. *Journal of Information Science*, 14:25–33, 1988.

Manzini G. Invited lecture: The burrows-wheeler transform: Theory and practice. In Kutylowski M., Pacholski L., and Wierzbicki T., editors, *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Proceedings*, volume 1672 of *Lecture Notes in Computer Science*, pages 34–47. Springer, September 1999.

Manzini G. and Rastero M. A simple and fast dna compressor. *Softw. Pract. Exper.*, 34 (14):1397–1411, 2004.

Matsumoto T., Sadakane K., and Imai H. Biological sequence compression algorithms. *Genome Informatics Workshop, Universal Academy Press*, pages 43,52, 2000.

McDonell K. J. An inverted index implementation. *The Computer Journal*, 20(1):116,123, 1977.

Moffat A. An improved data structure for cumulative probability tables. *Softw. Pract. Exper.*, 29(7):647–659, 1999.

Moffat A., Neal R. M., and Witten I. H. Arithmetic coding revisited. *ACM Trans. Inf. Syst.*, 16(3):256–294, 1998.

Moffat A. and Stuiver L. Exploiting clustering in inverted file compression. In *DCC '96: Proceedings of the Conference on Data Compression*, page 82, Washington, DC, USA, 1996. IEEE Computer Society.

Moffat A. and Stuiver L. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

Moffat A. and Turpin A. Efficient construction of minimum-redundancy codes for large alphabets. *IEEE Transactions on Information Theory*, 44(4):1650–1657, 1998.

Moffat A. and Zobel J. Parameterised compression for sparse bitmaps. In *SIGIR '92: Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 274–285, New York, NY, USA, 1992. ACM Press.

Moffat A. and Zobel J. Compression and fast indexing for multi-gigabyte text databases. *Australian Computer Journal*, 26(1):1–9, 1994.

Moffat A. and Zobel J. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, 1996.

Moore G. E. Cramming more components onto integrated circuits. *Electronics*, 38(8): 114–117, April 19 1965.

Moore G. E. Excerpts of a conversation with gorden moore: Moore's law, 2005. URL `ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf`.

Mott R. Maximum-likelihood estimation of the statistical distribution of smith-waterman local sequence similarity scores. *Bull. Math. Biol.*, 54:59–75, 1992.

Nevill-Manning C. and Witten I. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997. URL `citeseer.ist.psu.edu/nevill-manning97identifying.html`.

Oberhumer M. Lzo - a real-time data compression library. Technical report, 1997. URL `http://www.oberhumer.com/opensource/lzo`.

Pearson W. R. Rapid and sensitive sequence comparison with fastp and fasta. *Methods in Enzymology*, 183:63–98, 1990.

Pearson W. R. Empirical statistical estimates for sequence similarity scores. *J. Mol. Biol.*, 276:71–84, 1998.

Peterson E. H. *Holy Bible Paraphrase, The Message*. NavPress Publishing Group NZ, Christchurch, New Zealand, 2002.

Pontius J. U., Wagner L., and Schuler G. D. *UniGene: a unified view of the transcriptome*, chapter 21. National Center for Biotechnology Information, 2003.

R Development Core Team . *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

Rissanen J. and Langdon G. G. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, 1979.

Rissanen J. and Langdon Jr. G. G. Universal modeling and coding. *IEEE Transactions on Information Theory*, 27(1):12–22, 1981.

Sadakane and Imai . Improving the speed of LZ77 compression by hashing and suffix sorting. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, 2000.

Saracevic T. Evaluation of evaluation in information retrieval. In Fox E. A., Ingwersen P., and Fidel R., editors, *SIGIR'95, Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Seattle, Washington, USA, July 9-13, 1995 (Special Issue of the SIGIR Forum)*, pages 138–146. ACM Press, 1995. ISBN 0-89791-714-6.

Schindler M. Szip block sorting file compression program, 1996. URL `http://www.compressconsult.com/szip/`.

Schuegraf E. J. Compression of large inverted files with hyperbolic term distribution. *Inf. Process. Manage.*, 12(6):377–384, 1976.

Schuler G. D. Pieces of the puzzle: expressed sequence tags and the catalog of human genes. *J Mol Med*, 75:694–698, 1997.

Schwartz R. M. and Dayhoff M. O. Matrices for detecting distant relationships. *Altas of Protein Sequence and Structure, Natl. Biomed. Res. Found., Washington, D.C.*, 5(3): 353–358, 1978.

Seward J., Burrows M., Wheeler D., Fenwick P., Moffat A., Neal R., and Witten I. The bzip2 compression program. URL `http://sources.redhat.com/bzip2`. 2001.

Shannon C. E. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, July, Oct 1948.

Shieh W.-Y., Chen T.-F., Shann J. J.-J., and Chung C.-P. Inverted file compression through document identifier reassignment. *Inf. Process. Manage.*, 39(1):117–131, 2003.

Silvestri F., Orlando S., and Perego R. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 305–312, New York, NY, USA, 2004a. ACM Press.

Silvestri F., Perego R., and Orlando S. Assigning document identifiers to enhance compressibility of web search engines indexes. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 600–605, New York, NY, USA, 2004b. ACM Press.

Smith T. and Waterman M. Identification of common molecular subsequences. *The Journal of Molecular Biology*, 147(1):195–197, March 1981.

Smith T. F., Waterman M. S., and Burks C. The statistical distribution of nucleic acid similarities. *Nucleic Acids Research*, 13:645–656, 1985.

Srinivasan G., James C. M., and Krzycki J. A. Pyrrolysine encoded by uag in archaea: charging of a uag-decoding specialized trna. *Science*, 296:1459–1462, 2002.

Teahan W. J. and Harper D. J. Combining PPM models using a text mining approach. In *Data Compression Conference*, pages 153–162, 2001.

Teuhola J. A compression method for clustered bit-vectors. *Information Processing Letters*, 7:308–311, October 1978.

Tjalkens T. and Willems F. M. J. Implementing the context-tree weighting method: Arithmetic coding,. In *Int. Conf. on Combinatorics, Information Theory and Statistics*, page 83, Portland, Maine, U.S.A, 18-20 1997.

Trotman A. Compressing inverted files. *Information Retrieval*, 6(1):5–19, 2003.

Turpin A. and Moffat. A. Housekeeping for prefix coding. *IEEE Trans. on Communications*, 48(4):622–628, April 2000.

Waterman M. S. and Vingron M. Sequence comparison significance and poisson approximation. *Stat. Sci.*, 9(3):367–381, 1994.

Weaver W. and Shannon C. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, Illinois, 1949. republished in paperback 1963.

Webster N. *Holy Bible, Webster's*. Christian Classics Ethereal Library, Grand Rapids, MI, U.S.A., christian classics ethereal library edition, 1833. URL `http://www.ccel.org/ccel/bible/darby.html`.

Welch T. A. A technique for high-performance data compression. *IEEE Computer*, 17(6): 8–20, 1984.

Werthimer D., Cobb J., Lebofsky M., Anderson D., and Korpela E. Seti@home: massively distributed computing for seti. *Comput. Sci. Eng.*, 3(1):78–83, 2001.

Wilcoxon F. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6): 80–83, 1945.

Willems F. M. J., Shtarkov Y. M., and Tjalkens T. J. The context-tree weighting method: basic properties. *IEEE Trans. Info. Theory*, pages 653–664, 1995.

Williams H. and Zobel J. Compression of nucleotide databases for fast searching, 1997a.

Williams H. E. Effective query filtering for fast homology searching. In Altman R. B., Dunker A. K., Hunter L., Klein T. E., and Lauderdale K., editors, *Pacific Symposium on Biocomputing*, pages 214–225, 1999.

Williams H. E. and Zobel J. Indexing nucleotide databases for fast query evaluation. In *EDBT '96: Proceedings of the 5th International Conference on Extending Database Technology*, pages 275–288, London, UK, 1996. Springer-Verlag.

Williams H. E. and Zobel J. Compression of nucleotide databases for fast searching. *Bioinformatics*, 13:549–554, 1997b.

Williams H. E. and Zobel J. Indexing and retrieval for genomic databases. *IEEE TKDE*, 14(1):63–78, 2002a.

Williams H. E. and Zobel J. Indexing and retrieval for genomic databases. *Knowledge and Data Engineering*, 14(1):63–78, 2002b.

Williams R. N. An extremely fast ziv-lempel data compression algorithm. In *Data Compression Conference*, pages 362–371, 1991.

Wirth A. I. Symbol-driven compression of burrows wheeler transformed text. Master's thesis, The University of Melbourne, 2001.

Witten I., Bell T., and Nevill C. Indexing and compressing full-text databases for cd-rom. *Journal of Information Science*, 17:265–271, 1992.

Witten I. H. and Bell. T. C. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37:1085–1094, 1991.

Witten I. H., Moffat A., and Bell T. C. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

Witten I. H., Neal R. M., and Cleary J. G. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.

Wootton J. and Federhen S. Statistics of local complexity in amino acid sequences and sequence databases. *Computers in Chemistry*, 17:149–163, 1993.

Wootton J. and Federhen S. Analysis of compositionally biased regions in sequence databases. *Methods in Enzymology*, 266:554–571, 1996.

Young R. *Holy Bible, Young's Literal Translation*. Christian Classics Ethereal Library, Grand Rapids, MI, U.S.A., christian classics ethereal library edition, 1898. URL `http://www.ccel.org/ccel/bible/ylt.html`.

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

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

Zobel J. and Moffat A. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006. ISSN 0360-0300.

Zobel J., Moffat A., and Sacks-Davis R. Searching large lexicons for partially specified terms using compressed inverted files. In Agrawal R., Baker S., and Bell D., editors, *Proceedings of the19th Conference on Very Large Databases*, pages 290–301, Dublin, Ireland, 1993.