BIOLOGICAL SEQUENCE INDEXING USING PERSISTENT JAVA

PhD Thesis

Elżbieta Katarzyna Pustułka-Hunt

December 2001

University of Glasgow
Department of Computing Science

Abstract

This thesis makes three contributions in the area of computing science.

Our first contribution is the recognition that new data types produced by large-scale biological research techiques lead to a flood of data which creates new challenges in the areas of data indexing, integration, manipulation and visualisation.

The second contribution is a new research methodology which combines orthogonal persistence with an empirical evaluation of disk-resident suffix indexes. This methodology allowed us to develop a practical algorithm for the construction of suffix trees on disk up to any size supported by the available file and addressing space, which has hitherto not been possible.

The third contribution is a new experimental methodology for examining the usefulness of suffix indexes, and the use of this methodology in an empirical investigation of the indexing gain achieved by combining an approximate matching algorithm with a large suffix index.

Those results are presented against the background of the changing technological landscape affecting life sciences and bioinformatics research and the resulting need for new computing solutions.

Thesis statement

We postulate that it is feasible and efficient to use persistent indexing in large repositories of biological sequence data. We will test our hypothesis by performing experiments on persistent suffix trees and measuring the advantages of using such indexes over the use of sequence comparison tools based on sequential data scanning.

Acknowledgements

I would like to thank my family for their support, and in particular my parents who volunteered to spend a long time in Glasgow in order to help me concentrate on research.

I would like to thank my supervisors, Malcolm Atkinson and Rob Irving, and all my colleagues, both in Computing Science Department and in the area of life sciences for helping me to engage in this exciting research.

Publications and important presentations forming part of this research

- unpublished paper, Physical Map Integration Using a Relational Database: the Example of the Human Chromosome 21 DB, Ela Pustułka-Hunt, Hans Lehrach and Marie-Laure Yaspo, produced in November 1999 and updated in April 2000, reproduced as Appendix A.
- technical report, E. Pustułka-Hunt and D. Jack, *Case study: Use of computer tools in locating a human disease gene*, University of Glasgow, Department of Computing Science, TR-1999-28, http://www.dcs.gla.ac.uk/∼ela/case.ps, 1999.
- technical report, E. Pustułka-Hunt, D, Jack, G. F. Hogg, and D. G. Monckton, *Case study: CGT repeat expansion modeling using a Java applet and its PJama extension providing persistent storage for genetics data*, University of Glasgow, Department of Computing Science, TR-1999-31, http://www.dcs.gla.ac.uk/∼ela/CTGcase.ps, 1999.
- refereed paper, E. Hunt, *PJama Stores and Suffix Tree Indexing for Bioinformatics Applications*, 10th PhD Workshop at ECOOP'00, http://www.inf.elte.hu/∼phdws/-timetable.html, 2000.
- refereed paper, E. Hunt, M. P. Atkinson and R. W. Irving, *A database index to large biological sequences*, Proc. 27th Conf. on Very Large Databases, pages 139–148, Morgan Kaufmann, 2001.
- refereed presentation, E. Hunt, M. P. Atkinson and R. Irving, *Indexing the whole genome*, oral presentation at Workshop 9 (Genome Informatics) at Human Genome Meeting 2001 (HGM2001), April 19-22, 2001, Edinburgh.
- invited paper submitted to VLDB Journal in November 2001, E. Hunt, M. P. Atkinson and R. Irving, *Database Indexing for Large DNA and Protein Sequence Collections*, reproduced as Appendix C.

Contents

Chapter 1

Introduction

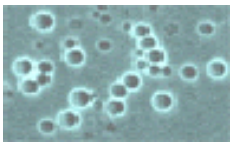


Figure 1.1: *Yeast YCC588, by Carsten Kettner, http://genome-www.stanford.edu*

The life sciences are a very exciting and important area for research. With the aid of computing technologies, they have the potential to bring about an understanding of life, disease and drug interventions which are so far known only at certain levels of abstraction but not in their entirety. For instance, the chemical activity of aspirin is well understood, but a full understanding of its molecular activity is still beyond our reach. Like aspirin, other drugs are prescribed, but the complex effects of those interventions on all living organisms and their environment are not appreciated. Such knowledge will become available when further research into the molecular mechanisms of disease and treatment explains the complex metabolic pathways involved. Such research will have human, economic, and safety benefits. It will allow the doctors of the future to use a drug only if it is not likely to cause an adverse reaction, or will let the farmer use safer pest and disease control agents. But to understand the complexity of life, new approaches to the management of biological knowledge are required, and our research explores some of the challenges.

The amount of data produced by biological laboratories is constantly increasing, and this data cannot be efficiently organised and accessed just by using simple office packages such as spreadsheets or word processing tools. Current data-storage techniques in biological laboratories rely on filing systems and do not use database management tools. In this situation, consistent data annotation does not exist on disk and access to data is slow. Without easy access to all types of data produced in distributed laboratories, efficient genomic research is not possible. Our research centres on database support for sequence searching, an activity which underpins many new technologies which attempt to analyse molecular interactions. Currently, every new protein or DNA sequence that is isolated in the lab is compared with known sequences. This comparison serves two purposes: first, establishing if this is a novel sequence, and second, finding similar sequences, so that a rough idea of the possible function and structure of this sequence can be gained. As the amount of known sequence is growing rapidly (November 2001 - 18 GB in Genbank[1], compared to 14 GB at the end of 2000), and several organisms are being sequenced currently, better and faster tools for sequence analysis are needed. Our research makes a contribution in this area.

[1] http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=Nucleotide

1.1 Terminology used to describe biological sequences

We briefly introduce the relevant biological terminology and the abbreviations used throughout this thesis. An excellent introduction to those issues, aimed at mathematicians and computing scientists, is available in Waterman [230]. An interested reader may want to consult one of the web information resources, for instance at the Virtual Library of Genetics[2] or the online Dictionary of Cell and Molecular Biology[3].

Deoxyribonucleic acid (DNA) is the genetic material of all cells and many viruses. It can be abstractly represented as strings over the alphabet of **A**, **C**, **G** and **T**. Ribonucleic acid (RNA) found in all living cells, and serving as the genome of some viruses, also has a four-letter code, with **U** replacing **T**. Human nuclear DNA is divided into 23 pairs of chromosomes of different length. The total length of human DNA is around 3 Gbp (3 billion base pairs) as read in one direction from one of the chromosomes in each pair, and complementary strands (A complements T, and C complements G) can be computed from this base sequence. DNA can be cut into smaller fragments using enzymes, and those fragments can be inserted into self-replicating DNAs of other microorganisms to give rise to clones. Clones reproduce by division and dutifully carry and pass on the human sequence along with their own DNA. Such clones can accept from a few Kbp (thousands of base pairs) for small microorganisms, to 2 Mbp (millions of base pairs) for yeast artificial chromosomes (YACs). In the cloning procedure, complementary DNA (cDNA)[4] can also be used. A cDNA clone can then be employed as a probe to locate the gene in the genomic DNA. Once DNA fragments are cloned into small unicellular organisms, reconstruction of the original sequence requires cutting the host including the insert with enzymes, sequencing of all fragments, and reassembly. Assembly of sequence for an individual clone is done by measuring the lengths of fragments produced by digesting those clones with different enzymes, and by comparing the sequence fragments. Once a full clone sequence is available, STS sequences (sequence tagged sites) help in further assembly. An STS should be a unique short sequence in a given genome, and is often isolated from a clone end. Using the polymerase chain reaction (PCR), the presence of an STS within a cloned fragment can be confirmed, and if the clone sequence is known, as is the case for most of the human genome now, electronic PCR can be used [195].

Cells use DNA as a blueprint for the production of enzymes and other proteins. In this process, there are intermediate messenger RNAs (mRNAs), and not all DNA is translated to protein. Genes are the parts of DNA that give rise to proteins and enzymes. In eukaryotes the part of DNA that contributes to the final product is called an exon, while the DNA that is dropped out during the gene processing is called an intron. A gene consists of exons, introns, and some regulatory regions, and the resultant protein will be a translation of connected exons where each triplet of DNA letters (bases) gives rise to one amino acid (AA) of the end product (AA alphabet has 20 letters). It is believed that humans have around 40,000 genes [226, 57], which amounts to around 1% of the human genome. During the process of protein synthesis this may lead to as many as 500,000 human proteins, as different exons within each gene can be selected for a particular gene in different cellular contexts, and later, protein structure modification will also contribute to protein diversity.

[2] http://www.ornl.gov/TechResources/Human_Genome/genetics.html

[3] http://on.to/dictionary

[4] Strong, cloned copies of otherwise fragile mRNA - the essential messenger element of the genes in the DNA which help in the coding of proteins, http://www.pbs.org/faithandreason/biogloss/cdna-body.html

1.2 Motivation

The initial motivation for this research arose from our work in the Chromosome 21 Transcriptional Mapping Project at the Max-Planck-Institute for Molecular Genetics in Berlin[5]. Our main contribution to that project was the creation of a database integrating known data sources of human chromosome 21 information, and the building of an integrated map of that chromosome [236, 122, 121] which was later used in sequencing, leading to the publication of the chromosome sequence last year [103]. An unpublished report concerning this work can be found in Appendix A. Part of the map integration exercise was to find possible sequence matches for all of the complementary DNA sequences (cDNAs) which were isolated during the mapping project. Such matches might help in drawing inter-species maps, and in annotating possible gene functions. If such matches were found in external databases, they could be added as map annotations, and provided as links. Matches to known clones or genes could also help us to position the cDNAs on the chromosome before sequencing, and help in the sequencing task itself, by providing anchors of known sequence [121]. The Chromosome 21 database was developed using object-relational technology, and sequence matching could not be performed in this context. An external service was used, and this task was performed by submitting 2,000 queries, each containing a cDNA sequence, to the BLAST server at NCBI[6]. Query sending was automated using a script, but the analysis of data returned from BLAST was performed manually by an expert biologist who knew how to distinguish between significant and insignificant matches. The chore of sifting through thousands of returned BLAST results was quite a considerable overhead and this task was performed several times during the project, so that new sequencing results could be included.

The second motivating experience came from the collaboration with Professor Tim Mitchell[7] of the Division of Infection and Immunity at Glasgow University. The problem set there was to investigate small protein motifs which might be responsible for the virulence of *Streptoccocus pneumoniae*. The underlying assumption in this analysis was that some repeated motifs in the protein sequence of this bacterium might give it special power to attack humans. If such motifs were found, they might provide a clue as to how this bacterium attacks human cells, and help in designing a vaccine which would produce antibodies recognising those protein motifs. Our initial investigation of this problem showed that BLAST sequence comparison tools [7, 8] are not designed to search for short motifs, and return no matches. The ACeDB system[8] [70] was also found not to possess sequence query capabilities.

With the protein motif problem in mind, two years ago, we switched attention from looking at automation of data processing in bioinformatics [183, 184] to sequence comparison methods which support genome-scale querying. In the meantime we discovered other possible application areas in which an efficient sequence searching solution is required. In particular, we know of two projects investigating human disease which require powerful sequence searching capabilities. Both involve using the published human DNA sequence

[5] http://chr21.molgen.mpg.de

[6] http://www.ncbi.nlm.nih.gov/BLAST/

[7] http://www.gla.ac.uk/Acad/IBLS/II/tjm/index.html

[8] http://www.acedb.org/. ACeDB is a data management and analysis system used very widely in genetics research. It provides two map viewers, a data editor and a query tool, plus interfaces to provide access to external software packages like BLAST.

to identify candidate genes thought to cause diseases, in combination with DNA data of a model organism for this disease (mouse or rat). In both cases sequence searching results have to be integrated with other data coming from external sources and lab experiments, and may involve statistical modelling. Research into large-scale genome-level sequence comparison methods is the leading interest of this thesis.

1.3 Thesis overview

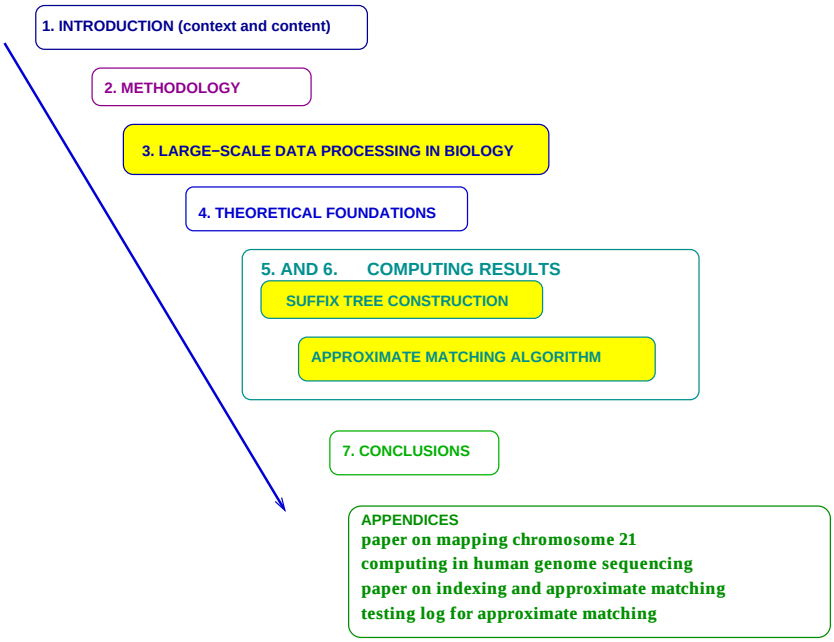


Figure 1.2: A schematic view of this thesis. Our contributions are highlighted in yellow.

We intend to demonstrate that sequence indexing can speed up large-scale sequence comparison. Figure 1.2 presents a graphical overview of this thesis. In this Introduction we presented the subject area by summarising the aims of life science research and introducing the terminology. In Chapter 2, Methodology, we reflect briefly on our research methods. In Chapter 3 we develop our argument with a discussion of new biotechnologies which give rise to very large volumes of data, and claim that efficient access to large data sets requires the development of indexing and other computing technologies. This leads to a focus on sequence data indexing. In Chapter 4 we introduce the computing theory used in this work, and in Chapters 5 and 6 we develop our argument further. We study a variety of indexes, and discover how to build very large indexes which were not possible previously. We then experiment with index building and approximate matching using a large index, and come to a conclusion that indexing is beneficial and requires further research. Additional research evidence is presented in the Appendices.

Chapter 2

Research methodology

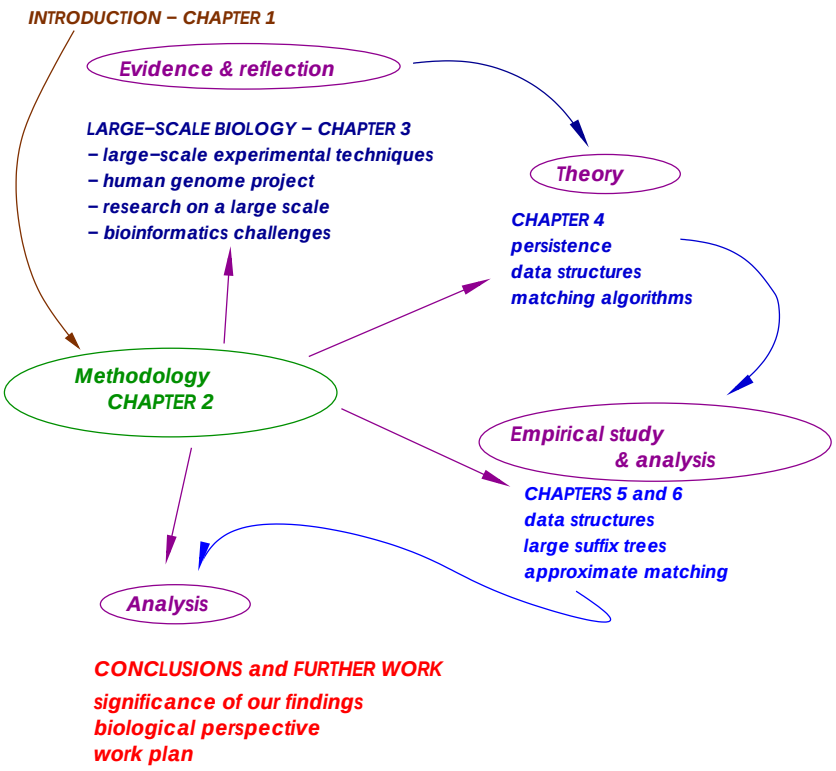


Figure 2.1: The role of evidence, theory, and empirical investigation in the research into new database mechanisms for sequence indexing.

Figure 2.1 outlines the structure of our thesis and the reserach methodologies we used. We used both empirical and analytical methods. Our initial approach was empirical and evidence was gathered by examination of existing software artifacts, in collaboration with biologists whom we observed during their bioinformatics activities. The second methodology was the use of computing and biological theory which provided a frame of reference for our understanding of existing software systems and a foundation for the design of new systems. The third method is the experimental work we carried out in an attempt to solve

the problems we noted in the evidence-gathering phase. And the fourth method was the analysis and evaluation of old and new artifacts, and reflection on how they could be further developed.

Figure 2.2: René Descartes, http://www.culture.fr

Our research methodology is reflected in the structure of this thesis. In this chapter we outline our methods. Chapter 3 draws together some of the *evidence* we gathered from interactions with biologists and the analysis of biological literature describing the use of software in biological discovery. It describes emerging large-scale technologies and the challenges for future bioinformatics work. In Chapter 4 we introduce some of the *theoretical foundations* of this work. Those include persistent programming language principles which allowed for transparent storage of application data, data structures which have a potential for DNA indexing, and exact and approximate matching algorithms which can be used in sequence matching. Chapters 5 and 6 describe the *experimental work* with the focus on the suffix tree and the suffix binary search tree, and the path we followed to develop an algorithm for the creation of very large suffix trees. We describe our research into approximate string matching using a suffix tree, and report all the results gathered during our investigation. This leads to Conclusions which used the technique of *analysis and reflection* to assess our achievement and draw a detailed plan of the work needed to deliver a prototype database index for approximate searching in DNA and protein strings.

2.1 Collecting evidence

Our motivation to explore the area of database indexing for sequence data came from the participation in the Chromosome 21 Transcriptional Mapping Project. Transcriptional mapping is synonymous with gene mapping, i.e. finding gene locations and their sequences. During that project we built a database integrating chromosome 21 data, delivered it on the web, and built an integrated map of the chromosome. For biological knowledge, we relied mostly on the information transmitted orally by the biologists we were working with. We found that the language divide between computing and genetics can be overcome slowly, given the motivation to collaborate on both sides. While working at the biological lab, we did not record observations on our research methods and results immediately, but the results of our mapping and database work were preserved electronically and are publicly available[1]. During the course of PhD research we decided to look back at our mapping work, and spent a month producing a manuscript [121] on the use of relational technology in genome mapping. That paper, reproduced in Appendinx A, was written to document the mapping procedure we followed, and was submitted to Bioinformatics just before the release of chromosome 21 sequence last year [103]. The reviewers recommended some minor changes. Because of pressing biological work, the co-author, Marie-Laure Yaspo, took a decision not to invest any time at that point, but intends to come back to it once her current work on the catalogue of chromosome 21 genes is complete, and release it as supporting material on the chromosome 21 website. We believe on our part that this draft will now be

[1] http://chr21.molgen.mpg.de

useful in the context of our future work on cross-genome mapping of human, mouse and rat genes, as needed in hypertension research that we are about to embark on.

Subsequently, we gathered further evidence about the use of available bioinformatics tools in the contexts of bacterial genomics, linkage analysis, and human somatic mutation simulation. The latter two have been documented as case studies [183, 184]. The method we adopted in this work was based on direct observation of the biologists using existing computational tools for their everyday work. A team of 2 people (myself and a research assistant David Jack) spent some time with the biologist in direct observation and a question and answer session where notes were taken and diagrams drawn. Case material was brought back, and analysed. During this analysis journals and web resources were consulted, so that a richer understanding of the biological issues was achieved, and then further questions were addressed to the biologists to clarify the outstanding issues. The case studies were then written up, sent back to the biologists for comment, and then discussed again. This led to the final technical reports and fed into grant applications and further research.

Other forms of gathering evidence were less direct but equally important. Three main sources of evidence come to the fore. One was participation in genomics meetings where new technologies and biological results were presented, the second one was the study of publications in the area of bioinformatics and biology, and the most important was participation in group seminars and meetings with biologists who explained their research. Attendance at external meetings was preceded by poster preparation which helped in the shaping and presentation of ideas [236, 122, 114, 115, 118, 116, 117]. Following the meetings, accounts of the issues raised there were prepared and published electronically[2].

Evidence gathering involved the study of bioinformatics literature. Nucleic Acids Research[3], published by Oxford University Press (OUP) has a methods section which regularly reports on bioinformatics methods, and devotes a yearly issue to bioinformatics databases. Bioinformatics[4], also published by OUP, reports monthly on a variety of computational problems and issues. Both journals report only methods that have been implemented and tested in biology, and where source code or a web site are available. Other biological journals also report on bioinformatics techniques, but quite often the coverage of computing science issues is so scant that the computing results could not be reliably reproduced based on the published articles. A slight departure from that tradition can be seen in the human genome paper in Science [226], analysed in Chapter 3, where the computing information is on a par with biology, within the confines of some 50 pages that were allocated to the paper. On the other hand, the Nature publication on the same topic [57] devotes much less room to computing techniques which influenced the success of the sequencing project and the annotation of sequence data.

Yet another way to gather evidence was to import bioinformatics software, or design and implement new software and use it to solve problems reported by biologists. We classify that under the heading of experimentation. We carried out the following experiments, some of them as software projects carried out by students.

- Management and searching of bacterial sequences, based on a whole-genome sequence of *Streptococcus pneumoniae*. This project went through several phases, from

[2] http://www.dcs.gla.ac.uk/∼ela/Report-Chester2000.html, http://www.dcs.gla.ac.uk/∼ela/Report-Cambridge.html

[3] http://nar.oupjournals.org/

[4] http://bioinformatics.oupjournals.org/

initial investigation, until the most recent prototype which is now being extended with additional functions. Initially, ACeDB [70] software was installed, and the genome, as well as predicted genes were loaded. It turned out that ACeDB did not have sequence searching facilities, and searching for short protein sequences in that genome was not possible using BLAST [7, 8] either. This experiment led to the follow-on work, described as next item.

- Experiment in the use of suffix trees for sequence indexing and exact searching. Ada code, provided by Rob Irving, implementing Ukkonen's algorithm for the construction of suffix trees, was re-implemented in Java [60], as further described in Chapter 5. That work was the starting point of empirical research with suffix trees.

- Implementation of suffix binary search trees in Java (translation from Ada) [125, 237] and further experimental work [120](see Chapter 5).

- Experiments in the design of genome browsers, i.e tools which allow for the examination of linear genome maps [134, 214, 221] and collaboration with David Leader who is working on a new version of a genome browser[5] allowing for cross-genome comparisons.

- Experiments in data integration for *Streptococcus pneumoniae* and *Streptococcus pyogenes* [87]. This project used MUMmer [66] to align the two genomes, a suffix tree to position 500 genes on the genome [113], ReadSeq sequence translation module [90] to translate between different sequence formats, and Artemis [15] browser and DerBrowser [97], redesigned by David Leader [118], to display data. BLAST searches were automated using Java and used to find sequence homologs of 500 genes.

2.2 The role of theory

We read a wide range of papers covering the areas of computing, biology and bioinformatics. Our study of biological subject literature (including Nature, Nature Genetics, Elsevier Trends in Genetics, Science, and Nucleic Acids Research) performed the role of widening our understanding of biological research issues. The study of computing methods and bioinformatics falls under the heading of theory, and requires a different approach from that needed in the study of biological issues. Computing literature covers a wide range of issues which can be roughly divided into theoretical and engineering subjects.

Our work made limited use of software engineering literature, and focused on using theory to produce new practical solutions. Theoretical computing science often uses graphical and algebraic methods to describe the idealised models of possible computations and data structures. Our background in software engineering and the expectation that this research is to be understood both by theoretical and practical computing scientists has tilted the balance more towards the graphical at the expense of the algebraic formulation. This makes our presentation accessible to a wider audience, at the expense of less concise presentation. As theory is not in our focus, we consider this departure from the usual style of theoretical investigation to be fully justified. This heavy reliance on graphical presentation probably reflects on the succinctness of notations and ideas used in computing, where mathematical and

[5] http://www.biochem.gla.ac.uk/Leader/Leader.html

algebraic terms are more information rich than they are in biomedical literature, and a full understanding of the implicit meaning can only be derived by analysing and checking the correctness of statements using additional graphical or algebraic forms of representation.

During the period of this research we accessed the rich resources of computational theory at times where further progress was thought to require deeper theoretical foundations. Simultaneously, out of curiosity, we read some superficially unrelated theoretical work and achieved enough appreciation of theoretical techniques to guide the shape of our experimental research. Within this thesis we give a reckoning of theory which informed most closely our work. This includes the foundations of persistence, string indexing structures and pattern matching algorithms. Our broader interests which include agent technologies, work-flows, software-engineering, semi-structured databases and XML [3, 91] will remain unaccounted for.

Theory is not a research issue within this thesis but a constituent part of our engineering research, and its role is to assist us in the building of future sequence databases. In this area we studied Ukkonen's work on suffix tree construction [224] as well as Kurtz's work on space reduction in a suffix tree [138]. We devoted considerable time to this research, only to discover that, surprisingly, our testing results *do not show* that a time-optimal suffix tree takes less time to build in Java as a sub-optimal *naive* one. We now think that this is due to the fact that worst-case analysis is not a universal predictor of programs behaviour, and that current models of computation do not reflect adequately the complexity of the full memory hierarchy in modern computers.

Our work directly reveals the limited power of worst-case analysis in making the connection between the time and space complexities of a computation. On the other hand, we appreciate the available theory, which in most cases correctly predicts that some experimental avenues are dead-ends and provides a good approximation of the performance of algorithms.

We also need theory to throw light on the way persistence can be achieved. This is a different type of theory which focuses not on complexity but classification of the computing universe using types and properties of computational processes. This theory does not use quantitative measures of goodness, but defines abstract classes of properties and abstract objects which we use to describe the world of computing artifacts we create. In this theory the language consists of terms like "persistence", "transactions", "scalability" and others described in Chapter 4. This vocabulary, specific to computing science, and database research in particular, allows us to describe computations from a more practical perspective, related to the use of resources, and the scheduling of tasks which model operations on data. Our work relies on this theory, but again the theory is not our focus, and we limit our interest to the most important features of the computing universe which we come into contact with.

The interplay of theory and experience is important in this work. The computational theory we have at our disposal is a simplification of the actual computations we perform, and the database theory we are using, that is orthogonal persistence, is constantly being refined. In our experimental work with persistent systems, there is no accepted theory yet to describe the complex interplay of disks and caches with operations in the Java language which have different computational costs. Tips to software developers [71], on what operations to use in their programs to achieve fast computation are practical intuitions and would require research to be made into science.

As an illustration of the complexity of comparing time measurements we have to abstract from is our comparison of two short Java programs of identical functionality, one

using String manipulation and Java Collections, the other one using byte arrays. In a test with 300 MB of data, we observed a ratio of execution times of 60:1. On another occasion, we executed the same Java program which constructs suffix binary search trees on two hardware configurations, and the execution times ratio was 20:1, possibly due to a difference in Java optimisation strategies. It is hard to say, perhaps because we are using different implementations of Java or running different implementations on different hardware platforms, if an accurate generalisation of operation costs and space/time tradeoffs can be derived in this context. Such phenomena are not limited to Java, and understanding of the space/time and hardware/software tradeoffs in different contexts would be a useful foundation for software engineering.

2.3 Empirical work

Our experimental work consisted of software reengineering (improvement of Java software produced by students), algorithm design, implementation from scratch, and testing. Standard Java was used alongside persistent Java, PJama [21, 20]. Successive code versions and Java/PJama versions were used during the project, and were tested with increasing amounts of genomic sequence. There was gradual progress in the amount of DNA sequence we could index. The first suffix tree could index less than 2 Mbp of source sequence. After code reengineering, trees for up to 25 Mbp became possible in memory (using 2 GB RAM). Then persistent trees for up to 15 Mbp DNA followed, and after an improvement to PJama [155], trees up to 20.5 Mbp were constructed.

Then the suffix binary search tree was produced in Java [237]. Some re-engineering was required to reduce the computation time on the Solaris OS. Transient trees for up to 40 Mbp were built, and persistent trees for some 50 Mbp followed.

Performance tests were then carried out on five tree structures, with exact matching and construction time measured for all.

The work switched then to suffix trees, and after studying Kurtz's work on space-optimal suffix trees [138], we decided to investigate space optimisations. To do that more simply than with the optimal Ukkonen's algorithm [224], we developed our naive suffix tree which is not time-optimal but more space-efficient than Ukkonen's tree. This led to experiments with larger suffix trees, and then to the algorithm for building a suffix tree in stages, currently for up to 300 Mb of source data, which is one of the main achievements of our research.

Subsequently, we implemented and refined an approximate matching algorithm using the naive suffix tree, and we performed various benchmarking tests as described in Chapter 5, Chapter 6, and Appendix B.

2.4 Software engineering methods

Software development activity consumed probably around 15% of the total of just over 3.5 years devoted to this research. Two types of programs were produced: Perl [228] scripts, used as a job control language for large-scale testing and a data preparation language, and Java classes which constituted the core research. The Java software which implements different tree structures consists of two main classes for each indexing structure we produced.

One class defines the nodes used in a tree, and the other class builds the tree and implements searching methods. Each index structure uses between 500 and 600 lines of Java code.

No software engineering tools were used. We used "nedit"[6] for word processing and found the development of Perl modules far more time consuming than necessary. Tracing errors in Perl programs was hard, because the language does not have any typing rules at all. We would have liked to have access to Java libraries capable of high level manipulations of ASCII files. For instance, high level functions which would recognise records within a data file, write them to a temporary file, submit as queries, and report timing and results to two separate files, would have been of great use. Such tools would be very useful to biologists as well, and may indeed be developed in the future (similar to data wizards in some office tools, but more flexible).

We did not use a code versioning system, but filed software according to production date, which corresponded to the progress we were making. As soon as testing results were produced, we wrote up the latest results as a report or paper, and filed the results data.

2.5 Research planning

We found it very difficult to plan work. Methods for work planning are not well taught, and managerial planning of software projects, which we had studied, did not prepare us for research planning. Software project planning is based on requirements analysis, while research planning can only be done once research requirements are known. Identifying the deliverables of a research project is the hardest part of planning. Once a deliverable is well identified, decomposing it into constituent parts and planning the actual work is easier. The role of experience in planning is essential. We believe that by having been encouraged to write funding applications, we made good progress towards learning the technique of research planning.

We prepared 2 grant applications to BBSRC which were not successful but fed into work currently done by a group of researchers here. We successfully applied twice for Masters and PhD studentships to MRC. We applied successfully to MRC for a personal post-doctoral fellowship in bioinformatics and we helped in the preparation of a ROPA grant to EPSRC which was successful. During last three months we also submitted a PhD studentship application to MRC and a bioinformatics summer school application to EPSRC. We contributed to a Wellcome Trust aplication for bioinformatics support for hypertension research, and to a British Council application for travel funds to support bacterial genomics research. We believe that preparing grant applications is a very useful part of the research methodology employed in this project. During the periods of grant preparation, which altogether consumed up to 6 months of effort, we performed a lot of research into existing computing technologies and research trends, and into biomedical issues. These periods of intensive effort included contact with biologists, which helped to refine our understanding of the issues we were addressing. The writing process itself was also very useful, as it helped us in phrasing our concerns in well-formed language. Another benefit was that the project plans we produced shaped our research, and helped us in day-to-day management of the process by refining our scientific goals and the strategy to achieve them.

[6] http://nedit.org/

2.6 Interdisciplinary aspects

Our interdisciplinary research consisted in a large part of interaction with biologists and medical researchers. We estimate that at least 10% of our research time was spent in meetings with biologists and attending biological seminars, and some additional 5% in reading biological literature. We attended weekly seminars and met with the neurodegenerative diseases group led by Keith Johnson[7] which helped us to understand the research techniques this group uses. We met with Tim Mitchell who works on infectious diseases[8] and conducted student projects in collaboration with him. We had several meetings with Julian Dow[9] who researches *Drosophila melanogaster* physiology. We also attended group seminars and had meetings with the cardiovascular research group at the Western Infirmary led by Anna Dominiczak[10]. We also met virologists, protein scientists, and statisticians to discuss joint research. We found that frequent contact with biologists was time-consuming but necessary. We maintained regular contact, with the exception of several months with very intensive software development and testing activities when we did not attend regular meetings, but had some individual meetings instead. We benefited from our collaborations and we believe that the biologists will trust us in the future. This is currently the case, as several collaborative grant applications are in preparation, and two have now been submitted.

2.7 Reflection

While preparing to write up this thesis, we attempted to gather information about research methodologies relevant to computing science. We found little relevant information. Websites with course summaries and book contents lists present summaries for research methodologies in computing science which are not adequate in our opinion[11]. Disappointingly, these research methods are limited to statistical data manipulation and computing literacy. There is little reflection on research methodologies which would tell us what issues are worth researching, how to formulate research questions, and how to prepare meaningful answers to the questions we ask. There is a vacuum there which is worth addressing.

There is however one notable exception. Tichy addresses the need for experimental [220, 219] research in computing science. He encourages experimentation for the following reasons:

- a computing theory will be accepted if it explains all facts within its scope and is borne out by experimentation,
- an experiment has the power to contradict theory and overthrow it,
- experiments can help explore areas where theory and deductive analysis do not reach, and help with induction: deriving theories from observation.

We believe our research falls within the third area, as there is currently no theory adequately describing the complexity of persistent systems. We hope that by testing index structures in

[7] http://www.gla.ac.uk/ibls/molgen/staff/johnson-kj.html
[8] http://www.gla.ac.uk/ibls/II/tjm/
[9] http://www.gla.ac.uk/ibls/molgen/staff/dow-jat.html
[10] http://www.medther.gla.ac.uk/bhf/bhfafdcv.htm
[11] see CMPUT 601 "Research Methods in Computing Science" at http://www.cs.ualberta.ca/ jeffp/cmput601/

different contexts, enough understanding will develop, so that perhaps new theories will be formulated in this field.

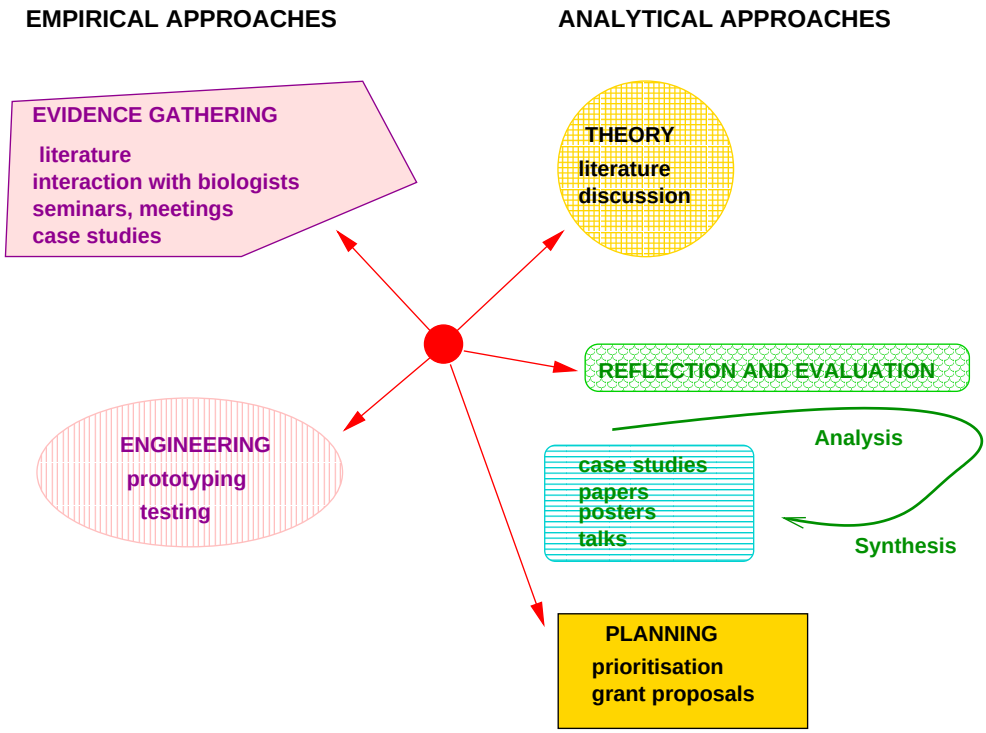


Figure 2.3: Empirical and analytical research methods which contributed to our suffix tree research.

As we found no literature which adequately addresses computing research methods issues, we summarise our current understanding of the way our research developed. Figure 2.3 outlines the main components of our inquiry. The research we performed can be roughly grouped into two kinds of enquiry: the empirical and the analytical work. Experience was the main motivating factor, and a wide range of activities which we undertook helped us to build a picture of the needs in the bioinformatics data analysis which we could remedy using persistent technologies. To get a full understanding of the experience we gathered, we subjected our experimental finding to analytical scrutiny. In this analysis we were guided by theory, and we prioritised possible work in the light of our understanding and experience. We drew plans which materialised as grant applications, and moved on to another round of experimentation - this time by constructing new computing artifacts and testing their performance against imported software which we used as benchmarks. Throughout the research process we regularly reflected on our understanding of theory and of the experiments we carried out, and wrote case studies, research reports and papers which provide a synthesis of our progress and our of understanding of the field.

With this short summary of research methods we close this chapter, and direct the reader's attention to Chapter 3 which discusses the biological evidence justifying the need for new database solutions to biomedical data management.

Chapter 3

Large-scale data processing in biology

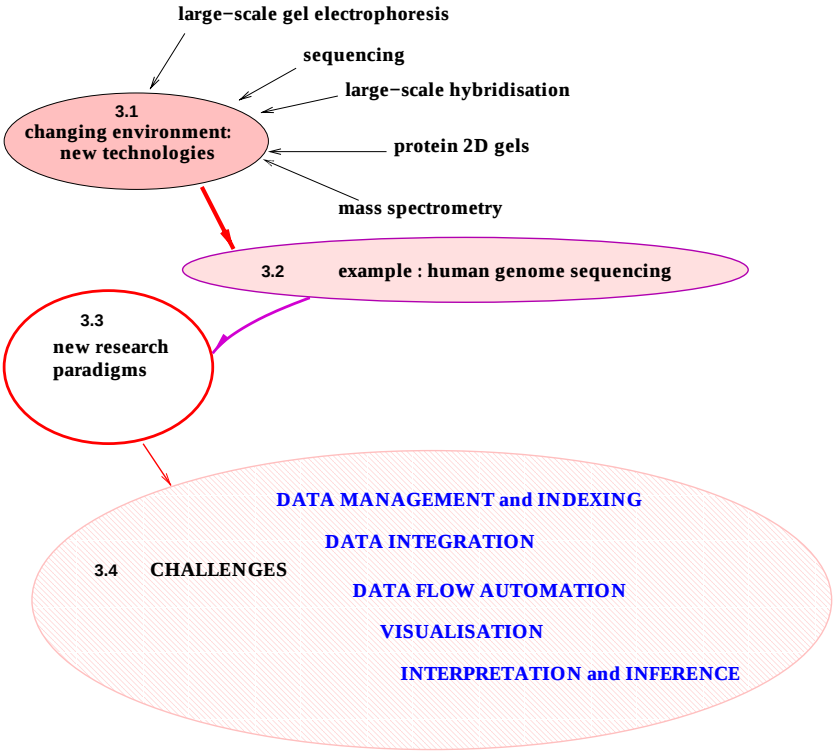


Figure 3.1: Overview of Chapter 3.

The speed of biological discovery seems to be continually accelerating. This results from improvements in science and technology including physics, chemistry, engineering, robotics, and optics, combined with the increasing power of computer hardware. The result is the availability of large-scale testing technologies (many thousand of data points in parallel) providing data to the biologist who is currently underequipped in terms of software, hardware, and experience to securely store, archive, index and conveniently manipulate those data sets.

In this chapter we focus on this sudden increase in the data generation capacity which is one of the main characteristics of a modern research lab. We show examples of relevant technologies and resulting datasets. We introduce those technologies from the perspective of data management and processing they require, and point out the features that they have in common. We also look at the first significant application of those large-scale technologies to the delivery of the human genome sequence, which illustrates the need for specialised computing support in this area. This leads us to a discussion of a new research paradigm, which we call data-driven research, and to a short analysis of the future challenges in bioinformatics which result from the application of large-scale data generation technologies in biology. Figure 3.1 summarises the plot of this chapter.

3.1 New technologies

We review the following: gel electrophoresis, sequencing, microarrays, protein gels, and mass-spectroscopy data. This leads us to consider the commonalities between those large-scale data production techniques. We finally argue for the preservation of those data sets, with appropriate annotation, so that they can be re-used and re-examined in future research.

3.1.1 Large-scale gel electrophoresis

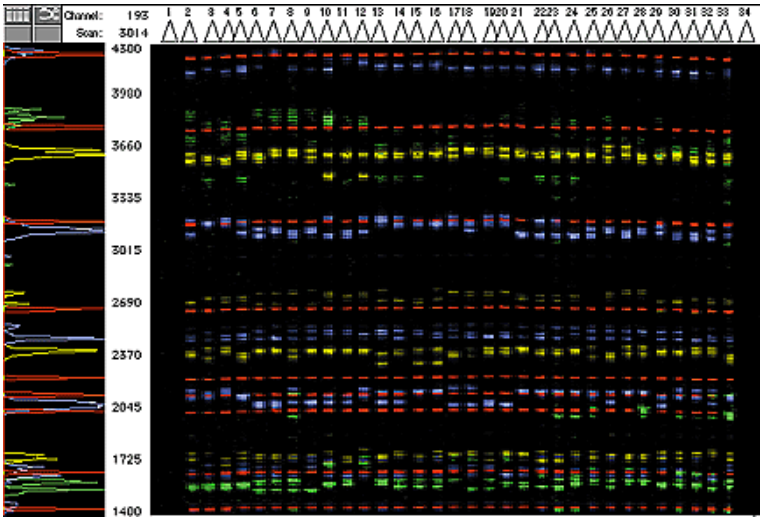


Figure 3.2: Large-scale gel electrophoresis

Large-scale gel electrophoresis delivers images representing up to a few thousand data points, see Figure 3.2[1], where the horizontal dimension is used to number the lanes in which DNA samples are placed, and the vertical dimension represents the distance a DNA fragment travelled in the gel in the electric field (which is proportional to fragment length). A single experiment may involve pooling DNA from blood for groups of individuals, using PCR (polymerase chain reaction) to selectively amplify (increase the amount of) a particular DNA fragment, and placing the samples in gel lanes in an electric field. The distance

[1] http://www2.perkin-elmer.com

travelled by each fluorescently labelled fragment is translated into fragment length by the associated software. Individuals who provided the samples will be grouped according to the particular combination of those fragment lengths, and their genotype will be represented as a fingerprint of those lengths in some predefined order. Such tests produce a list of DNA lengths for the selected fragments for each individual, and this data can further be used in locating disease genes, or in criminal and forensic investigations. The underlying biological explanation for this use of gel electrophoresis is that stretches of DNA differ between individuals, but family members share similar DNA fragments, and those are inherited in contiguous pieces together with neighbouring genes. For a study of a hundred individuals, and 300 markers, two lengths will be recorded per each individual which leads to some 60,000 data points. In a search for a disease gene, using the technique of linkage analysis, medical data will also be stored about each individual, together with the family trees for each family. Several additional data sets will also be used in the process of finding the gene position, as described in our previous work [183].

3.1.2 Sequencing

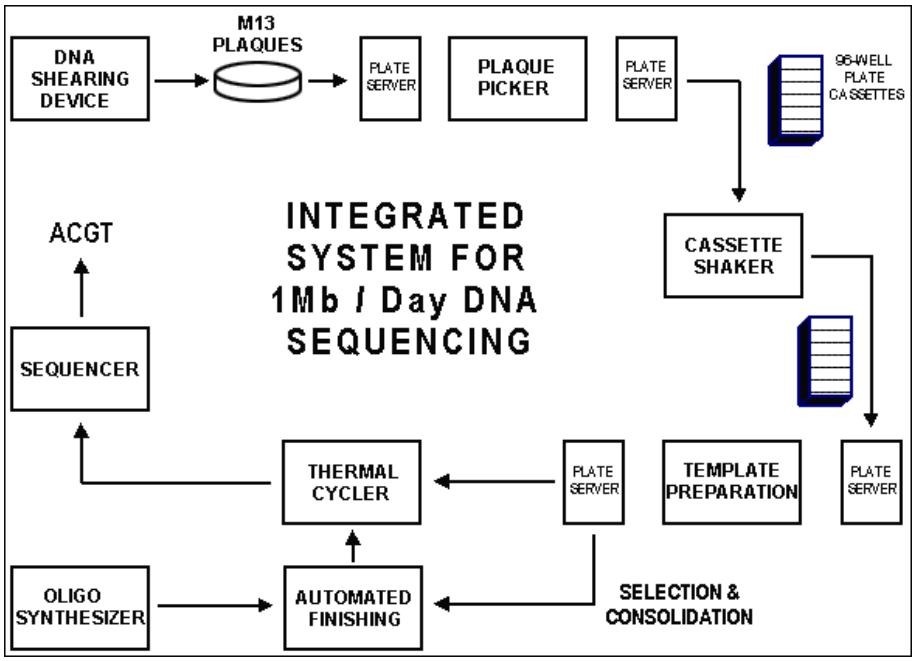


Figure 3.3: An integrated high-throughput sequencing system at Stanford University. DNA is cut with enzymes and inserted in carrier microorganisms where it multiplies. Those samples are placed in cassettes and then stacked, subjected to further processing and finally passed on to the sequencer.

Powerful sequencing technology, see Figure 3.3[2], which has now produced an almost-complete sequence of the human genome [226, 57], uses the same underlying gel electrophoresis technique as described in the previous section, but this time applied to produc-

[2]http://sequence-www.stanford.edu/group/techdev/auto.html

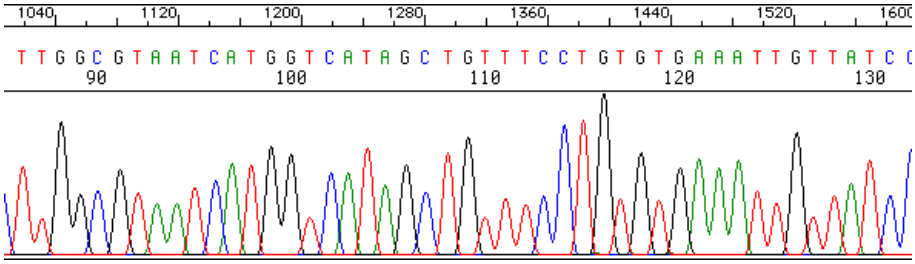


Figure 3.4: A sequence trace produced from a sequencing run.

ing a coloured band for every base sequenced. An introduction to this technology can be found on the web[3]. Gel images are assembled by the computer into colour-coded traces (see Figure 3.4), and finally represented as strings over the alphabet of 4 letters (A,C,G,T), with occasional N for bases which could not be determined. Those are delivered as ASCII files of variable length, generally less than 800 characters. Those fragments are then "cleaned" to remove the sequence of the carrier organism and assembled using string-matching algorithms enhanced with statistical measures of error. Currently, the volume of public DNA sequence data exceeds 18 GB, where one DNA letter is encoded as a byte. The source data leading to the final assembled sequence is a multiple of the published size, as data of high quality can only be achieved by re-sequencing the same pieces of DNA several times. For instance, in the human genome project, 8-fold sequence coverage was required, Celera now have a 6-fold coverage of the mouse genome which they have assembled[4], and the public mouse sequencing project has now achieved 4-fold coverage. Beside human and mouse genomes, rat, chimpanzee, and dog genomes are being sequenced, each of those being in the region of 3 GB of finished data, i.e. 15-30 GB of raw sequence each. An overview of world-wide sequencing work can be seen at the Genomes OnLine database at http://wit.integratedgenomics.com/GOLD/.

3.1.3 Large-scale array experiments

A large-scale hybridisation image produced using nylon filters is shown in Figure 3.5[5], and a microarray slide in Figure 3.6[6]. Those techniques immobilise thousands of catalogued sequence fragments on a nylon membrane, glass, or other base, and a radioactive or fluorescent sample is applied on top, in order to find affinity or detect reactions between the sample and the immobilised fragments. Affinity means that the probe on the filter and the sample have complementary sequence (come from the same DNA fragment), or bind biologically in some other manner. Current array technologies mainly involve DNA, but new techniques for protein immobilisation and protein-DNA reactions follow the same pattern. The resulting data is a combination of the image and subsequent evaluation, for instance a table including spot identity and the level of signal recorded. Associated information should include a description of the experiment [38], technical data from the scanner and software used, the name of the biologist and date. Data points may be gathered as an image or a

[3] http://seqcore.brcf.med.umich.edu/doc/educ/dnapr/sequencing.html
[4] http://www.celera.com
[5] http://www.molgen.mpg.de
[6] http://www.nhgri.nih.gov/DIR/LCG/15K/HTML/img_analysis.html

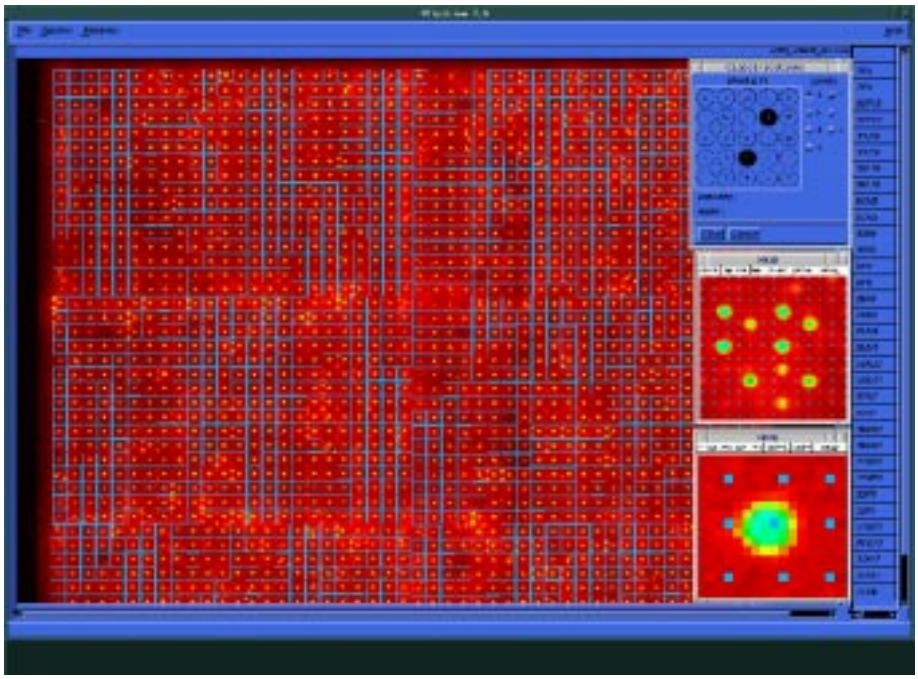


Figure 3.5: Large-scale hybridisation on a nylon membrane, involving over 20,000 data points, viewed using an image-analysis program Xdigitise, from the Max-Planck Institute for Molecular Genetics (MPIMG) in Berlin.

series of images with time stamps, with an associated map of spot identities and linked to the underlying sequence placed in each spot. In a typical experiment, thousands of spots are present, and depending on technology and the required resolution, between one and four images are taken per time point. Each image takes up to 5 MB storage, and in a time-series experiment with 20 time points, this may lead to 100 MB of data. A complex investigation will use control arrays as well as examine a range of different experimental conditions. We estimate that persistent storage needed to record one complete investigation may be in the region of a few GB of images, plus some extra space for the associated annotations and derived results.

As an example, we quote an early use of large hybridisations in the mapping of parts of human chromosome 21, in which we were involved. To prepare a detailed map of a part of this chromosome (totaling 3 Mbp i.e. 1% of the human genome) before sequencing [105], large-scale hybridisation experiments were carried out. Filters containing over 50,000 clones (each clone in duplicate) were treated with 112 probes (at least one image per probe). Filters were then digitised, signals scored semi-manually using a computer program Xdigitise (shown above), and the signal for the spots which showed a hybridisation was scored as weak, medium, or strong. This procedure led to the identification of approximately 2000 clones which mapped in the area of interest and were used in further analysis. Minimum image storage was for 112 TIFF[7] images of 5 MB each (0.5 GB in total), and the resulting clean data (after removing hybridisations with too many hits) lead to the identification of over 2,000 clones, out of which 27 clones and 50 short unique sequences (STS)

[7]Tag Image File Format, http://www.libtiff.org/

Figure 3.6: Microarray image involving two dyes (green and red).

were selected for use in sequencing.

3.1.4 Proteomic techniques

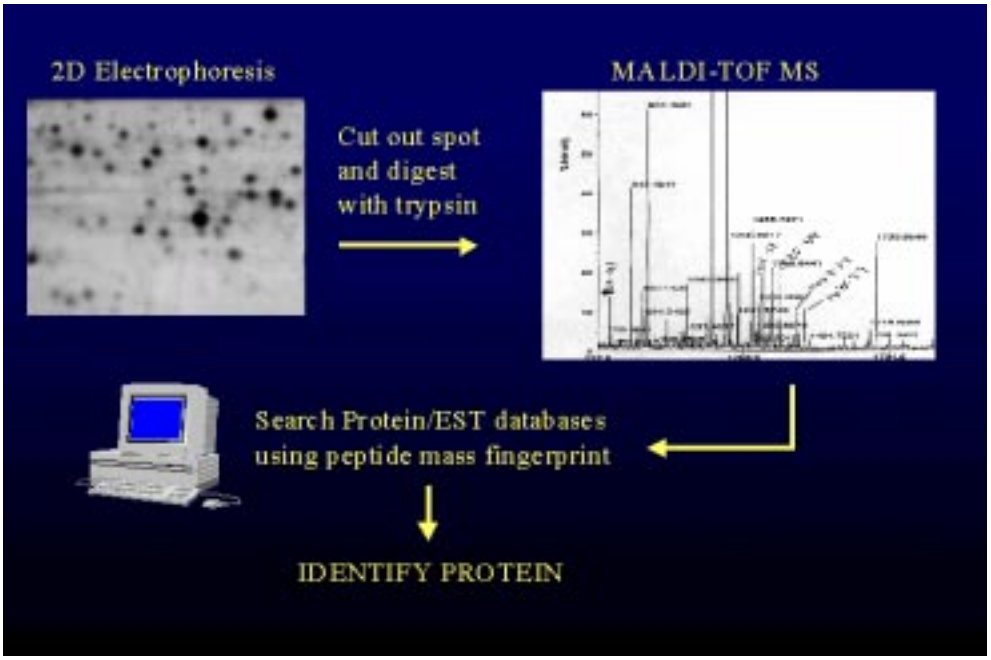


Figure 3.7: Protein separation and analysis of proteins using mass spectrometry and database searching.

The procedure for two-dimensional protein separation is illustrated in Figure 3.7[8]. In that experiment a living cell or some cell fractions are homogenised, and the sample is placed on a gel which has both a pH and an electrical gradient. This leads to two-dimensional separation of proteins. The gel is photographed, and can be used as a fingerprint of the tissue and condition in question, and compared using image analysis to other samples. However, new methods, in particular mass spectroscopy, can now be used in combination with this procedure. Robots can excise some or all of the protein spots, and pass them on to the mass spectrometer. Resulting data consist of images with corresponding annotation of the

[8] http://www.gla.ac.uk/ibls/II/jw/webmaldi.jpg

sample, including the procedures and equipment used. Image storage is the most bulky component of that data, and may require up to 5 MB per image. The number of potential images will depend on the design of a particular experiment, and may be in the region of 50 to 100, allowing for duplicates to ensure good quality of results, i.e. leading to up to 500 MB of disk storage per experiment.

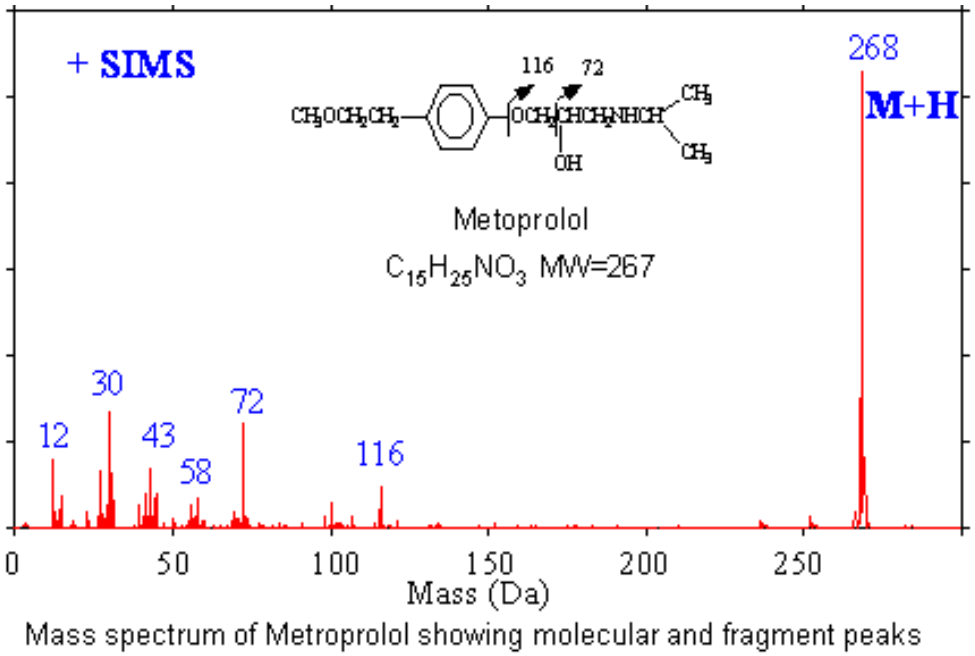


Figure 3.8: Mass spectrum example.

Mass spectrometry, see Figure 3.8[9], can now follow on from protein 2D analysis. A spot picked up from the 2D gel is broken enzymatically into smaller fractions, and the molecular weights of those fragments are determined. By analysing databases of known proteins, matches against known protein fragments are found, and some proteins can be characterised, or close relatives found. Output data from spectrometry will consist of a trace or a list of molecular weights found. Repeated application of digestion and spectrometry can deliver a more exact resolution of the sample.

Other new technologies are also appearing in this context, and like microarrays, and mass-spectrometry, they will deliver large-scale data sets. For instance, protein-protein interaction data from the Yeast Two-Hybrid Screen[10] will result in large images, and protein localisation and activity information gained using indirect immunofluorescence [188] data will also contribute significantly to the data volumes.

3.1.5 Common features of new biotechnologies

These technologies share the reliance on the underlying image which is then processed using image-analysis tools, and leads, via statistical techniques, to numerical data used in further

[9] http://www.phymetrics.com/gen.asp?GID=283
[10] http://www.uib.no/aasland/two-hybrid.html

research. A substantial amount of source data evaluation happens before numerical results are produced (image analysis programs vary in the way they process and deliver data), and data are also produced by robots, microscopes, and other hardware and software.

Data storage requirements

Experimental research in biology often combines sequencing, microarrays, proteomics, and digitalised microscopy images. Assuming an image size of 5 MB, 5 time-series experiments of 20 time-points each (each scanned twice), plus 20 protein gels, and 10 images from a microscope, images for one experiment may require around 1 GB of disk storage. Sequence data and annotation will slightly inflate that number. It is hard to estimate the global volumes of data produced using all those technologies. However, looking at the web pages of the market leader in microarray technology, Affymetrix[11], we learn that up to the 31st of March 2001, 450 Affymetrix systems (arrayers or scanners or both) have been sold, accompanied by 70,000 GeneChip units, usually consisting of several chips. The price of Affymetrix GeneChips is still high (over £300 per chip) and much more chip development is done independently, using either Affymetrix systems or other equipment.

There is no doubt that keeping all the images in their original form, as captured, is a prerequisite for their proper use in the future. There are several reasons for that. Images cannot be re-acquired from old microarrays, as the microarray fades during scanning and storage. Re-acquisition of images can only be done by repeating the expensive experiment. Software that currently exists is still imperfect, as technologies of image analysis will be improving to match the requirement for better image resolution. And statistical techniques of data analysis will also be changing so that old images will have to be re-assessed using new techniques.

We believe that data storage needs resulting from the large scale data generation technologies will be hard to meet. Techniques like storing data on a CD-ROM are no longer adequate. If we assume around 1000 research groups which carry out biomedical research at Glasgow and Strathclyde Universities and use those technologies, if each of them carries out 5 such experiments next year, (each experiment producing 1 GB of data), we arrive at a figure of 5 TB per year, which is entirely attributable to new technologies. This is a real challenge to data management and indexing, if this data is to be made available to all interested scientists.

Annotation

To be of value in future research, image data have to be properly annotated. Efforts to standardise the annotation of microarray data are under way[12], and minimum annotation will have to include the descriptions of the experimental design, array design, samples, hybridisation procedures and parameters, measurements, and normalisation controls. Those annotations, as well as the identity and brightness of spots in the images will allow for data indexing and further analysis. Standards for the annotation of data produced using other techniques will have to be developed along similar lines. Such standards will be required for protein data in particular.

[11] http://www.affymetrix.com/

[12] http://www.mged.org/Annotations-wg/index.html, http://www.mged.org

Aspects of data use

The integrated use of complex data sets presents a significant challenge. For instance, solutions to sequence assembly and analysis, in the Human Genome Project, were created by putting together pipelines of software programs which preprocessed, analysed and merged source traces from the sequencer into contiguous stretches called contigs. Such pipelines are now becoming available in the sequencing context, but they need to be integrated with laboratory information systems (LIMS) which track sample preparation and progress, and may have to be adjusted to match new hardware or procedures, i.e they have a high associated software engineering cost. Similar pipelines will be constructed in the future for mass evaluation of microarrays, protein 2D gels, mass spectrometry, and other data, and will call for software engineering solutions which can hardly be met cheaply using our current technologies. It is clear that data and software management to match the biological needs will have to be developed, and deliver solutions which are both reliable and extensible. This is a great challenge because of data and software diversity. There is a requirement for new and more powerful indexing technologies, new approaches to data compression, new software engineering solutions and new more powerful databases.

Focus on sequence

Out of the sea of possible bioinformatics research issues arising from large-scale data generation, we selected one. Our focus is on sequence data, and therefore, in the next section, we explore data processing operations which had been used to produce the draft of the human genome sequence, the main experimental material of our algorithmic research.

3.2 The delivery of the human genome sequence

In this section we compare the data management approaches used in the public genome sequencing effort [57] with that used by Celera Genomics [226].

The aim of the Human Genome Project[13] is to identify all human genes and their protein products. To achieve that aim, the human genome is being sequenced, and the public effort is coordinated by the International Human Genome Sequencing Consortium (IHGSC), while the private project is carried out by a company called Celera. We now proceed with a comparison of both approaches.

3.2.1 Two sequencing projects

After a long preparatory period, from 1988 to 1998, public funding bodies agreed to sequence and publish the human genome by 2005. Celera Genomics, led by Craig Venter who previously worked in the sequencing of smaller organisms [83, 106] and had left academia to form his own company, announced in spring 1998 that Celera would sequence the human genome by 2001. This caused some vivid discussion, and the public consortium decided in the autumn of 1998 to change the deadline and deliver a draft sequence by 2001, and a fully annotated complete sequence by 2003. In the meantime, Celera enlisted the help of experts in theoretical computing science (Arthur L. Delcher, Gene Myers, Michael S. Waterman, and others), and tested their approach with the much shorter sequence of *Drosophila*

[13] http://www.gene.ucl.ac.uk/hugo/

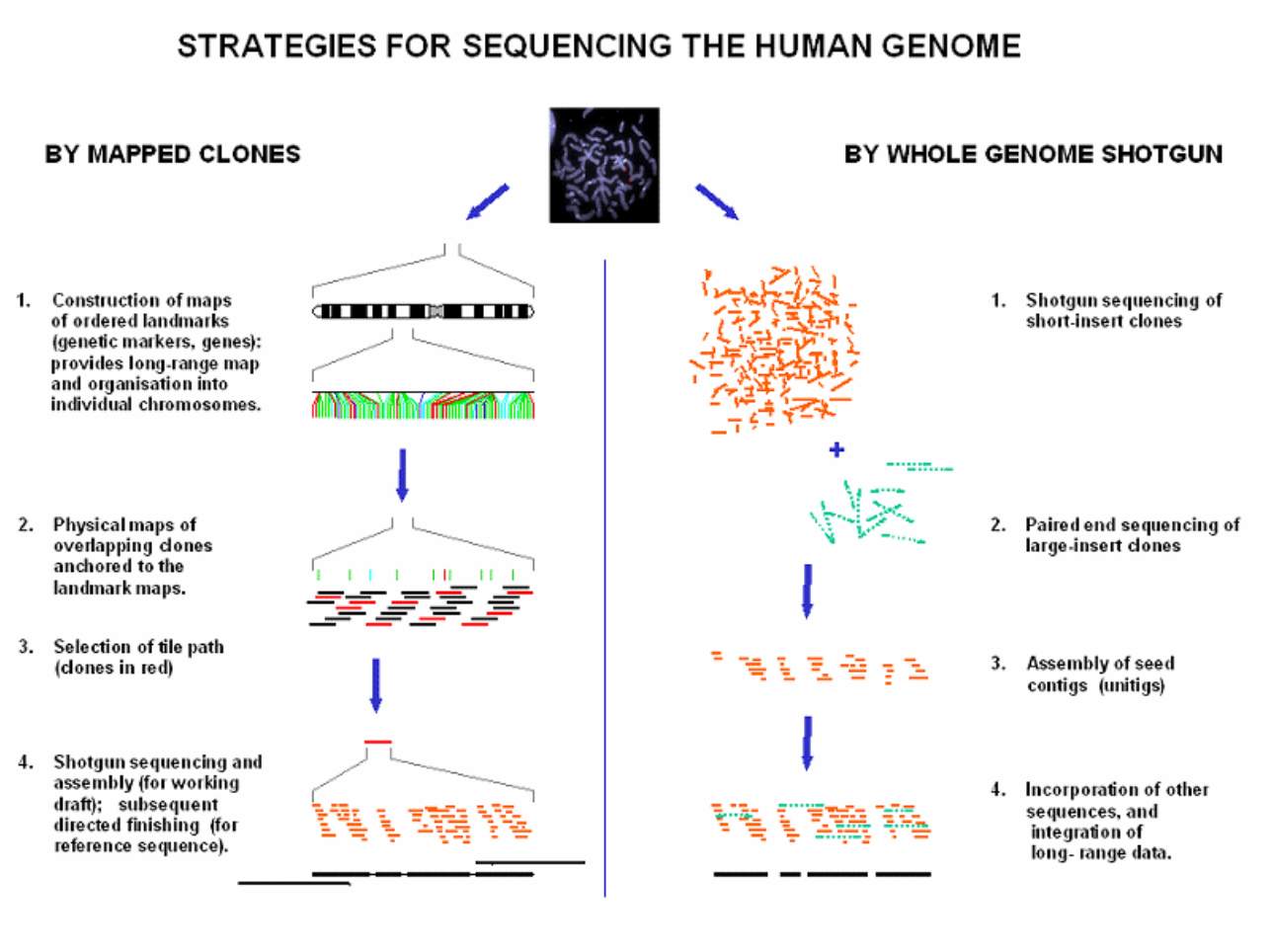


Figure 3.9: Competing genome sequencing strategies.

melanogaster [165, 4], in collaboration with a research group of Gerald Rubin [189, 108] at Berkeley. *Drosophila* sequence was delivered in part, i.e. 120 million base pairs (Mbp) out of the total of 180 Mbp. This failure to deliver a more complete sequence showed an underlying weakness in the purely computational approach in the areas of genome where repetitive sequences abound, and in the final pinning of assembled fragments to chromosomes, which had to rely on high resolution genome maps requiring considerable skill and effort to prepare[14].

Figure 3.9[15] compares the strategy adopted by the public consortium (sequencing by mapped clones) to the approach taken by Celera (sequencing by whole-genome shotgun). Celera skipped the labour-intensive step of mapping of individual clones to chromosomes. Instead, they used publicly available data from the IHGSC databases to position their data on the mapping scaffold created by the publicly funded consortium.

Given no access to public-domain maps and data, Celera's sequencing effort would

[14]Maps were constructed using BAC clones and sequence tagged site (STS) markers, and relied on a very large body of screening experiments

[15]http://www.sanger.ac.uk/HGP/draft2000/gfx/fig2.gif

be of limited use, as it turns out that the human genome consists of up to 50 per cent of repetitive sequence which cannot be resolved by sequence alignment alone. For instance, a pattern of 2 or 3 letters repeating over several thousand symbols in the human genome in several positions cannot be computationally resolved, because current sequencing machines can deliver up to 750 bases in each run, and the length of the overlap of repetitive stretches is impossible to determine computationally. Therefore, a fully automated assembly is difficult, and reference to chromosomal assignment (position of each fragment on a chromosome, orientation and order of fragments) is the minimum required[16]. To overcome that problem, Celera used public mapping data as well as a 3-fold sequence (9 GB) from public genome resources in their assembly process, so that sequence fragments could be oriented and ordered and the coverage of the genome could be complete enough to enable assembly. As Eric Lander, leader of the Whitehead Institute / MIT sequencing lab, put it during the recent Human Genome Conference (May 20001, Edinburgh):

> *Celera took the publicly available copy of the human genome, which was 95 percent complete, mixed it with its own version of the genome, and SURPRISE!, what came out was a 95 percent complete version of the genome.*

We now look at the computational issues in the human genome project. A listing of hardware and software used in both projects can be found in Appendix B. Next section provides a high-level overview.

3.2.2 A computational comparison

The task of the genome project is to deliver the annotated sequence for the human genome. This work consists of two phases, one of them being the creation of sequence, and the other, sequence annotation.

Sequence production consists of sequencing, sequence cleaning and assembly. Contaminants and the microorganisms in which human DNA was replicated will be sequenced as well as the human DNA. There are also errors in reading the gel which have to be corrected for. Data cleaning involves both rejecting foreign DNA and correcting for mis-read bases. Then, assembly for each clone will be carried out separately, based on the lengths of DNA inserts as measured on gels by electrophoresis [177]. The position of each clone in the genome is determined by a variety of laboratory techniques (PCR or hybridisation or another test for sequence identity), and based on this positional information, the full genome sequence can be produced.

Stage two is the annotation. Genes can be predicted from the sequence using purely computational techniques, which succeed to some extent, but do not recognise all genes, and miss some gene parts. Genes can also be predicted based of the knowledge of genes known from other organisms, or based on genes expressed in cells (which might be turned into proteins by cell machinery). Often, before gene prediction is carried out, the sequence undergoes a filtering process in which repetitive sequences are removed. Subsequently, gene prediction and annotation can be carried out just on the DNA, or on the DNA conceptually translated into protein.

[16] In the subsequent assembly of the mouse genome, known human genes and human-mouse synteny maps can be used in sequence assembly.

Sequencing, sequence assembly and gene function prediction are all currently based on computational techniques. Therefore we start this comparison by looking at the computing resources employed by the public consortium and by Celera.

Appendix B presents more detail gathered from [226], and [57]. A summary of the publicly released human genome data can be found on the web[17]. Access to Celera's data is via their website, after previous registration, guarded by password and non-disclosure agreements.

Hardware

It is hard to compare the computing resources exactly. The public consortium consisted of over 16 parties, and the main contributors were the Sanger Centre and the Whitehead Institute. We feel that estimating the overall computing power for all public project participants is hard, as the information on the computing setup does not clearly distinguish between sequence production and sequence analysis. Sequence production was distributed between the sites, and sequence assembly and analysis were partly distributed as well. The Sanger Centre site, which produced roughly half of the sequence, used a heterogeneous computing environment managed using a Load Sharing Facility (LSF) software provided by Platform Computing Ltd[18]. A detailed listing of the hardware now used for annotation by the Ensembl project is presented at http://www.ensembl.org/Docs/computation/.

It appears that both sides had access to very large computing resources without which the computations required in the project would not have been possible. Celera's computers had larger RAM sizes (34 GB RAM for analysis and 64 GB RAM for sequencing and assembly), and that allowed them to make an attempt at whole-genome assembly, which perhaps would not have been possible in the public project, unless additional machinery had been acquired. However, for the final assembly of contigs significant computing resources were used [133] by the public consortium as well, and significant resources are available to the ongoing genome annotation effort.

Celera report having access to more disk (100 TB for all operations), against 1 TB used for sequence management and analysis in the public consortium, and the total of 22 TB disk reported for the Sanger Centre [185] in January 2001. However, we have no exact numbers on what the total data storage for the public sequencing effort was, because it was a distributed project.

Celera's powerful hardware configuration, coupled with access to the mouse sequence which they were producing allowed for testing of a different approach to sequencing and gene prediction. Without hardware support, their whole-genome assembly strategy would have been much harder to implement, if not impossible.

Software

Appendix B lists the software packages used in the project. We provide a high-level evaluation from a software-engineering perspective. Celera's approach to software was industrial, with the use of state-of-the-art database and software technology, and high integration of software systems. This contrasts with more separate pieces of software used by the public project, and lack of industrial strength database technology supporting the operation at

[17] http://www.wi.mit.edu/news/genome/factsheet.html
[18] http://www.platform.com/products/

the Sanger Centre, as shown on the web site[19]. A freely available database MySQL[20] is used instead to support the Ensembl web site presentation[21]. The Whitehead Institute, the other main contributor, reports having moved to Sybase[22] to provide a robust system for sample tracking and analyzing trends in data quality. Sanger Centre seems to have relied on ACeDB and other pieces of software which were glued together to provide similar functionality. Max-Planck Institute, which was a minor contributor, used a database for data presentation and map integration but not to track the experiments needed to create the mapping data. Those experiments were recorded in lab books and flat files, and data quality issues due to illegible handwriting or mistyping required attention during the course of the project. The costs of using more pieces of software and paper-based data management are hard to quantify, but are perhaps comparable to the cost of implementing integrated software packages based on database technologies. However, a more complex data processing environment will have a negative impact on data quality and work efficiency.

Issues

In the comparison of both sequencing projects we single out the following issues related to data and software management.

- All data on sequencing, sample preparation and analysis can be more easily reviewed and checked wherever a database management system is used. As Celera report, this led to high quality data with a small rate of errors. The high degree of automation, and ready access to a database mean that any errors and discrepancies can be traced easily. The database provides an audit and quality record and increases work efficiency.

- At Celera fewer software programs were used, and more integrated solutions were produced. This probably translated into a productivity gain for anyone using this software, due to fewer user interactions required.

- The approach to gene prediction taken by Celera, called Otto, was evidence-based (i.e based on homologous genes in other organisms, and on available partial human gene sequences) and rule-based. The criteria of gene assignment were very clearly stated and encoded in the rule system used by Otto. The multitude of gene prediction programs used by the public consortium may have been harder to manage. Overall, a simpler and clearer automated gene prediction strategy should translate into more reliable predictions. Celera also had access to more mouse sequence than the public consortium. This may have led to better gene predictions, because exons are highly conserved between the human and the mouse.

- Data access tools released by the public consortium with the first draft of the human genome were not satisfactory. It appears that this aspect was not planned well in advance. Celera prepared more comprehensively and the demonstration of data analysis tools shown at one of their presentations (Edinburgh, spring 2001) revealed a range

[19] If relational database software is used there, which we believe is the case, (one of the ex-colleagues at Max-Planck Institute was recruited to join the Oracle team at the Sanger Centre), it is not used to present data on the web, and is not mentioned on the web pages.

[20] http://www.mysql.com

[21] http://www.ensembl.org

[22] http://www.sybase.com

of interfaces to view and query the data. Celera's tools were based on a combination of Java and a database. Ensemble web site[23] and the Santa Cruz site[24] still provide access to maps via clickable images which give the user no control over data presentation, and make the task of data analysis hard. Access to Ensembl will soon be enhanced by the use of more powerful database technology, and adoption of more modern browsers, currently being developed by the Apollo project.

Celera's contribution

We summarise how the public consortium benefited from the Celera's challenge. We think that the influence of Celera's approach has already had or will have the following impact.

- It speeded up the human genome delivery, as already discussed.
- It influenced the decision by the public consortium to adopt the whole-genome sequencing approach in the construction of the public mouse genome sequence.
- It showed that the industrial approach to sequencing works on a large scale, and that sequence discovery in itself is not a research issue, and could be outsourced.
- It showed that unfinished sequence is valuable, and can lead to biological discovery, by using mouse data in human gene prediction.
- It demonstrated the importance of industrial strength database and software support and business planning for all aspects of data starting with production and finishing off with data presentation and access tools.

Several of the aspects of Celera's work are still a subject of debate. This refers for instance to their different assessment of the volume of repeat sequence in the human genome. They quote a lower repeat figure, possibly due to the fact that repeat sequence cannot be assembled using purely computational techniques. Since the mouse genome is not publicly available, it is hard to make judgements about the quality of mouse sequence data or its presentation.

The public consortium's data strategy

We now turn to the public consortium's work. We think that the data and software management and presentation issues did not get timely attention in the public project, and justify this as follows.

- It is our personal experience that reliance on ACeDB [70] for data management, at was done at the Sanger Centre[25], is not an efficient work practice. If a data management system does not offer full database facilities, including transactions, rollback, constraints, archival, and a powerful query language, all those facilities are then gradually added with considerable programming effort, or are performed manually with ample scope for error and with a high labour cost. Due to the high reliance on ACeDB during the project, and lack of planning for a large scale data presentation

[23] http://www.ensembl.org
[24] http://genome.cse.ucsc.edu/
[25] http://www.sanger.ac.uk

system, current Ensembl facilities are not satisfactory. Current presentation has now adopted MySQL for data query on the web, which is much more powerful than using ACeDB, and will soon include Java browsers for maps, which are being delivered by the Apollo project.

- The number of software tools used by the public consortium shows a very high complexity. As new genomes are being sequenced, the current system will cost more to maintain than a lower complexity software system. Other potential sequencing centres are now looking at the possibility of sequencing stretches of other genomes, and trying to find out what technologies to adopt. The technologies from the Sanger Centre do not include robust data tracking mechanisms, and are not the best candidate solution for the labs which want to start sequencing now. Installation of large-scale systems like the one at the Whitehead Institute is expensive, but could in the long time be cheaper. We believe that Celera's lesson that cheap sequencing should be done industrially in a centralised fashion and not at each centre separately, should be heeded.

Conclusion about the human genome sequencing

We conclude, that from the point of view of data management, the Celera approach is superior as it is possibly less labour intensive at the point of use and system maintenance, and less exposed to human error. However, it required significant investment, and involved an element of risk, in adopting new tools and techniques. A similar conclusion can be drawn for software complexity, and it appears that Celera is well equipped to sequence many genomes fast and reliably using the tools developed for this purpose. In the final score of efficiency and quality, Celera's organisation of work was better in our opinion. Celera is a small organisation, geographically and ideologically coherent, and united in its commercial goals. It used the available public resources efficiently to gain a competitive advantage and produced tools which delivered results reliably in a short time scale.

The strategy of the public consortium was conservative, and chose evolutionary software development. This had its strengths as well as weaknesses. By using techniques known to themselves, they minimised risk. However, by the same factor, they could not make efficiency gains possible with a new system implemented from scratch. The advantage was the extremely high sequence data quality, but a disadvantage was inadequate data access and presentation.

3.3 Research paradigms

This short illustration based on the human genome project furnishes an example of how large-scale sequencing bears on the way biological research will now be conducted. With the availability of the DNA sequence, small-scale questions of the identity of one gene can now be related to large questions about groups or classes of genes and proteins across many organisms. Other large-scale data production technologies we discuss will need similar software and hardware support and will have a similar impact on research, by allowing a shift towards large-scale data analysis. Until recently only hypothesis-driven research was possible. A problem was formulated, experiments were performed, and the subsets of data supporting the hypothesis were presented and published.

3.3.1 Hypothesis-driven research

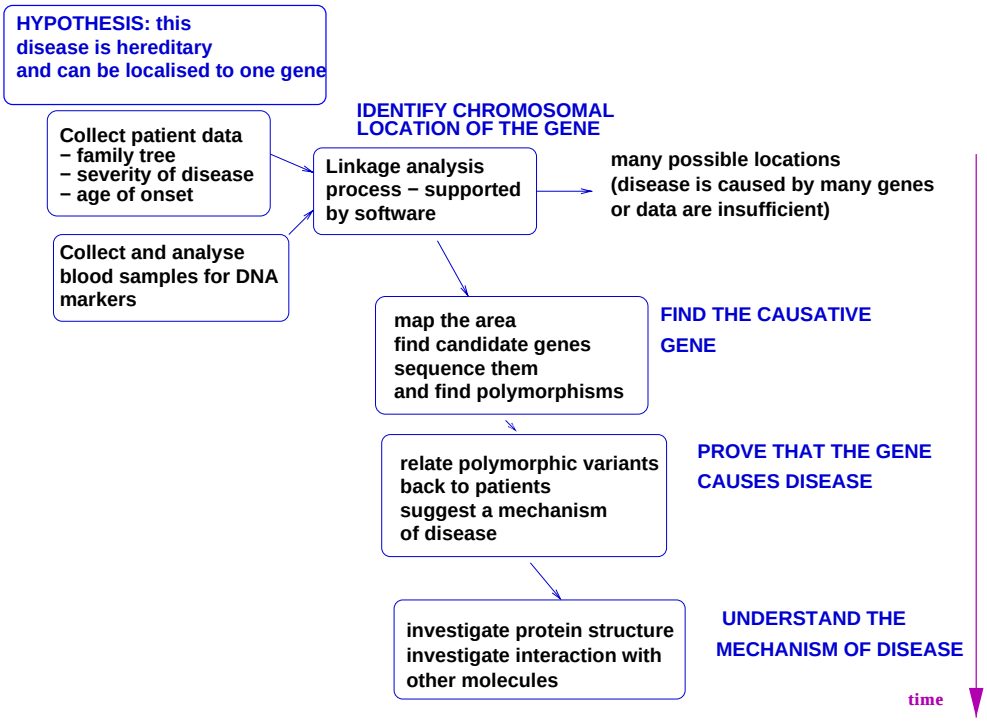


Figure 3.10: A simplified view of hypothesis-driven research in the field of disease gene finding.

A traditional approach to finding a disease gene is shown in Figure 3.10. In 1999, we conducted a case study of this process [183] and found that the improvements in underlying technology (availability of large scale gel electrophoresis used in genetic profiling of family members) have now made it possible to conduct successful gene finding within short timescales. Previously finding and cloning a gene (finding its DNA sequence) would have taken years of team effort. Now, after the publication of the human genome sequence, this process is even faster, and positional cloning[26] is simplified. It now involves examining the sequence in a particular region on a given chromosome for potential genes and mutations. In fact, during the final years of the genome project, some 30 genes were identified that way [57]. However, despite the availability of the sequence, the interface between the traditional methods of research and new technologies still presents a challenge which can only be solved using software technologies. In the manual organisation of the data flow, common in linkage analysis, shown in Figure 3.11, we see that the amount of error-prone manual data processing required to carry out the analysis is excessively large. We see this as a problem that *bioinformatics* research needs to address, and elaborate on the data flow problem in Section 3.4.3.

[26] http://www.ich.ucl.ac.uk/cmgs/posclone.htm, positional cloning: isolation of a gene knowing only its chromosomal location, which is typically identified by linkage analysis. Construct physical and genetic map of candidate region, identify the genes within the region, investigate each candidate gene until the disease gene is identified.

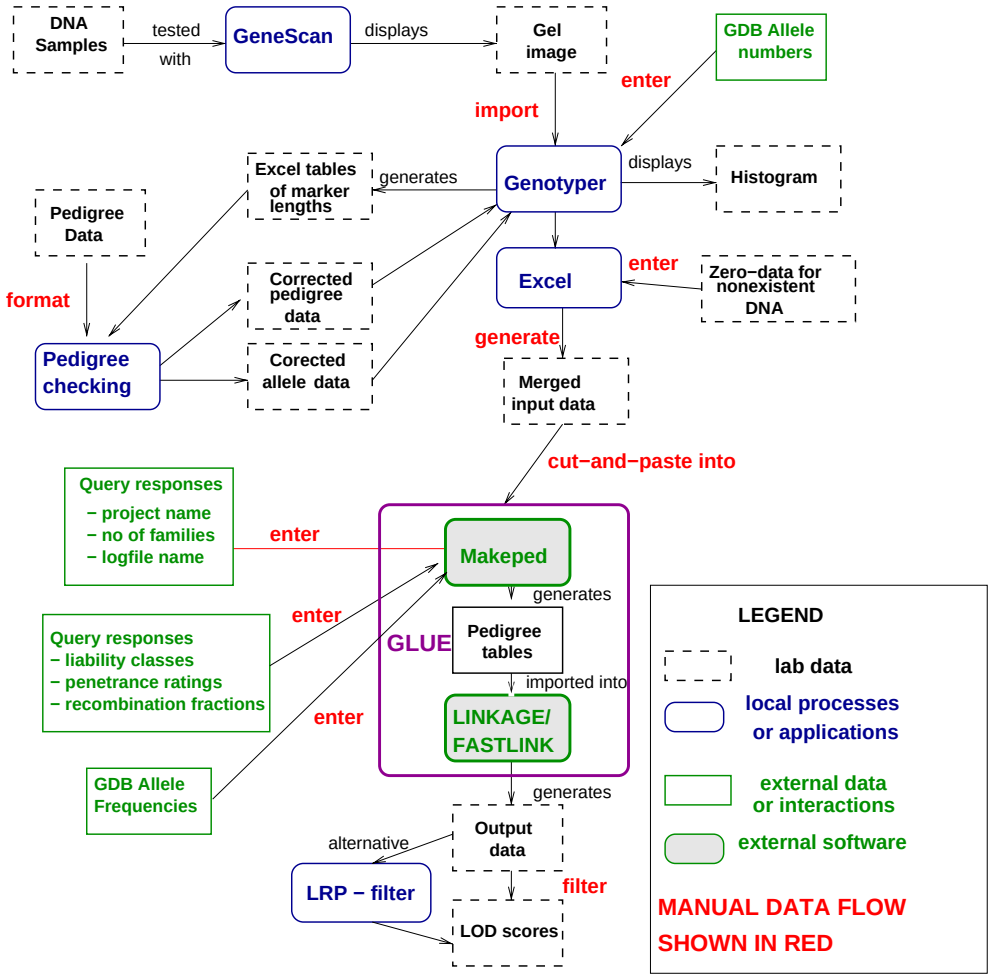


Figure 3.11: User interaction with local and distributed software typical of a linkage-analysis study using large-throughput DNA profiling [183].

3.3.2 Data-driven research

The traditional hypothesis-driven approach to biological research is now often complemented by the data-driven approach. Substantial amounts of data are collected first, and then patterns in data are analysed, and conclusions and generalisations are formulated. It is only recently that the view that all data produced in an experiment are relevant is emerging. Both negative or positive results are beginning to be shown as supporting evidence, especially with the move to make supporting evidence available on the web, as now required by most journals. For instance, in the case of PCR experiments testing the genomic position of a DNA fragment, usually only the positive results were reported, as focus on small fragments of chromosomes made accounting for all the negative results impractical. However, with the large-scale genomic approach, reporting of all PCR results or microarray results becomes imperative, and preservation of that data for future research should become the norm. Preserving and making all the original data and protocols accessible allows for data re-analysis. This re-analysis may be used in connection with improved analysis tech-

niques which might become available or might help to formulate new hypotheses, using for instance preserved data from several experiments.

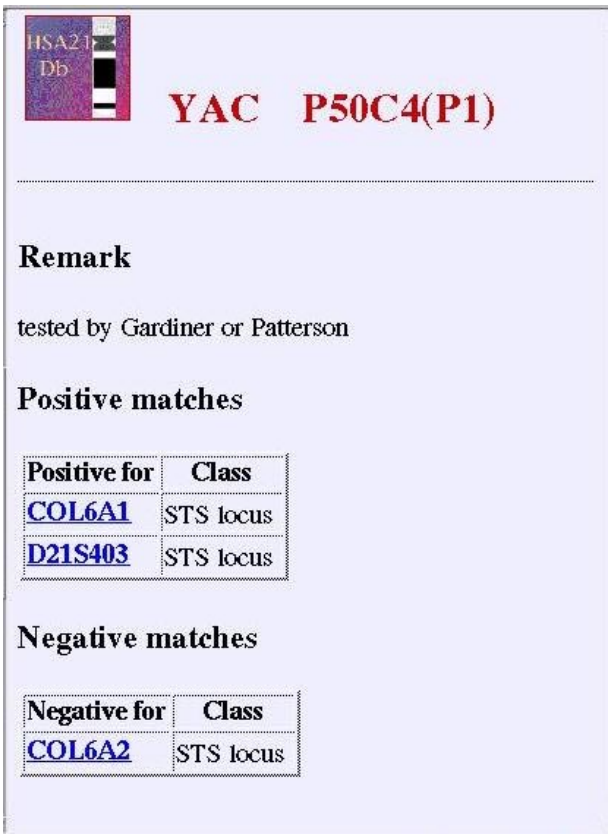


Figure 3.12: Positive and negative results from a chromosome 21 mapping paper [88] preserved in a database at http://chr21.molgen.mpg.de

An early example of using both negative and positive results is shown in Figure 3.12. The construction of one of the early maps of human chromosome 21 used negative and positive PCR results to create a map [88]. This mode of reporting, where all data are important and may be used by future research, should become the mode of operation for all large-scale experiments, where all results are relevant and may contribute to new research not envisaged by the original investigator.

A mixture of hypothesis-driven and data-driven research is shaping rapidly. Global questions relating to evolution, models of proteins, or gene interactions are emerging. Examples of such investigations include the bacterial gene classification project COGS [136, 216], or the large sequence-based comparisons between different species [181] which our research is hoping to facilitate. We expect questions which refer to large data sets to become more and more important in the near future, and see data-driven research as a challenge to the data management technology which we are trying to respond to.

3.4 The challenges of bioinformatics

New technologies and trends in biological research call for new computing approaches. We concentrate on a few possible trends in computing and *bioinformatics* research which in our opinion will follow from the data-intensive biology we described. We believe that future research areas will include:

- data management (database issues),
- data integration (via meta data),
- data flow and automation (data processing and software issues),
- data representation (visualisation), and
- data interpretation and inference (data mining).

We now explore those challenges in more detail.

3.4.1 Data management

Data management encompasses a wide range of issues. Because of our concern with the increase in data volumes, we concentrate on the need for the preservation and public release of biological results, and on the issues of indexing techniques needed for new data types. We also suggest the ultimate solution - an extended file system which organises, indexes and self-tunes.

The need for data management

The issue of data preservation and availability is of importance not only for the researchers currently producing large data sets, but also for future researchers and the society as a whole. Inadequate data management provision will lead to the repetition of costly experiments because previous results were not preserved or were not adequately described and catalogued. Current data management practices are inadequate. Large data sets are created and deposited on local disks without appropriate annotation including date, researcher's name, laboratory conditions, protocol, source data and the tools used in data evaluation. Without consulting the lab books themselves, which are paper-based and often illegible, it will be impossible to share those results with other research groups. This problem has two causes. On the one hand, funding bodies do not have a policy of enforcing the preservation of experimental data and making it widely available. Only data leading to a publication are made available, and a few years later they disappear from supporting web sites. A data preservation policy on the part of the funding councils would recognise the cost of data preservation as being lower than the cost of subsequent data re-creation, and make suitable policy decisions, for instance enforcing the purchase of database tools and an audit of database management and sharing provisions. This is currently the case only for a minority of biological projects, and good examples can be found in some of the web databases seen today. On the other hand, we recognise that biologists may not have the necessary competence in the area of data management, and they need expert advice and technical skills which should be costed in grant applications.

Maurer and colleagues [151] describe their perspective on data management in similar terms. They stress yet another point, namely that "discrepant information can be exploited to identify errors and recommend best values". Some of their examples come from physics and they stress the importance of advanced databases in the scientific progress of physical research. This argument can be extended to biological databases, and their plea for Single Nucleotide Polymorphisms (SNPs) and microarray databases is currently being met[27].

Data capture and indexing

Before specialised databases can be built, data need first to be captured at the point of origin, and saved for future use. One of the hardest problems in a biological lab is data capture with full annotation. The problem which follows is data indexing, which is indispensable if data are to be located fast. Current support for text indexing, image indexing and sequence indexing is poor. One of the currently popular annotation languages is eXtensible Markup Language (XML) [3] and techniques for XML indexing are an active research area [58]. Similarly, techniques for XML views and querying are just emerging [9, 175].

Our work focuses on new data types, in particular sequences and images. This thesis examines one of the subproblems of indexing for new data types, namely indexing of sequence data. Other future indexing needs in biology will encompass protein structures and interactions, metabolic pathways [28], mass spectroscopy data, and phylogenetic trees, just to mention a few. This list will expand to reflect new data types which are bound to accompany the appearance of new technologies and techniques. It is very likely that bioinformatics research will explore new data description and indexing schemes providing better access to such data types.

Future databases

A challenge, much harder to meet with current computing technologies, would be to make database technology available to the scientist directly. Bioinformatics databases could appear as extended file systems. Such databases would be self-describing and self-indexing, and would recognise similarities in data items stored using general meta-methods. Ideally, such databases would provide indexing, visualisation, security, and web publishing facilities at the touch of a button. They would be easy to use, and provide a range of functions, some of which can only be implemented now with a great deal of custom-built code.

3.4.2 Data integration

Data integration is an important source of software-engineering work in business organisations and a curse in biological research. Every merger of two companies is faced with the data and systems integration dilemma, and complex decisions are made as to what data should be extracted from existing production systems to the merged information management systems. In biology, the process of selection is also important, but often there are no technical means to automate the process of data gathering, filtering and reconciliation.

This section elaborates on the state of the art in biological data integration techniques, on the database foundations of such work, and on the need for ontology-based specific solutions in bioinformatics.

[27] http://snp.cshl.org/, http://www.ebi.ac.uk/microarray/

[28] http://www.ebi.ac.uk/research/pfmp/, http://www.genome.ad.jp/kegg/, http://ecocyc.doubletwist.com/

Bioinformatics data integration

With the development of relational databases and database-related bioinformatics, first attempts to solve data integration problems in this data domain appeared in mid-1990s [41]. This coincided with the appearance of web-based bioinformatics systems, and the need to merge disparate data sets to get exhaustive information. For a biologist trying to find a gene location, based on linkage analysis, the work required to assemble all of the relevant data is extremely onerous, as already shown. As changes to external data sources on the web happen daily, a biologist will never reach all data which might be relevant. Therefore, the premise of data integration for most biologists is the fact that data will often be out of date, and data acquisition will be laborious.

The labour involved in importing data sets and merging them to fit the formats required by different data manipulation tools if often significant. Simple word-processing solutions, like regular-expressions or macros, are not widely used, and they are not robust as data formats change frequently. There are no usable interfaces to Unix tools like `awk` [5] or `grep` [218] and the power of regular expressions in many word-processing packages is limited. The common solution for a biologist is to learn Perl and produce scripts which are hard to debug, and impossible to test for correctness. Current solutions are brittle and cost a lot of work.

Database research and solutions

Data integration problem is a widely recognised issue not only in bioinformatics, and has been the subject of considerable research in the database field. Some of this research relates to data translation between different formats, for instance from HTML to XML [193, 192]. Some research explores different type systems and query languages which could support data transformations [62, 41, 175] and some explores data unification problems [159, 9]. Beside the database approaches mentioned above, other approaches are also known, and they use diverse methods including information retrieval techniques [129].

The difficulty of biological data integration lies in the syntactic and semantic filtering and unification of data. Most data transformations used in current bioinformatics applications are hard-coded, and have to be re-adjusted each time one of the participating formats changes. Extensive research, based on this underlying principle includes strategies using database languages which operate over data collections, and optimise the evaluation of queries over those collections [63, 61].

Another strategy used in data integration is based on technologies which use unique database identifiers, as exploited in the SRS system [76]. This system builds tables of associations between database items in different underlying databases, and provides queries over a large number of databases. The user selects the databases to query and query conditions, and pages of clickable links to data found by the system are presented. No data unification is possible, and multiple links have to be followed to retrieve the data. This system is used only by expert users.

Future solutions in the database area may be based on meta-language descriptions of data, a mechanism which may allow for semantic data unification. By separating content and presentation, as done in XML [3] and its predecessor the Abstract Syntax Notation [1], data translation and unification can be performed on a higher level. In particular, current research into ontologies for biology [16], now reflected in several databases including the

Mouse Database[29] and the yeast database SGD[30], is very promising.

A meta-ontology

The approach of using ontologies to describe the semantics of data could be extended to encompass the entire universe of biological data processing. Future ontologies could describe data sources, applications, type of data flow, data transformations and even users and their profiles. Possibly all web resources relevant to bioinformatics research could be associated and unified using meta-data. Biologists will have to be involved in developing and reconciling different views of data, while the computing science researchers will concentrate on efficient mechanisms of storage and query using ontologies and ontologically described data. The possible ways of storing and deploying ontologies are a subject of current research [31, 46] and performance optimisations will provide ample scope for future work.

3.4.3 Data flow and automation

Data flows are a conceptual tool used in business application design [229]. User interactions with data are captured and encoded using this technique, and applications are built to automate the flow of data. This solution assumes that data flows are static, and system re-engineering is required only if changes in business practices occur.

In biology data flows are very complex, and change frequently. They span the internet and focus on one user or a small group of users. It appears that traditional data flow solutions would be too expensive to solve the data flow problem in this domain.

We examine two sub-problems in this area. The first one is concerned with individual user interaction with web interfaces, and the second one concerns the flow of data between applications.

Using web interfaces

Current ways of interacting with biological databases on the web are constrained by the stateless HTTP protocol[31]. Most of the web interfaces used in biology are built as web forms which use CGI [152]. Data are retrieved by a combination of queries and link traversals. This approach is appropriate for the analysis of small amounts of data, but becomes inadequate if more data need to be retrieved. Web sites are designed to provide this direct interaction with data and not to provide bulk delivery of data sets. It is not realistic to expect that all web data providers create varied data access modalities. This problem was identified in our case study of linkage analysis [183], and we see a possible solution in providing agent programs for the users who need to access data in bulk. Such programs could be relatively simple to start with, with simple means of defining the database to query and the items to be retrieved, and later developed into sophisticated tools which remember previous retrievals and allow for building complex retrieval and filing scenarios.

[29] http://www.informatics.jax.org/
[30] http://genome-www.stanford.edu/Saccharomyces/
[31] http://www.w3.org/Protocols/

Application data flow

The flow of data from external sources into applications and between applications is an important research area which we identified. This problem is closely related to the business data-flow problem. The main difference comes from the high variability of biological data flows, and the high rate of change in underlying data and software components. In this context flexible mechanisms are needed. We believe that future research in this area will strive for solutions which are user-focused and not application focused, and will use techniques which allow for tracking and auditing of the data flow.

Future trends

Future solutions to the data flow problem will probably employ the following techniques:

- graphical languages for workflow composition and evolution that could be used directly by a biologist,
- meta-data for data and software description and data flow capture, including usable interfaces to meta-data (editors and viewers), using appropriate visual metaphors, as current viewers support only hierarchical data representation, see http://www.geneontology.org,
- theoretical foundations of data transformations to be applied to the evolution of meta-data descriptions, including typing for data, software and flows,
- safe implementations of internet data flow mechanisms,
- software architectures to support evolvable data flow applications.

Some of the issues in this area have already been investigated by Baker and colleagues [31], but current solutions to the problem, including Pise [141] and AppLab [200] assume that the data flow will be engineered by a computing scientist, and therefore will not be easy to change by a biologist. Newer approaches to this problem take into account the need for evolvable data flows, and the trends can be seen in the work by Hull and colleagues [111] and the Vortex and E-services projects at Bell Labs[32].

We envisage future systems using standardised descriptions of data, software and flows, and graphical toolsets which would allow the biologist to compose and generate data flows as required. In a shared environment such data flows would be stored, could be re-enacted and modified, or re-executed automatically.

3.4.4 Visualisation and representation of data relationships

Visualisation can be defined as an appropriate rendering of multi-dimensional relationships using two-dimensional presentation media like the screen or paper.

Genome maps

In one of the common activities — drawing a genome map, see Figure 3.13, we simplify the 3D structure of DNA and represent it as a long line, with distances between points on the line corresponding to physical DNA lengths or other scales. This map representation has

[32] http://www-db.research.bell-labs.com/user/hull /e-services-interoperation/index.html

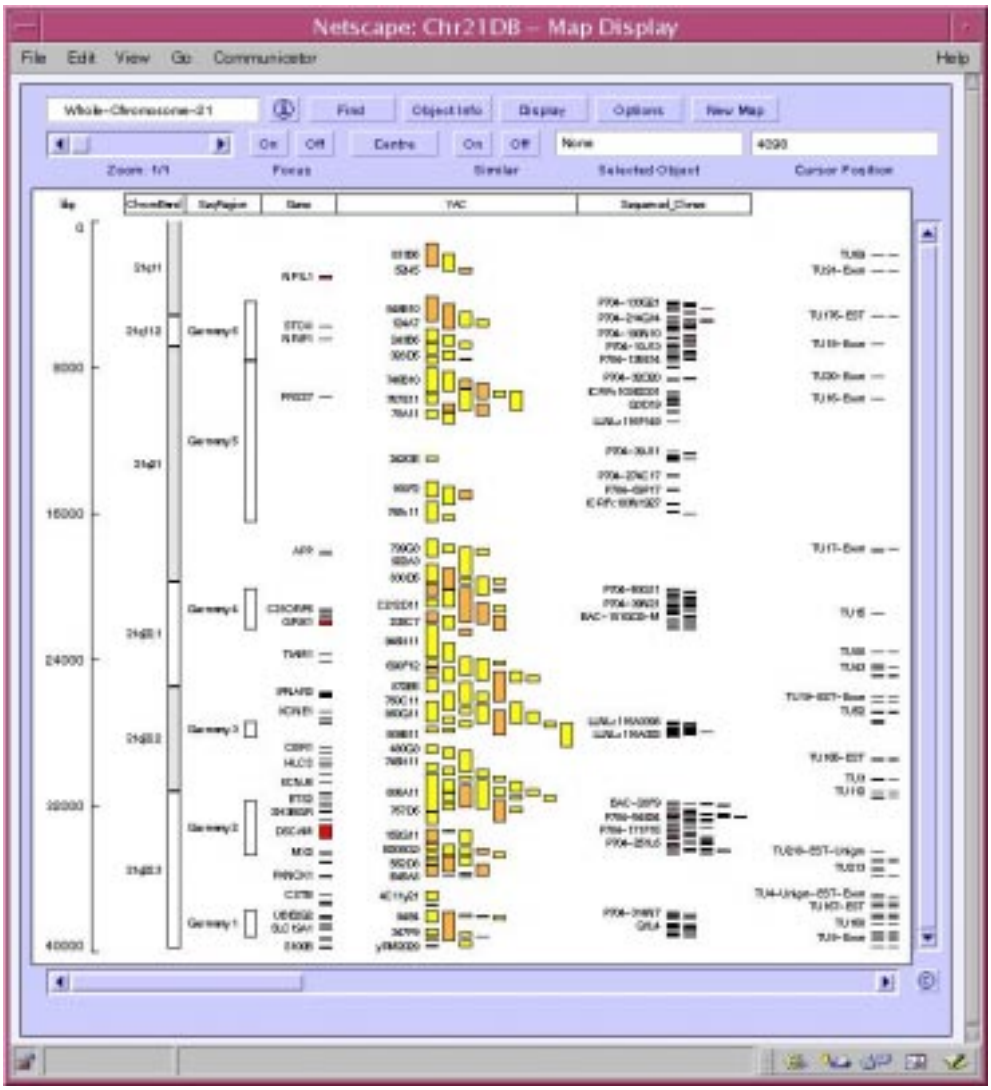


Figure 3.13: A visualisation of human chromosome 21 sequence data at http://chr21.molgen.mpg.de

taken some time to develop, and in the plethora of "genome browsers"[33] only a few are easy to use. Implementing this basic visualisation technique in a computer program which has to be fast, flexible and capable of representing genetic objects at different scales of resolution is a challenge which is still being addressed. The usability of mapping tools in the area of genome representation and annotation has not been investigated, despite the existence of several such systems. We believe that further work is required to find out what solutions provide good access to this type of data. Further to that, a new type of browser is now needed, one that can show comparisons between multiple genomes. Our short examination of current answers to the problem of inter-genome comparison as provided by ACeDB[34]

[33] http://compbio.ornl.gov/channel/index.html, http://www.ncbi.nlm.nih.gov/cgi-bin/Entrez/map_search?chr-=hum_chr.inf&query, http://genome.ucsc.edu/goldenPath/hgTracks.html

[34] http://www.wormbase.org/

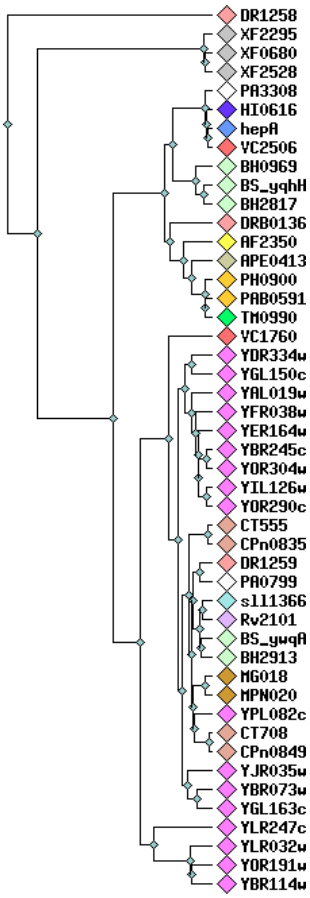


Figure 3.14: Cluster of 47 orthologous proteins COG0553 from several bacterial species as shown at http://www.ncbi.nlm.nih.gov/cgi-bin/COG/palox?COG0553

and ACT[35] showed that they are inadequate.

Hierarchies and taxonomies

Another area of great importance is the representation of biological hierarchies and taxonomies. Both of these can be represented abstractly as graph structures, but it is not obvious what representations and visualisations are appropriate once the size of the problem is greater than 10 discrete items. It is also not clear what underlying database structures serve graph models best. This problem is currently showing in the difficulties of searching and navigating hierarchical file directories, and is not resolved for the case of viewing two hierarchies at the same time (participation in two hierarchies is not allowed in a file system, but quite common in taxonomy [182]). The difficulties of scaling up data displays are apparent in the representation of protein families [216], reproduced in Figure 3.14, where the tree display shows a cluster of 47 orthologous proteins from several bacterial species. Beside being hard to read, such representations do not provide an optimal view of the complex relationships within this cluster.

[35] http://www.sanger.ac.uk/Software/ACT/

Metabolic pathways

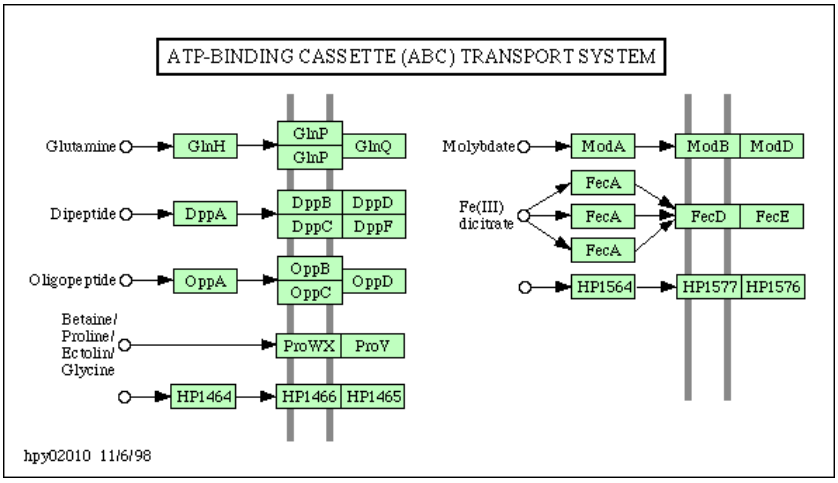


Figure 3.15: A metabolic pathway as shown at the Kyoto Encyclopedia of Genes at http://www.genome.ad.jp/dbget-bin/show_pathway?hpy02010

Another biological example comes from the display of metabolic pathways, see Figure 3.15. Metabolic pathways model biological interactions which should be viewable at different levels of resolution. Current tools produce images (usually GIF format) which have embedded clickable links, enabling drill-down to the database entries. This technology is limited to displaying objects at a pre-set scale and does not allow any client-side, user-driven exploration or adjustment of the image data (scrolling, zooming, colour modification, selection of a subset of data for analysis). Better display technologies are needed to provide access to such data as it is produced by the various microarray and protein interaction experiments described in this chapter.

3.4.5 Data interpretation and inference

Dealing with large quantities of data is a challenge and new data mining techniques are being developed with the view to simplifying the analysis of large data sets. Data mining techniques are already used in the context of sequence analysis [37], protein classification [74] and micro array data analysis [72]. Data reduction techniques are now used in microarray analysis packages from Affymetrix, www.affymetrix.com, which come equipped with data mining support from Spotfire, www.spotfire.com. Similar techniques will probably soon appear for other types of data. The challenge in this area will be to find approaches which can combine different sources of data in analyses which consider many factors. This direction has not been explored sufficiently yet, and possibly different statistical approaches will be applicable. It appears that both the industry and academic researchers are working very actively, and the laborious analysis of large data sets will soon be performed using statistical and visualisation packages running on top of powerful databases. For microarrays alone, PubMed database, http://www4.ncbi.nlm.nih.gov/PubMed/, lists in excess of 1,000 citations, and the strength of existing statistical techniques seems to provide adequate foundations for the required inference engines.

Use of information retrieval tools in this area has already been mentioned, and data mining of publications [129, 146, 137] is now becoming common. Also systems which combine text mining of publications and database descriptions to automatically classify proteins are being investigated [112].

3.5 Summary

We now close this introductory chapter which argues that new computing technologies and data storage standards are needed to manage the flood of biological data created by new large-scale experiments. We have provided examples of new data types, and an illustration of large-scale data processing issues encountered in sequencing the human genome. We discussed the new trend in research which we term “data-driven research”, and outlined future directions in bioinformatics which may contribute computing solutions needed in this area. Our thesis addresses just one possible technology which could speed up sequence data searching. Before we describe our contribution in this area of research, we devote Chapter 4 to the introduction of theoretical concepts on which our work is based.

Chapter 4

Theoretical foundations

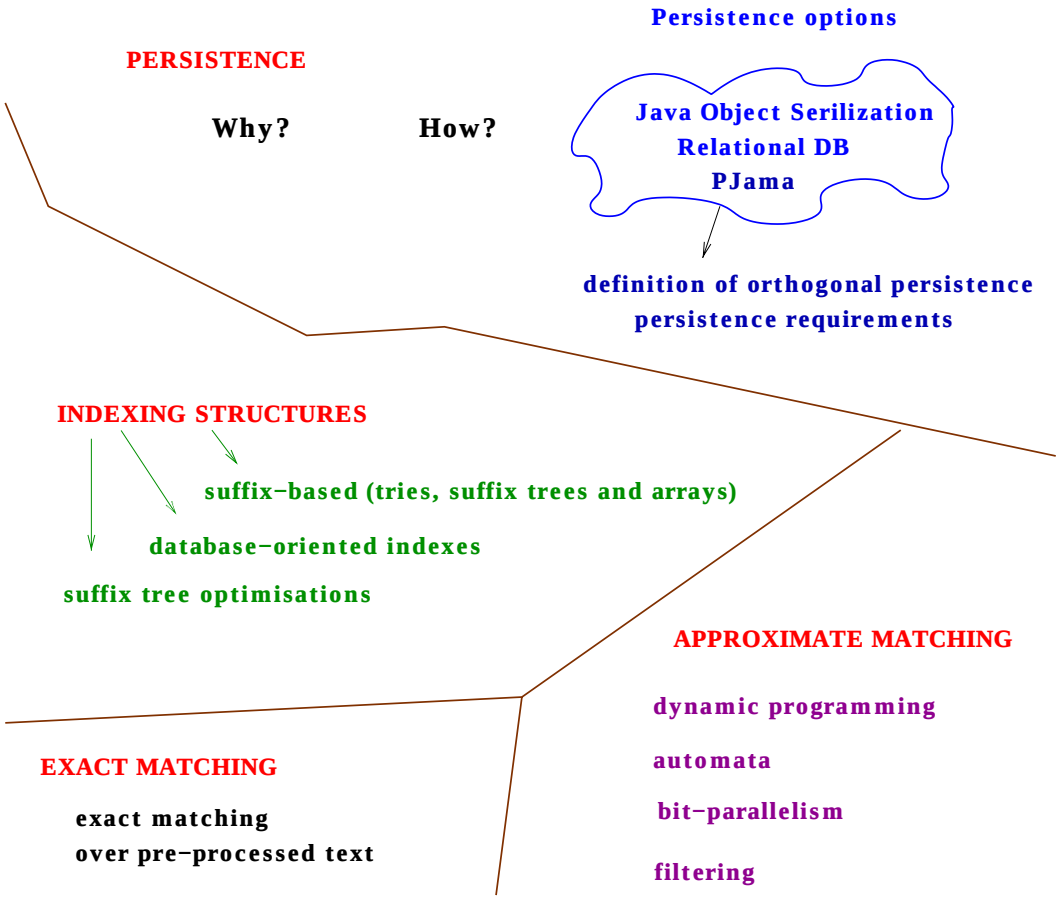


Figure 4.1: Overview of Chapter 3.

Our work on indexes to biological sequence is based on two underlying requirements. One is for speed of access to sequence and the other one is for programmer's convenience in integrating sequence searching into complex bioinformatics applications. This requires making choices among the technologies, data structures, and algorithms to be explored. Our path leads from the traditional in-memory searching as done by most sequence-searching tools to

secondary-memory searching. Secondary storage aspect requires persistence, and the first section of this chapter will describe the theory and practice of persistent applications. A data structure (index) which can speed up searches in secondary memory is also required, and Section 2 will look at potentially relevant indexing structures. And finally, exact and approximate matching algorithms are needed. We review some of the standard methods in Sections 3 and 4. More detail about approximate matching in biology will be presented in Chapter 5. A graphical overview of this chapter is presented in Figure 4.1.

4.1 Theoretical foundations of persistence

Figure 4.2: Disintegration of the Persistence of Memory by S. Dali, http://www.salvadordalimuseum.org/

We explore techniques of potential use in building persistent indexes supporting fast searching on biological sequence data. To demonstrate the feasibility of our approach, we want to use an implementation of persistence which can support any data structures encoded in Java which we had chosen for ease of integration with existing data viewers and data translation software. Persistence can be loosely defined as the longevity of computations (which implies longevity of associated data structures). This is normally achieved by encoding the computations (and data structures) in a storage medium which does not degenerate with time. In the world of databases, the storage medium remains intact, even in the event of a power cut or system failure, and the effect of the past computation which created an index to data remains available to future computations.

4.1.1 Motivation for persistence

The main reason to preserve computation state is economic. Economy means minimising the computations required, and minimising the human effort needed to design and execute the computations (programming effort needed for software creation, and effort needed to initiate the program execution). There is also a safety and reliability dimension. Human beings are prone to error. Complex programming paradigms and complex interfaces are more error-prone than simple ones, and error may be unsafe or detrimental to business or science. If computational state is preserved, on program re-entry direct reference to the past state is made, and complex initialisation rituals need no longer be performed. In the case of suffix trees for the human genome, building a memory-resident suffix tree takes at least 9 hours and 45 GB of RAM, and building and writing one to disk would take 140 hours (extrapolating from our experiments which required 14 hours for 10% of the human genome) and use only 2GB RAM. If we intend to use a tree many times, and do not have a

machine with 45 GB RAM, it is advantageous to preserve it on disk[1].

The other economic factor is programming labour. Using tools which simplify and automate data storage is a way of producing applications more cheaply. Managing data and computations using just one programming language which automatically stores data and computation state is cheaper than having to support two programming languages (a procedural language and a data definition and access language) which are not compatible in their view of data, as one is record-based and the other one set-based. Two views of data, as paramount in JDBC [233], for instance, require more intellectual effort, and mastery of two programming languages, instead of one. Current training for computing scientists includes the teaching of both relational (functional) and procedural languages, so that persistent applications can be built using a combination of at least two technologies: a database language and a procedural language. In the case of using Java Serialization, only Java is needed, but this mechanism is suitable only for small applications and not for use in multi-user environments needed for instance for e-commerce.

Finally, the third economic consideration is the need to produce correct software. Code which deals with two programming paradigms in one system is bound to be more likely to have errors and will take longer to debug than code produced using software which abstracts over persistence. Errors are detrimental not only economically, but they may have impact on human lives. Therefore using simpler (transparent) arrangements for data and application storage, may impact the quality of life, even if the initial impact is just the perceived economy of code production.

4.1.2 Available persistence mechanisms

Java Object Serialization^{TM}

Java Object Serialization [93] defines ways of serialising objects which are to be written to disk. In *Java Serialization* the whole object-graph has to be present in memory while being serialised or de-serialised, and additional data structures may be needed to cope with cycles in a data structure being serialised. This imposes a size restriction and limits the index to the size of RAM. This restriction is unacceptable in the database context. Random access to parts of the object graph is needed, and this is not supported. A work-around using this technology would involve partitioning the structure into smaller units‘ and possibly using a relational database to store parts of the tree. In an object-relational mapping relations could store subtrees, for instance indexed on a common prefix.

A custom-built technique of encoding the tree to disk, not involving *Java Serialization*, would require defining a storage format, possibly partitioning the structure, and using random access disk reads (equivalent to seek function based on a disk address). This solution would require new code for each data structure under investigation. We did not investigate that path, as it would seriously impede our progress.

Relational databases

Relational databases have two mechanisms which are of potential use for tree storage. We look here at the facilities available in the ORACLE database [144]. We could use relations or Large Objects (LOBs) to represent a tree.

[1] We expect to reduce the time required for tree building.

We could use three tables, one for internal nodes, one for leaves, and one for links from parent to child. Each node could be uniquely identified by the first node character and by an index of that character. Link traversal would be by views joining the tables, some of which might be materialised. This solution would probably have a high storage overhead, and graph traversal would be prohibitively expensive.

Another available mechanism is the use of Large Objects (LOBs), where each LOB may contain up to 4 GB of data [75]. LOBs can be manipulated using either a PL/SQL package called DBMS_LOB or via an API called the Oracle Call Interface (OCI).

Recently we became aware of yet another alternative, called Oracle 8i Extensibility Framework [211, 207, 13] which seem to be promising and worth exploring.

There are two problems which might have to be faced if ORACLE were to be the persistence platform. The main issue would be the labour required to create the mappings between a programming language we would be using, and ORACLE tables or LOBs. A considerable amount of programming would be spent on code which does not directly bear on our research issue, and we would have no guarantee of good performance. The guidelines for the use of LOBs are such that extreme care is required, so that performance does not suffer. On the other hand, we think that using a table representation of our tree would disable the use of optimisations built into this database. SQL is not capable of expressing recursive queries of unknown recursion depth, so that any searching would have to use procedural constructs. The same performance penalty would be incurred if we used LOBs. We did not explore this avenue because of the high labour cost and high perceived risk.

Other databases

Our previous experience of SHORE [45] under the guise of Predator [201] was disappointing. We experienced memory leaks and subsequent database corruption. Those may have been due to Predator incorrectly assessing SHORE objects. This meant that the use of SHORE might give us trouble as well, because the complexities involved in creating object definitions, compiling them, and then filling up with procedural code may have consumed considerable resources. We also had reports from our colleagues which pointed to inefficiencies of several known object-oriented database systems, mainly in the areas of performance and scalability. We wanted to minimise time spent on software development to free time for indexing research, and we took the risk of using the persistence paradigm rather than taking the risk of spending additional time on coding. We now explain the theory behind PJama.

4.1.3 PJama and orthogonal persistence

Orthogonal persistence envisages a unified view over all computations and data. This vision which has guided the development of successive persistent language implementations of which PJama is a recent embodiment. The traceable roots of persistent philosophy reach back to Pascal/R in 1977 [194], and are then developed in early 1980's [19, 18] as an implementation of a persistent language PS-algol. Successive stages and experiments with persistence include Napier88 [40] and PM3 [107].

The philosophy of persistence is a reaction against the strong dividing line usually drawn between persistent data and volatile memory structures accessible during program execution. Persistent languages offer to cure this dichotomy, and propose a unified view pre-

serving data and computation between successive program invocations. Such view is partly implemented in databases, but this implementation is imperfect. Database implementations store data, or the results of computation, only if such storage is explicitly requested. They store data which can be typed into the few available categories, and data have to be accessed using a limited repertoire of commands, for instance SQL [52], PL/SQL or embedded SQL for relational databases, or OQL [47] for object creation and retrieval, and C++ or Java for data update, in object-oriented databases [47]. The database paradigm of data persistence has proven essential to the growth of economy but falls short of the ease of use, and of the completeness of persistence, which could encapsulate the entire computation.

Defining orthogonal persistence

Orthogonal persistence is based on the belief that a programmer should never have to write code to move or convert data for long or short-term storage. The main tenets can be found for instance in the FIDE book [24], and an up-to-date overview of achievements is available in [23]. We restate the principles of orthogonal persistence.

- The principle of *persistence independence*. The form of a program is independent of the longevity of the data.
- The principle of *data type orthogonality*. All data objects and computations should be allowed persistence independent of type.
- The principle of *persistence by reachability*. Language reachability criteria are used to define the extent of objects made persistent.

PJama uses the typed language Java [93] to implement orthogonal persistence by reachability. By designating a root or roots of persistence (a designated class or classes), we achieve persistence of all objects reachable from the roots. In PJama persistent classes are defined as static and added to the set of persistent roots. Adding persistence to a complex Java application involves adding a few lines of code in the class or classes which have the `main` method for a given application. In the case of our tree indexes, we make the main tree class static, and add checkpoints - which correspond to database commits, to write the tree structure to disk when the tree grows by a certain factor[2]. The following lines of code are required;

```
import org.opj.*;
static  OPRuntime.roots.add(TreeClass.class);
static TreeClass self;
OPRuntime.checkpoint();
```

We also define the tree root and the array of symbols to be indexed as static variables which amounts to 6 lines of code altogether.

[2]PJama currently cannot detect that the available memory is full and checkpoints have to be performed explicitly for objects which exceed RAM size.

Persistence requirements

To make data-persistence automatic (as is now the case with automatic garbage collection), and useful to software developers, several requirements have to be fulfilled. We summarise those, closely following [23]:

- Orthogonality. Any object must be capable of persistence.
- Independence. The language must be unchanged and conform to a standard, here the Java Language Specification (JLS) [93]. This is a practical consideration which ensures that compiled software can run equally well in a persistence-capable environment as it does in a transient setting.
- Durability. Persistent system has to be able to recover from crashes without corrupting previously committed computations. This is equivalent with the trust that we put in database software.
- Scalability. Computations of any size should be supported.
- Evolution. Application should be able to evolve in a controlled manner (equivalent to database restructuring techniques).
- Migration. Mechanisms enabling migration to new platforms (hardware, OS, language implementation) will be provided.
- Endurance. Continuous operation with no downtime is required.
- Openness. Interaction with the external world, independent of the type of connectivity and data is needed.
- Transactions are needed to capture the semantics of complete isolated operations which are concurrent and may fail.
- Performance must be comparable with other ways of implementing persistence.

Those persistence requirements give rise to several technological challenges, some of which are still being solved. PJama [21, 180] addresses most of those requirements, but each of them is satisfied to a different extent. The main contribution from the software engineering point of view, however, is in showing that the dichotomy of persistent and transient computation can be overcome with minimum programming effort on the part of a software engineer. The impact of such a system is very significant in research and prototyping where several alternative solutions have to be evaluated, and their performance compared. In this respect, for object-oriented data modelling, PJama is functionally equivalent to the use of a standard database product for fast prototyping, but much easier to use because of one-language environment with data modelling capabilities limited only by Java language constructs.

In theory, producing a software system implementing fully orthogonal persistence should have a dramatic influence on software engineer's productivity. However, the acceptance of orthogonal persistence is not high. This is due to several factors. One of them is ignorance and inertia, because the heavy indoctrination by database vendors, makes thinking about applications which do not have a separate database language unthinkable. Another is lack

of experimental results showing that it is cheaper to construct applications using automated persistence than using databases. It would be hard to construct and finance a valid experiment to prove that orthogonal persistence delivers business solutions faster, see Atkinson [20] for a similar argument. A valid experiment would have to include 2 teams producing and maintaining a complex application using 2 alternative technologies over a long period of technological change. To make such an experiment viable, the underlying platform would have to fulfill all of the requirements listed above. And at the end of the project, evaluating software that was designed 15 years earlier, and tested over 10 years would also be hard. In the meantime, other technologies would be developing faster than the platform under testing. It seems that the only way of influencing the database market of the value of persistence will come from some other research direction where transparent storage of some type of data will be needed, and indeed, some forms of automatic persistence may appear for XML data [3].

The benefits of persistence

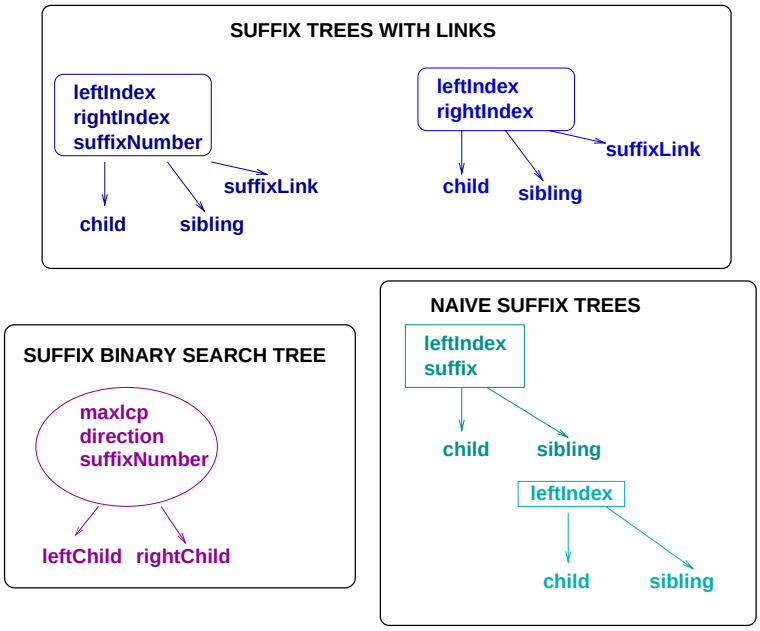


Figure 4.3: Five different indexing structures investigated.

Persistence played a very significant role in this research. Our aim was to develop and compare different string indexing structures, without having to produce hand-crafted disk representations of all of the structures we investigated. PJama enabled this research by providing zero-programming-cost transparent storage for large data sets. During the course of this research we investigated five alternative data structures, shown in Figure 4.3. A performance comparison of those data structures was facilitated by the fact that data storage happened with minimum programming intervention. Due to the simplicity of the persistent programming model, we could easily compare in-memory performance to on-disk performance of each data structure.

Not all of the features of PJama were equally relevant to our work. One of the main issues we faced was scalability, and this requirement necessitated some improvements to the PJama platform [155]. We were also concerned with durability, particularly with store recovery after a crash caused by multiple thread interactions. As we needed 14 hours to create a store, fast recovery was essential as we wanted to carry on further tests without having to recreate a store. Endurance was also indispensable, as it supported our long running store creation. Finally, store performance was important, so that our results would be competitive with approaches based on in-memory string comparison techniques. Our investigation did not use data evolution or openness, but we know that those would be necessary in an operational system. Features like orthogonality and independence were most important of all, as they saved time and enabled code reuse between the transient and persistent contexts.

We currently return to other theoretical considerations underpinning our research, and discuss the data structures which we considered while looking for a database index structure to use with DNA and protein sequences.

4.2 Data structures for string indexing

Indexing of DNA differs from text indexing. We presently do not understand what the structure of DNA sequence is, and cannot divide it into words which could be indexed efficiently using one of the known inverted file structures. We are dealing with long strings of length up to 263 Mbp (human chromosome 1, the longest human chromosome), and with the requirement to search for exact and, more importantly, approximate matches, where overall sequence similarity may be as low as 20 per cent. Additionally, biologists want to find common structures in genomes, encompassing all human genes, or all human, mouse, and rat genes, or DNA from a group of related plants. Indexing structures serving this purpose must be able to cope with large amounts of fairly static data, for instance indexes over two or three mammalian genomes totalling around 9 billion letters. At the moment the human and mouse sequence datasets are being updated frequently, but in two years time the genomes will be well known, and the challenge will be to compare them with other genomes which will be sequenced then. This leads to the requirement for both static and dynamic indexing structures. In this section we outline most commonly used text indexing structures, and discuss their appropriateness for DNA sequence indexing, taking into account the fact that DNA cannot be easily broken into words.

There are two main approaches to indexing text data with potential application to biological sequence. One is based on indexing suffixes. Formally expressed, for a string $S = s_1 s_2 .. s_n$ of length n, a suffix index indexes some or all strings $s_j s_{j+1} .. s_n$ for $j's$ between 1 and n. The other approach is based on indexing all words of a given length, or up to a given length. Given k, some or all substrings $s_j s_{j+1} .. s_{j+k-1}$, of maximum length k, are indexed. In biological sequence searching in memory both approaches are used. In our opinion the use of suffix indexes is possibly preferable. The reason for that is that if a similarity between two strings is found, biologists want to see similar sequences aligned in the areas of homology. If only very short substrings are indexed, building an alignment spanning over 100 000 bases (if a human and a mouse gene are being compared) or 5 million bases for

two bacterial species will have a high computational complexity[3][4]. Moreover, it is widely accepted that approaches based on hashing (q-grams) are only appropriate for low error levels (high string similarity) [168, 39]. Both problems can be overcome by using suffix trees. Recent results on the use of suffix trees in biology include:

- yeast gene promoters analysis (6000 gene promoter sequences of 600 bases each) [37] which used a fraction of the yeast genome and employed a suffix tree structure to list repeating motifs,
- *Helicobacter pylori* genome analysis with 1,667,876 bases of DNA [225],
- suffix tree for repeat analysis, REPuter [139]. REPuter can index both strands of a genome of up to 67 Mbp sequence, or a single strand of up to 134,217,727 bases ($2^{27} - 1$),
- the comparison of mouse and human genomes [226], using MUMmer [66]. In this application all human and mouse genes were translated into protein sequences, and repetitive sequence motifs were read off the suffix tree. The tree was held in memory, and the size of the indexed protein strings must have been in the region of 20 Mb (the total of gene coding DNA for human is around 30 Mbp, translation to protein will reduce that to 10 Mb, and adding the same again for the mouse makes 20 Mb).

We note that in the case of whole genome sequence alignment, BLAST [7] cannot be used, as testified by [66], and BLAST2 [217] or PipMaker [199] are not powerful enough either. The most practical alternative is the use of MUMmer (provided enough RAM is available) or an all-against-all BLAST comparison [57, 145] with 2-way BLAST analysis. Such a comparison of the human and mouse genomes will require a lot of computation. If we assume 40,000 human genes, and a similar number of mouse genes, an all-against all analysis in both directions (because BLAST is not symmetrical) will need

$$2 \left(\begin{array}{c} 80000 \\ 2 \end{array} \right) \approx 6.4 * 10^9$$

gene alignments, which slightly exceeds the sum of the lengths of both genomes. Taking an average gene of 1500 bp of DNA, we need to fill in $6.4 * 10^9$ square matrices of size $1500 * 1500$, i.e. perform a matrix calculation for

$$14.4 * 10^{15}$$

cells. In comparison, building a suffix tree for the combined mouse and human genomes requires in the worst case

$$(3.2 * 10^9)^2 \approx 10.24 * 10^{18}$$

character comparisons, but in the average case will only need

$$3.2 * 10^9 * \log(3.2 * 10^9) \approx 7 * 10^{10}$$

character comparisons, and one tree traversal. If we could build a tree for 6 Gbp of DNA, we might in the future be able to perform sequence comparisons by traversing a suffix tree, instead of performing all-against-all BLAST. Another scenario could use a tree for two or three mammalian genomes to perform an "indexed BLAST" against other organisms faster.

[3] See also Appendix B, and [102]

[4] We use the widely accepted notation of Kbp meaning 1000 letters of DNA code, kb meaning 1000 letters of protein code, and Mbp and Mb standing for millions of letters

4.2.1 Suffix based indexes

In this section we discuss the suffix tree *ST* which is a version of a digital trie [92], the suffix binary search tree *SBST* [125, 126] which can be viewed as a tree equivalent of a suffix array [148], and the suffix array itself. Further sections describe disk resident data structures including q-grams, a prefix index, different implementations of suffix indexes, and finally, suffix tree storage optimisations.

Some of the indexing structures we present here are derived from tries. Tries are recursive structures which use the characters, or digital decomposition of the key, to direct the branching. The name trie comes from the word re*trie*val. Different versions of the trie structure are known. Binary tries branch like binary trees with always 2 children at a node. For digital tries, the size of the alphabet used dictates the maximum number of children per node, and each child has a different starting character. Suffix trees are compacted digital tries. They can be built in $O(n)$ time, and searched in time proportional to the sum of the length of the query and the number of query occurrences. *SBSTs*, on the other hand, branch in a binary fashion, which enables tree re-balancing and may guarantee good searching behaviour, by minimising the average length of the search path. For randomly constructed *SBSTs* building times are $O(n \log n)$ and query times are proportional to the sum of the tree height h, length of query m, and the number of pattern occurrences in the tree, x, i.e. $O(h + m + x)$. The suffix array has similar building and searching characteristics as the SBST, but more compact storage. Which of those structures are better when used on disk, and what kind of sequence searching they are best suited for [168] is still to be investigated.

Suffix tree

A suffix tree indexes all or some of the suffixes of a string. It is a compressed version of a digital trie. An example trie indexing all suffixes of **ACATCTTA** is shown in Figure 4.4. A digital trie has $O(n^2)$ nodes, and at each node each child starts with a different symbol, so that the number of children is limited by the alphabet size. A trie can be compressed to form a suffix tree. In the compression process single nodes which have only one child are merged with this child, recursively, and annotated with the length of the string they index. A tree constructed in this manner for **ACATCTTA** is shown in Figure 4.5. It is often referred to as PATRICIA tree (which stands for "practical algorithm to retrieve information coded in alphanumeric") [161, 135]. The process of compression is illustrated in Figure 4.6. This edge contraction is the underlying feature of a suffix tree. To change a Patricia tree into a suffix tree we need one more modification which ensures that each suffix is represented by a leaf. In Figure 4.5 an inner node represents suffix number 8. To ensure a one-to-one relationship between leaves and suffixes, we add a terminator symbol to the end of the string.
A suffix tree with a terminator symbol, showing path traced in searching for character **T** is shown in Figure 4.7.

Exact searching in a suffix tree involves tracing the query string from the root down the branches. If a match is found, all nodes below the match are reported as matching the query (a traversal of all leaves below the matching node has to be made). Suffix trees have been widely studied, but mostly from the point of view of main-memory performance of $O(n)$ algorithms. A very thorough overview of the suffix tree and other data structures used in string searching can be found in Gusfield's compendium of string searching algorithms [99]. Suffix trees are characterised by fast construction, and good search performance. A

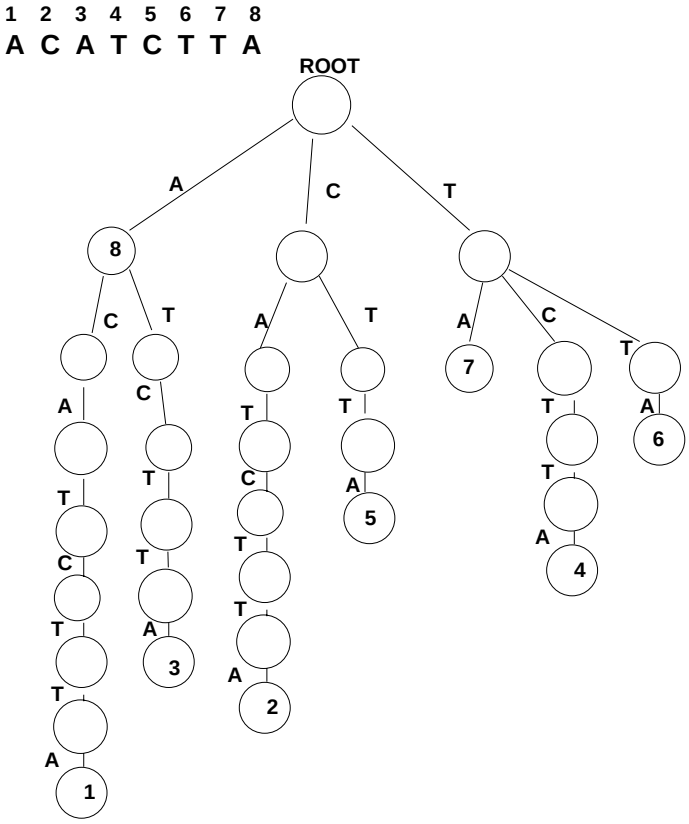


Figure 4.4: Trie indexing **ACATCTTA**.

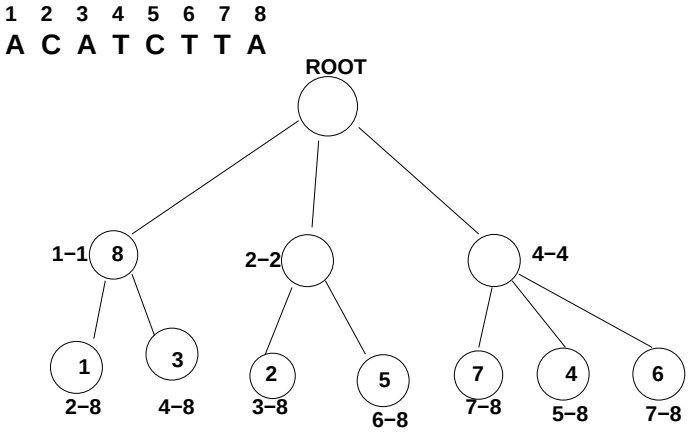


Figure 4.5: Suffix tree.

suffix tree for a string S of length n (for small alphabets) can built in $O(n)$ time, and exact search for a pattern of length k has the cost of $O(k + m)$ where m is the number of times the pattern is present in S. Suffix trees have $O(n)$ space cost, and most efficient implementations require at least 10-12 bytes per DNA character indexed (indexing up to $2^{27} - 1$ bp of sequence), excluding the storage of the string S. For large sequences, such as the human

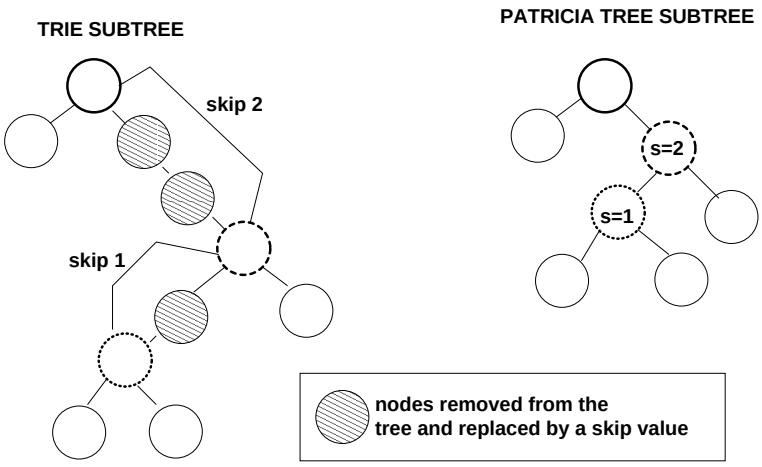


Figure 4.6: Transformation from a trie into a Patricia tree.

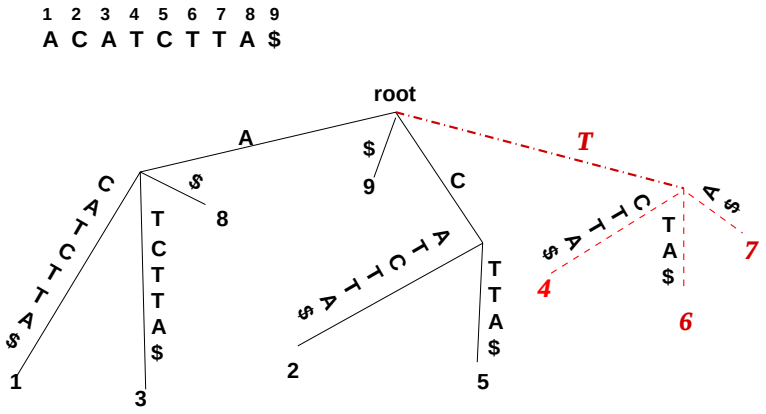


Figure 4.7: Suffix tree with a terminator **$** and traced search for **T**.

genomic sequence around 3 billion bases long, it is not feasible to have a memory-resident index, as RAM is still too expensive[5], and the index would take 40-45 GB of memory and some 9 hours to build. Recent suffix tree implementations [138, 140] optimise the space requirements of suffix trees and improve theoretical bounds. Our research builds on those methods to produce large tree indexes for genome-sized disk-resident data sets. The long term aim is to combine such database suffix tree implementations with probabilistic measures so that ultimately approximate matching of query to indexed sequence is possible, in the way similar to BLAST [7] but without incurring the cost of the sequential scanning of all data against the query sequence. The suffix tree is a suitable candidate index structure, because once an index is built, the search time for a query is predicted to be faster than for other data structures (for instance the suffix array). On the other hand, the suffix tree is characterised by very irregular topology which seems to stand in the way of efficient disk storage. Alternatives to the suffix tree exist, and we proceed to discuss those. In a further section we also discuss storage optimisations for suffix trees.

[5]However Celera Genomics (http://www.celera.com) use an Compaq Alphas with 32 and 64 GB RAM [226]

The suffix binary search tree

Suffix binary search trees are binary trees optimised for fast searching on strings [125, 126]. Figure 4.8 shows an example tree and the path traced along the tree during a search for string T. As in a common binary tree, the tree encodes an ordering of the nodes, in this case

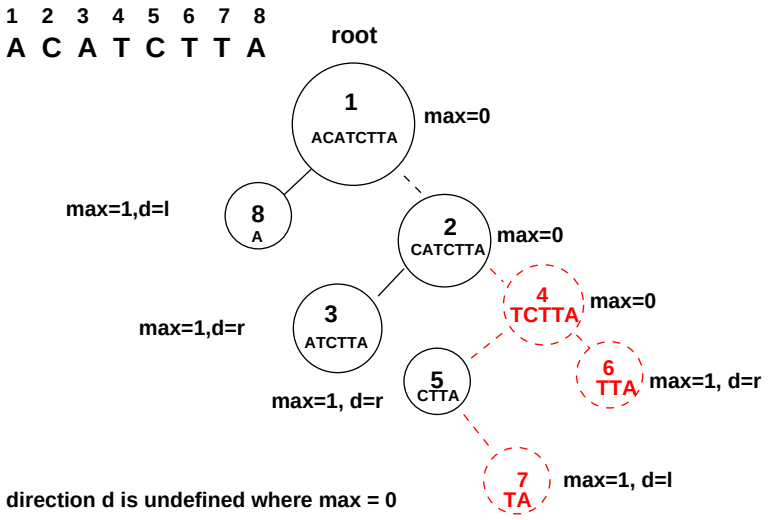


Figure 4.8: Suffix binary search tree, nodes 4, 6, and 7 contain indices of query **T**.

lexicographic ordering, and each suffix is a separate node. Reading all nodes "in-order" will produce a lexicographic ordering of all suffixes. Unoptimised string comparison, leading to the construction of an *ST* or an *SBST* has the complexity of at most $O(n^2)$, and in an *SBST*, measures are taken to reduce this time complexity to $O(n \log n)$, as explained further.

Tree suffixes can be inserted into the tree in any order. Given the fairly random distribution of bases in a DNA string, one can expect a balanced tree structure. To guarantee a balanced tree, rotations are required. This will carry an additional space cost and a performance cost, and is discussed in [125]. The complexity of building an SBST is $O(n \log n)$, due to an optimisation which places additional information in the tree nodes. In each node an integer is used to encode the maximum longest common prefix computed over all of the node's ancestors, and a Boolean value to encode whether this prefix comes from a left, or a right ancestor. For instance, for node 5 coding for suffix 5 **CTTA**, nodes 1, 2, and 4 are ancestors, and node 5 shares only one initial character with node 2 **CATCTTA**, and no initial characters with other nodes. Therefore the maximum longest common prefix is set to be 1 (max = 1), and direction value, indicating the direction is set to r (dir = r) because node 5 is in the right subtree of its ancestor node 2 with which it shares this prefix.

Given the values of the maximum common prefix and a direction indicator, the building (and tree searching) algorithm keeps track of the maximum number of characters matching with the node being inserted (or the query), and the direction of the ancestor which has this longest match. Any node further down the tree will only be involved in character comparisons if its maximum longest common prefix and direction indicate that a further comparison is needed. Once a node encoding the whole new string (or query) is found, a traversal of the subtree is performed to find all other occurrences of a query (or, in the case of tree building, a straightforward insertion follows). In an SBST the time required to

search for a string will be limited by the depth of the tree (as the path from the root to the node encoding the suffix will be traversed) and by the number of occurrences of the query string. Empirical work with the *SBST* structure is described in Chapter 5.

The suffix array

This data structure was introduced by Manber and Myers [148]. It is often used as the supporting data structure in q-grams (next page). The suffix array has a smaller storage requirement than a suffix tree or a suffix binary search tree. It can be built in $O(n \log n)$ time, which is slower than for a suffix tree (Manber and Myers show results for DNA where the suffix array takes 6 times longer to build than a suffix tree). The query time using a suffix array is $O(m + \log n)$ where m is the query length and n the length of the indexed text.

A suffix array is a lexicographically ordered array of suffixes accompanied by information about the *longest common prefixes (lcps)* of certain pairs of suffixes held in the array, which reduces the search to efficient binary search in the space of ordered suffixes. The space requirement of a suffix array is $12n\ bytes$ (assuming 4-byte integers, two integers hold the right and left *lcp* value, and the third integer is the suffix number[6]). This data structure can be used in both exact and approximate searching. A binary search compares the query string to the leftmost and rightmost strings for a particular interval, using the *lcp* values at the pivot point. Two values for each pivot point are kept, the *Llcp* and *Rlcp*. *Llcp* is the longest common prefix length shared by the pivot and the leftmost string of the current comparison, and *Rlcp* holds the longest common prefix length shared by the pivot and the rightmost string. During the search, superfluous character comparisons are avoided by keeping track of the last character compared, so that the complexity component related to the query size remains $O(m)$, and the binary search component is $O(\log n)$.

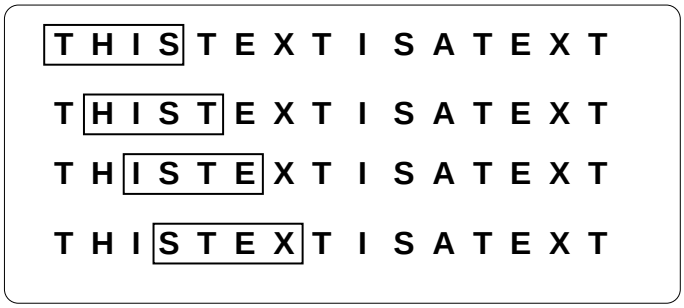
There are several algorithms for building the suffix array, and all require more than $12n\ bytes$ at construction time, due to the need for suffix sorting. For instance, a suffix array can be built from suffixes read from the suffix tree or suffix binary search tree [126]. Other suffix sorting approaches can also be used [148, 2]. If the suffix array were to be used in a database context, two problems would have to addressed. One would be the way to partition the array, and the other to optimise it for tree creation, exact searching and approximate searching. Array partitioning and optimisation would aim to minimise the number of disk reads required to scan it under different patterns of use. We believe that the naive approach of using one array to cover the whole range of suffixes indexed is not appropriate, as the parts of the array which are rarely used (the leaves) would be brought into memory together with the parts that are needed, both at tree construction time and during exact searching. During approximate searching the main requirement is to keep in fast memory the entire top of the tree. It appears that in analogy to the approach adopted in the B-tree, see Section 4.2.2, a flattened tree-like structure would be appropriate. Baeza-Yates and Navarro [25] show preliminary results in approximate searching using a suffix array, without revealing the full data structure implementation details, and we refer to their work in Chapter 6.

[6] Refinements of this encoding are possible, and they may reduce the space needed.

4.2.2 Data structures for database use

We discuss hybrid data structures which combine or modify the structures already discussed, as well as the *q-gram* data structure. Beside the q-grams, disk-resident *String B-trees*, *LC-tries*, and prefix indexes are introduced at this point.

Q-grams



INDEX TERM	POSITION
THIS	1
HIST	2
ISTE	3
STEX	4
TEXT	5, 11
EXTI	6
XTIS	7
TISA	8
ISAT	9
ATEX	10

Figure 4.9: Q-gram construction for *THISTEXTISATEXT* using a window of size 4.

By sliding a window of a predefined size over the DNA string, we can count all occurrences of substrings of a certain length, and record all locations of those substrings (q-grams) [169, 171], see Figure 4.9. This q-gram analysis can be performed on the query only, or on both text and query. Because q-grams can be converted to integers using simple coding schemes, very efficient string comparison is possible. Each q-gram is encoded as a number, and a comparison of the query q-gram to an index involves integer comparison which is faster than string comparison of several consecutive characters. This approach is very closely related to hashing techniques [59], and is used in many computing contexts, including routing techniques for networks[7]. Another name used for this approach in the information retrieval context is “n-gram” [86].

A q-gram which is to be used as an index requires a data structure on disk. A suffix array may be used in this context. The actual position of the q-gram within the text may be replaced with a pointer to the block of text in which it is placed, and then the whole block has to be examined for the occurrences of the q-gram. Q-grams have been shown

[7] for instance in the Web Cache Communication Protocol, see http://www.cisco.com/warp/public/732/wccp/-history.html

to be effective for approximate string searching in several biological contexts [44, 157, 7]. Suffix indexes which we focus on are an alternative to hashing, and have been researched and made use of to a much smaller extent. As we did not perform experiments with q-grams, we do not elaborate on this issue further. We only note that q-grams were used with ESTs (expressed sequence tags, which are short DNA sequences around 500 characters long) [44] to enable EST clustering and quality control, with datasets of around 300 Mbp. In this context q-grams were found to be effective in searching for very similar sequences. The tests of q-gram algorithms performed by Navarro [168] suggest that these structures are only appropriate in the context of low error levels, and are inferior to suffix arrays.

String B-tree

We start with an overview of the B-tree which is needed as background for the *String B-tree* (the authors spell *String* with a capital S). A B-tree structure is one of the main data structures used in database technology, and indexes fixed-length keys. A detailed analysis of B-tree operations and properties can be found for instance in [59]. A B-tree uses the disk

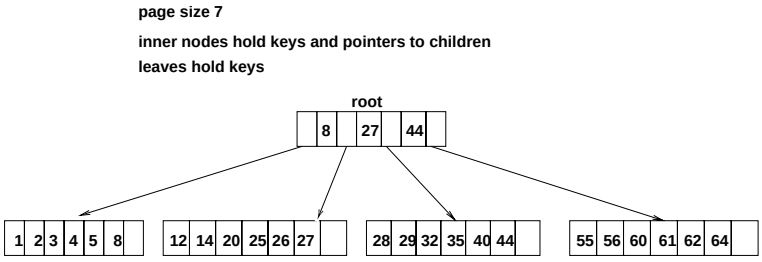


Figure 4.10: A B-tree indexing a set of integers.

page size, b, as a unit of storage of the tree nodes. Each inner node or leaf fits on one page (or one disk access unit of size b). Each leaf has space for a fixed maximum number of keys, and each inner node contains keys and pointers to other nodes. B-trees have a high branching factor, and are balanced, to guarantee constant time key access. A sample B-tree is shown in Figure 4.10. Because keys in a B-tree tend to be of the same size, this often results in a limit on a string key to lie around 255 characters, and this limit is constant in a given database system.

The String B-Tree [81] was developed to serve as a database index to string data. It is a combination of a Patricia tree with the B-tree. Patricia tree is a compact representation of a trie where all nodes with one child are merged with their parents (see preceding sections). B-trees guarantee the worst-case search time to be proportional to the height of the tree. To guarantee that property, there has to be a limit on a composite key length, or more complex indirect addressing methods have to be used. In fact, pure B-trees have a limited power to support searching on DNA data, as DNA cannot be broken into words. String B-trees only solve the problem of variant key lengths over relatively short words as normally a few keys would have to fit on a disk page. It seems unlikely that they would solve the problem of indexing very long DNA suffixes. So far no test results for large biological sequences are available. In a String B-tree sets of strings are sorted lexicographically, and each leaf of the String B-tree stores the ordered string set and a Patricia trie built on that set to enable fast searching within the leaf. Internal nodes hold indices of left- and rightmost strings for each

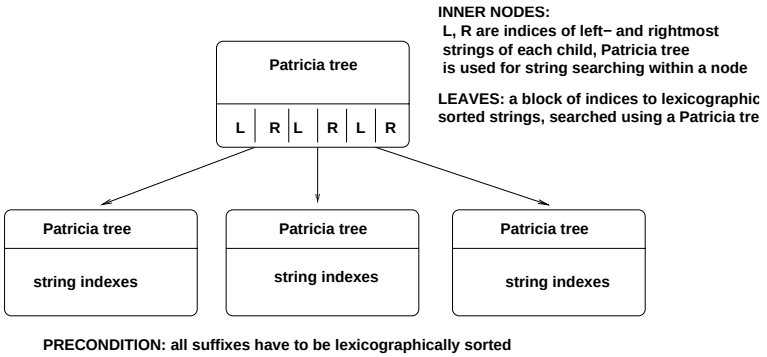


Figure 4.11: A String B-tree index structure.

of their children, and a Patricia trie for searching on those strings, shown in Figure 4.11. Both internal nodes and leaves are limited to page size, to optimise data retrieval from disk. Algorithms for String B-tree construction, update and searching have good theoretical bounds, but are very complex. Because of the underlying requirement to sort all strings (suffixes), it is possible that performance problems similar to those observed for DNA in string sorting for LC-tries might arise (see next page). In fact, a String B-tree is just another way of implementing a suffix tree on disk, but one that could be useful for indexing short strings like exons which are constituent parts of genes.

Prefix index

This work by Jagadish and colleagues [127] is aimed at general-purpose string indexing, and it claims that the results are applicable to unbounded length strings. One of the motivations behind it is to enable multi-dimensional indexing for the retrieval of XML data. The authors contribute the following.

- They present a way of mapping strings to rational numbers, using fractions.
- They represent strings of unbounded length in an index leaf page by a fixed length offset to an external key.
- They store multiple elided tries (Patricia trees, see Section 4.2.1), one per indexed textual dimension, in one index page. This helps them to prune search over several string dimensions during the traversal of index pages.

The mapping of strings to numbers ensures that extensions of a particular prefix are clustered while preserving lexicographic ordering. The mapping function uses the reversal of commonly used coding using Horner's scheme, and treats the first letter as the most significant constituent of a number, with further letters contributing less. The range of the mapping is $(0, 1)$ and strings are fractions in base $\alpha + 1$ where α is the alphabet size. A string $S = s_1 s_2 .. s_n$ is represented by

$$t_1/(\alpha+1) + t_2/(\alpha+1)^2 + .. + t_n/(\alpha+1)^n$$

where t_i is the code for s_i. One challenge in the implementation of this solution lies in representing long fractions accurately. The other challenge lies in distance metrics for strings,

which is only solved for prefix matches. It seems that this solution is of limited use in DNA or protein matching.

The authors propose to store long strings in index pages as pointers to external data structures. We are not sure if this representation is going to be useful in comparing very long strings.

Elided tries are a way of circumventing memory problems in the face of externally stored keys. An elided trie is closely related to a Patricia tree (see preceding section), and the authors come close to specifying a suffix tree as the structure of choice within a tree node. They suggest that only the first letter at each branching node is stored, together with the skip value. This means that after a "successful" attempt to locate a string, we still need to do the exact string comparison, as the elided trie stores only a part of the string it indexes. Database operations like searching, insertion, split and merge are then defined in terms of elided tries.

The index performance is tested using an implementation of string B-trees and string R-trees. The string B-tree is said to perform better, however full supporting data for the experimental analysis are not quoted. The total volume of data is unclear (200,000 2-dimensional strings) and details of queries are not available (100 prefix queries of "low, medium and high selectivities"). The number of disks accesses for both data structures is compared, and leads to the conclusion of the superiority of string B-trees in prefix searching.

LC-tries

Level-compressed tries (LC-tries) [11] are tries which use path compression, level compression and data compression to build an efficient implementation of a suffix tree. The authors use the technique of "adaptive branching" where the number of descendants of the given node depends on how the tree nodes are distributed, and they arrive at an efficient representation of a binary trie.

We first describe this approach conceptually. A binary trie of all suffixes is built, based on Huffman [110] encoding of the text. Then, if the i highest levels of the tree are complete (fully filled with nodes), but level $i+1$ is not, we replace the i highest levels by a single node of degree 2^i. This replacement is repeated top-down to produce a level-compressed trie (LC-trie). In this structure the expected average depth is much smaller than that of a trie. We call this trie a level-compressed trie. Then we add path compression. Each node with only one child is merged with this child, and a skip value of each node, which is initially set to 1, is updated by 1. After this operation has been applied recursively, nodes with large skip values appear in the tree, and they code for long paths of bits, i.e. achieve path compression. The implementation of an LC-trie uses an array, and each node is followed by its siblings. A node is represented by three numbers: *number of bits (characters) to be skipped*, relative to the parent node, *position of the leftmost child* (a pointer), and *the number of children*. The number of children will always be a power of 2, and can be represented by a small number of bits. Further compression is achieved by replacing two fields in each node record, called "branch" ($\log$ of the number of children) and "skip" with one integer, one bit of which distinguishes between the two possible types of value. This is possible for internal nodes which have either a positive skip value or a branching factor larger than 2.

The algorithm for the construction of an LC-trie can use a suffix tree for all suffixes which is transformed into a binary trie and compressed. The limitation to have a suffix tree first, which requires much more space (the most important limiting factor in suffix

tree creation), and the complexity of the algorithms required to compress the tree, make this structure an unsuitable candidate for prototype development. Another possible method of construction is based on sorting all suffixes first, placing them in an array, and then constructing the tree top-down. For disk-based implementation of the LC-trie the authors use a partial LC-trie stored in main memory and used as an index into a suffix array stored on disk.

The authors abstract from the difficulty of dealing with the non-uniformity of real experimental data, and present simulation results for exact searching, by counting the number of disk accesses which would have to be made. The datasets used range from 2 to 140 KB for both main memory and secondary memory implementations. For secondary memory simulations on most datasets, except DNA and random text, the LC-trie appears superior to a suffix array based on buckets. Because of the nature of experimental data provided by the authors, we find it hard to make judgements about the possible practical advantages of using this data structure. We believe that further experimental work in this area is justified.

4.2.3 Compact suffix trees

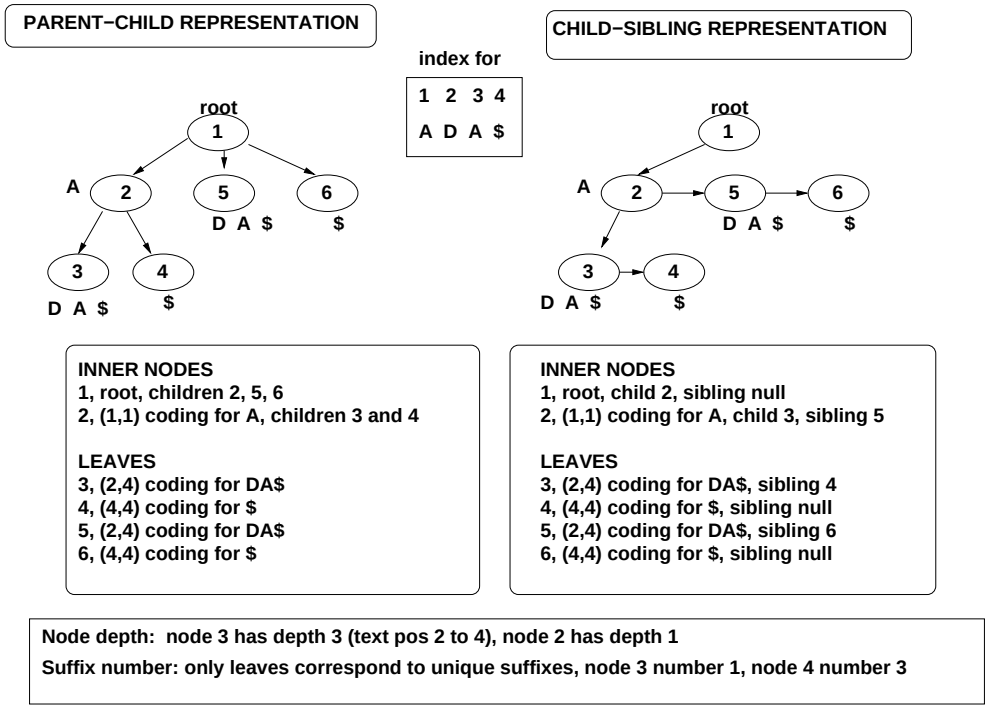


Figure 4.12: Two possible representations of a suffix tree.

As the suffix tree is the data structure of choice in string searching, and the only disadvantage in its use is its excessive space requirement, efforts have been made to compress it. We report here work done by Kurtz, and Clark and Munro.

Kurtz [138] presents 4 storage techniques. Those 4 different implementations are then tested with different data sets, and actual times required to build the tree are measured. Kurtz then classifies known bioinformatics applications into two groups: those that traverse

part of the tree, and those that perform a full tree traversal, referring to Gusfield [99], and presents implementation guidelines for suffix trees, based on the prospective application. (In another paper, [139] he presents an application of his trees to the finding of repeats in sequence data.) He finishes by concluding that 40.5 GB of main memory would be required to build a human genome tree, and the building of that tree would take less than 9 hours, on a 64-bit architecture, implemented in optimised C. We proceed to discuss the 4 implementations: the Simple Linked List, the Simple Hash Table, the Improved Linked List, and the Improved Hash Table.

We introduce the relevant terminology first. For further details the reader may want to consult [99]. The suffix tree consists of internal nodes and leaves, as shown in Figure 4.12. In the representation of the tree based on linked lists, internal nodes have children and siblings, while the leaves have no children (but may have siblings). In other possible tree representations an inner node will explicitly or implicitly lead to its children, and the notion of sibling may not be used. Each node codes for a substring, and internal nodes correspond to substrings which appear more than once in the text. The substring the node codes for will usually be identified by the left and right index into the text, shown as a pair of integers in brackets. The length of the text indexed by the node will be the node's "depth". Suffix number will be the starting index of the suffix that a given leaf corresponds to (this number may or may not be stored in the leaf). A suffix link will lead from a node indexing a string aw where $|a| = 1$ to the node indexing w. This means that if we trace the string aw from the root down to the node, we can traverse the suffix link and find the node which is the end of the string w, as traced from the tree root. Further details and figures showing suffix links and the tree structure are provided in Chapter 5.

Simple Linked List Implementation (*SLLI*)

The simple linked list implementation uses two tables, one for leaf nodes, and one for branching (inner) nodes, see Figure 4.13. Leaf nodes are indexed by leaf number and have an entry indicating the right sibling node number, or null if there is none. Inner nodes are numbered separately, in the order of their headpositions (explained below) and their entries contain node numbers for child, sibling, and suffix link, string depth of node (integer) and an integer which is called *Headposition*. Headposition is defined to be the leftmost branching occurrence of the substring indexed by this node, shown in Figure 4.14. This is in contrast to the conventional annotation of a node by the leftmost substring which gave origin to that node during tree creation (MOO would have had the index 400). Headposition is available during tree construction, directly in McCreight's algorithm, and also in Ukkonen's algorithm, whenever a node is being split. Formally,

$head_1 = \epsilon$
$head_i$ is the longest prefix of S_i which is also a prefix of S_j for some j between 1 and $i - 1$.

Using headposition instead of the leftmost string occurrence requires explicit calculation of the left and right labels of each node during tree traversal. Those labels are calculated by adding depth values of the preceding nodes, during the descent down the tree. Storing the depth of the node instead of the length of the incoming edge to a node is supposed to be useful in some tree traversals, and can be exploited in storing the tree more efficiently (see hash table implementation — next section).

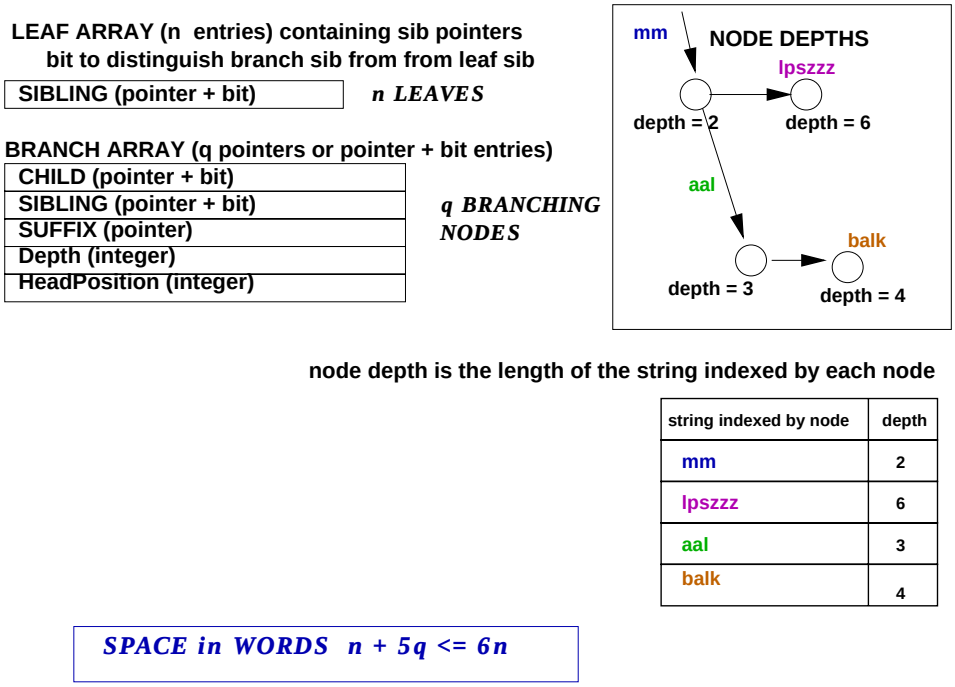


Figure 4.13: A Simple Linked List Implementation of a suffix tree.

Simple Hash Table Implementation (*SHTI*)

The main reason for investigating a hash table-based implementation is to guarantee constant-time access to all children, which may be important for large alphabets, as a traversal of an average of 10 links in the protein alphabet of 20 might be inefficient. A simple hash table implementation, shown in Figure 4.15 is based on using a hashcode for a combined key $(headposition(w), a)$ where w is a substring, as traced from the root, and a is the first character of the outgoing edge from the node indexing w. Headposition refers to the headposition of the parent node. Since the number of edges is bounded by $2 * n$ (where n is text length) this limits the size of the hash table. Beside the hash table, another table stores edge records recording string depth, headposition and suffix link. This table requires $3q$ integers, where $q < n$. The hash table itself stores two integers, the hashed value, and the headposition, as the character a can be retrieved in constant time (by looking up the text array using the headposition value). As there are maximum $2n$ entries in the hash table, the total space requirement is $4n + 3q$ words.

Similarly to the SLLI, the initial character of each node is not stored but looked up. The hashing technique used is open addressing hashing with double hashing to resolve collisions. The size of the hash table is a prime number larger than $2n$. Details of hashing schemes can be found for instance in [59].

Further possible space savings

Kurtz explores suffix tree redundancies to arrive at his compressed storage scheme, which is then implemented in two versions, an improved linked list implementation (ILLI), and

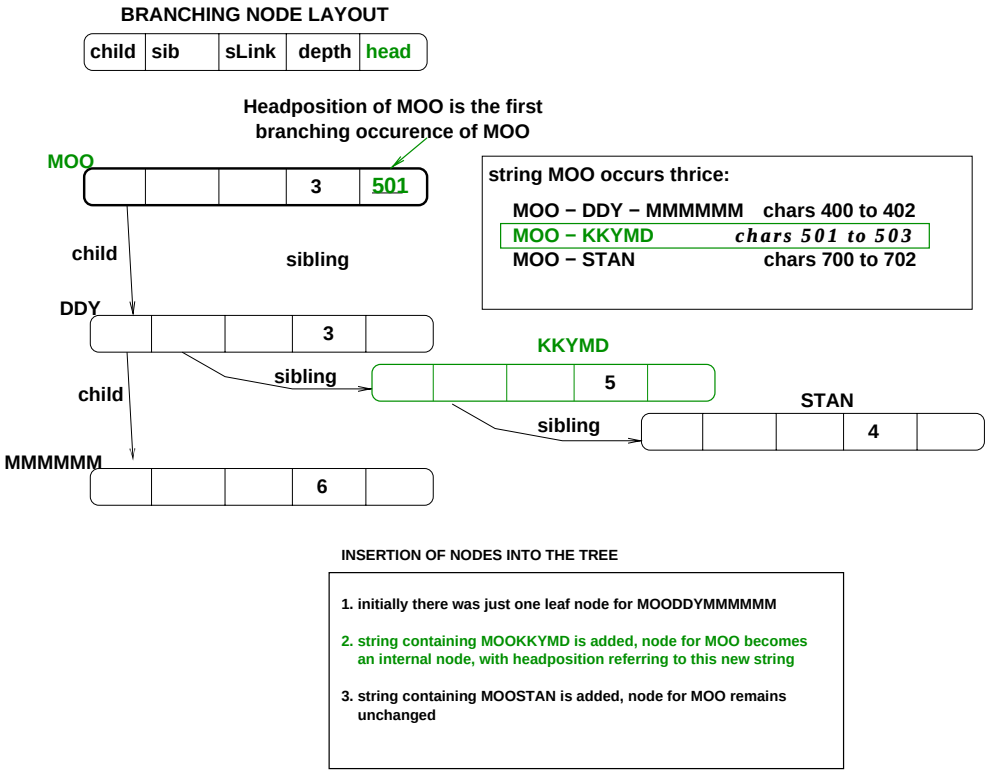


Figure 4.14: Headposition example.

improved hash table implementation (IHTI), see the following sections. Kurtz finds a relationship between nodes connected by suffix links. He observes that headpositions are unique node identifiers, and a suffix link always leads to a node with a headposition decreased by at least 1. Based on that observation, a node aw is classified to be small if its suffix link's headposition is smaller by 1 ($nodenumber(aw) + 1 = nodenumber(w)$), or large otherwise ($nodenumber(aw) + 1 > nodenumber(w)$), i.e. small nodes are always directly followed by another branching node, and the last branching node is a large node. Accordingly, sequences of branching nodes connected by suffix links can be partitioned into chains of small nodes followed by a single large node. Kurtz observes that branch records for small nodes have redundant information. Namely, within a chain $b_l, ..., b_r$, for $l <= i <= r$ there exist the following relationships:

1. $b_i.depth = b_r.depth + r - i$
2. $b_i.headposition = b_r.headposition - r + i$
3. $b_i.suffixlink = b_{i+1}$

Those are exploited in the improved linked list implementation *ILLI*.

Improved Linked List Implementation (*ILLI*)

The improved linked list, see Figure 4.16, distinguishes between large and small nodes. Small records store $w.distance$, $w.child$ and $w.sibling$ where $b_i.distance = r - i$ is the

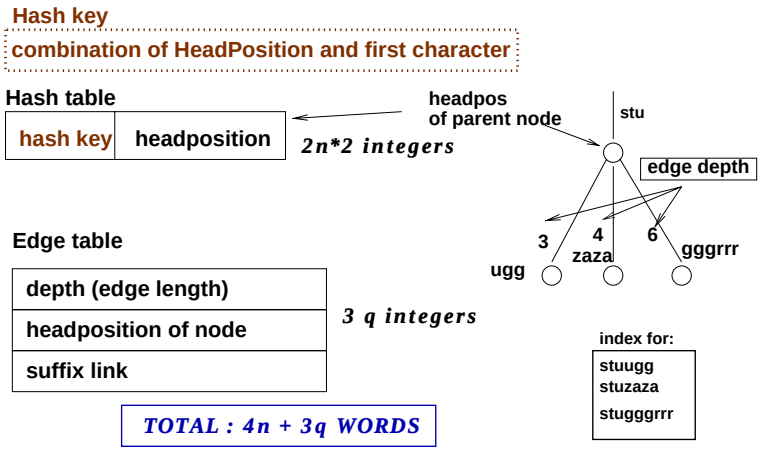


Figure 4.15: Simple hash table implementation.

distance to the large node, and they occupy 2 integers (under the assumption of maximum indexed string size smaller than 2^{27}). Large records occupy 4 integers, and they store the child, sibling, depth/suffix link and headposition. This compact representation uses a parameter α to store both depth and suffix link more compactly. If $depth < 2^{\alpha} - 1$, then both the depth and suffix link are stored within the large record, and this record is complete. If the depth value is equal to $2^{\alpha} - 1$, or exceeds this value, it is stored as follows. Let v be the rightmost child of this large node (call it w). Between w and v there are at most $k - 1$ references (k is the alphabet size). If v is a leaf, then leaf table will have a null sibling reference for v, and one bit in the leaf entry will be set to mark that there is a null sibling. Then the integer in the leaf record has unused bits, which will store the suffix link. So, to find the suffix link for an incomplete large node, all its children are traversed, and the suffix link is found. Similarly, if v is a branching node, then its sibling will be null, and will encode the suffix link. This special encoding is used only for large nodes whose depth exceeds $2^{\alpha} - 1$, and α will be chosen such that these nodes are usually very rare.

To guarantee navigation within the branch table, nodes are ordered by their headpositions, and they are referenced by their *base address* in the branch table (the index of the first integer of the corresponding record).

Space requirement for this representation is calculated as follows.

- We assume a base address stored in β bits. A reference is either a base address or a leaf number, and we need 1 bit to distinguish those, so we use $\beta + 1$ bits.
- Each depth and each head position require $\gamma = \lceil \log_2 n \rceil$ bits.
- Distance values are compressed using a constant δ to distinguish between small and large distances which are encoded differently. In the worst case (for a string a^n), where there is only one chain of length $n - 1$, maximum distance value is $n - 2$. However, in most other cases, distance can be constrained to be 2^{δ} maximum, and the maximum length of a chain will then be 2^{δ}. So, for a longer chain, an artificial *large* node can be introduced (a *large record*, taking up 4 integers of space), and distance can be encoded using δ bits. This saves $\gamma - \delta$ bits for most records in a chain.

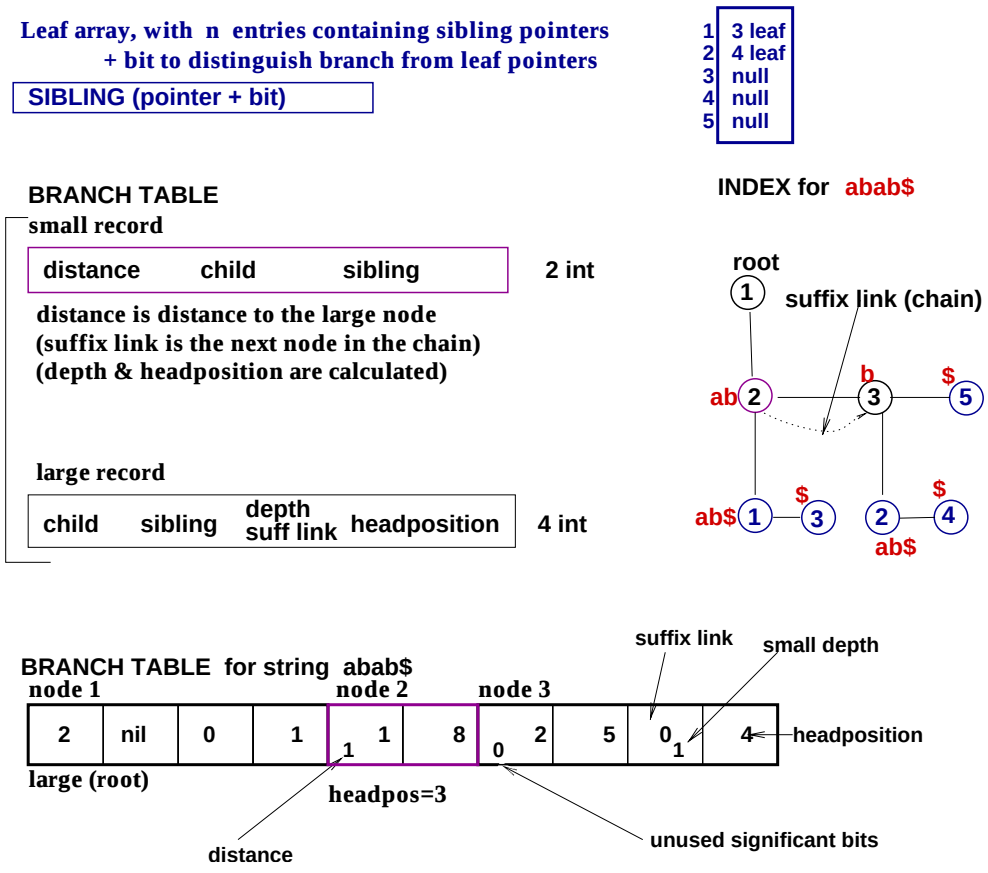


Figure 4.16: Improved linked list implementation.

The size of a *small* node is then $2 \cdot (1 + \beta) + \delta + 1 = 3 + 2\beta + \delta$ bits. A *large* node stores two references (child, sibling), one nil bit, and one *complete* bit to indicate if the node is complete or incomplete. The head position takes up γ bits. In a complete record β bits are used for the suffix link, and α bits for the depth. In an incomplete record, γ bits are used for the depth. δ bits remain unused, to store the distance of 0. This allows us to distinguish between large and small nodes, as small nodes always have $distance > 0$. Altogether the large node will occupy:

complete node $2 \cdot (1 + \beta) + 1 + 1 + \gamma + \beta + \alpha + \delta = 4 + 3\beta + \gamma + \alpha + \delta$ bits, and
incomplete node $2 \cdot (1 + \beta) + 1 + 1 + 2\gamma + \delta = 4 + 2(\beta + \gamma) + \delta$ bits.

Taking into account the constants ($\alpha = 10$ and $\delta = 5$ are chosen), and assuming the size of input string to be $2^{27} - 1$ maximum (which implies $\gamma = 27$), this produces a small record of 2 integers and a large record of 4 integers (32-bit integers). This leads to even addresses, max address being $4n - 4$, and $\beta = \gamma + 1$, so that a small node requires 64 bits, incomplete large record 119 bits, and complete large record 130 bits. The 2 bits missing in 4 bytes to represent a large record are borrowed from the leaf entry. Every leaf entry has 2 spare bits (the leaf entry with the same index as headposition is used) as it uses 29 bits for the reference and one nil bit, so 2 bits are spare.

In this representation, each leaf and each large node saves one integer, and each small node saves three integers (as compared to *SLLI*), so if there are many small nodes, large space savings are achieved. The tree building algorithm used in this representation is based on McCreight's work [153], and branching nodes of the tree are constructed in order of their head positions.

Improved Hash Table Implementation (*IHTI*)

A similar optimisation to that applied in ILLI, which relies on the distinction between small and large nodes, can be used in the hash table-based implementation. In this case only the records for large nodes are stored, and small nodes entries for suffix link, head position, and depth are calculated, relative to the large node entry and the so-called *reference pair*. A reference pair is defined to be $(0, j)$ for the leaf j, and a large node is referenced by the reference pair $(1, i)$ where i is the node's index in the sequence of all large nodes of the tree ordered by their head positions. With reference to a large node, all small nodes which create a chain leading to this large nodes, are referenced by pairs $(d + 1, i)$ where $d > 0$ is distance from the large node. To implement this space saving, the hash key remains the same (combination of head position and first character of a node), but the hash table now stores the pair (head position, reference pair) plus the hash key.

Another space saving in the size of the hash table results from the edges which end in the terminator symbol. For instance inner nodes 2 ab and 3 b in Figure 4.16 lead to leaves ending in the terminator symbol (leaves 3 $ab\$$ and 4 $b\$$). These edges are called *identity edges* and they can be deduced. There is at most one identity edge for each branching node. This edge will not be explicitly stored in the hash table, but a single bit will be used to mark the existence of such an edge.

This hash table implementation uses a single integer to encode the reference pair. The maximum chain length is restricted to 31 (32 was used for ILLI), and therefore maximal distance limited to 30, i.e 5 bits are required for the first element of the pair. 27 bits are then used for the second component (leaf number, or number i of large node). This limits the indexed string length to $2^{27} - 1$. Thus the space requirement of IHTI is $4n + 3\lambda$ where λ is the number of large nodes.

Discussion

- *Sequence length.* The maximum sequence length that can be indexed using Kurtz's methods is currently 134 million, which is not sufficient to index large mammalian chromosomes (up to 263 Mbp) or the fly genome, but will satisfy the needs of bacterial genetics. At the price of additional storage, the same storage optimisations can be used to address larger sequences, with increased storage cost for longer addresses.

- *Sequence type.* The worst case implementation of ILLI is $5n$ integers, and it is $7n$ integers for IHTI. Therefore, for the DNA alphabet one would opt for the ILLI implementation, as traversing up to five siblings (two an average) should not present a significant overhead. However, for proteins, it might be more suitable to use a hash table implementation IHTI. Kurtz's experiments were run on a small computer (Sun-UltraSparc, 300 MHz, 192 MB RAM), and it is to be expected that smaller data sets will produce structures with different topologies from larger data sets (for instance the top level of the tree will not be dense for short sequences), therefore any evaluation of

the relative speed of both implementations would have to be assessed with large data sets, to be significant, as there is no theoretical work predicting the expected branching or the relative numbers of small and large nodes. Hashing, on the other hand, might be slow, as modulo operation required may be a bottleneck, unless masking is used.

- *Traversal type.* Long sibling chains which have to be traversed during the tree build and exact searching may slow down computation. However, in approximate matching, where all nodes at the top of the tree are to be traversed, it may not matter whether we use a hash implementation or a linked list implementation. Indeed, it may be more efficient to use linked lists instead of trying to compute the hash function for all the possible letters of the alphabet.

- *Linked list versus hash table.* The ILLI consumes less space than IHTI, and this may be important in some applications. In exact matching hash access to nodes may be more efficient, but in approximate matching it will not be more efficient as most nodes near the root will be visited.

- *Running time.* The building time of Kurtz's suffix tree is acceptable for genomes around 100 Mbp (13 to 18 minutes, see Chapter 5), provided sufficient memory is available (1.3 GB was the reported use of memory for the *C.elegans* genome of 97 Mbp on our hardware configuration). However, for larger data sets (the human genome), the time to build will be around 9 hours, and the memory required around 40-45 GB, which is not viable for interactive use of a suffix tree index. For large genomes, persistent suffix tree indexes would be much more attractive.

- *Java.* Re-implementing most of the optimisations techniques discussed above in Java would require a departure from our desire to compare and contrast available data structures without investing a lot of time in complex coding schemes. The simplest strategy available in Java would be to use inheritance to encode different types of nodes. The nodes could be classified into: leaf with no sibling, leaf with sibling, inner node with no sibling, and full inner node. Further refinement and the distinction between small and large nodes could be made, however information sharing between different nodes would run against the grain of providing distinct objects with distinct services, unless we see the tree as one object. On the other hand, if we considered a Java implementation following Kurtz's ideas, we could achieve a high level of tree compression. We would then have to devise an appropriate interface for using this efficient disk-based data structure. We believe that a whole variety of possible space optimisations should be studied, in order to characterise different compact tree encodings and compare their performance.

Compressed suffix tree

This work by Clark, Munro, and others [55, 164], available in two papers which we refer to as the earlier and the later paper, aims to achieve the best possible theoretical bounds for suffix tree encoding and searching. The earlier paper quotes some test results and the later paper is purely theoretical. We did not explore this approach because we wanted to investigate more direct suffix tree mappings first. We summarise the main contributions here, as they are relevant to future research in this area.

The first step in the tree compression is binary encoding of the indexed string (including the terminator symbol). A similar compression scheme is advocated by Larsson [140]. All suffixes are encoded in binary and placed in a trie. Queries are coded in binary as well before being compared with the trie. This trie is a binary tree in which all the internal nodes have exactly two children, and $n + 1$ suffixes will give rise to n internal nodes and $n + 1$ leaves.

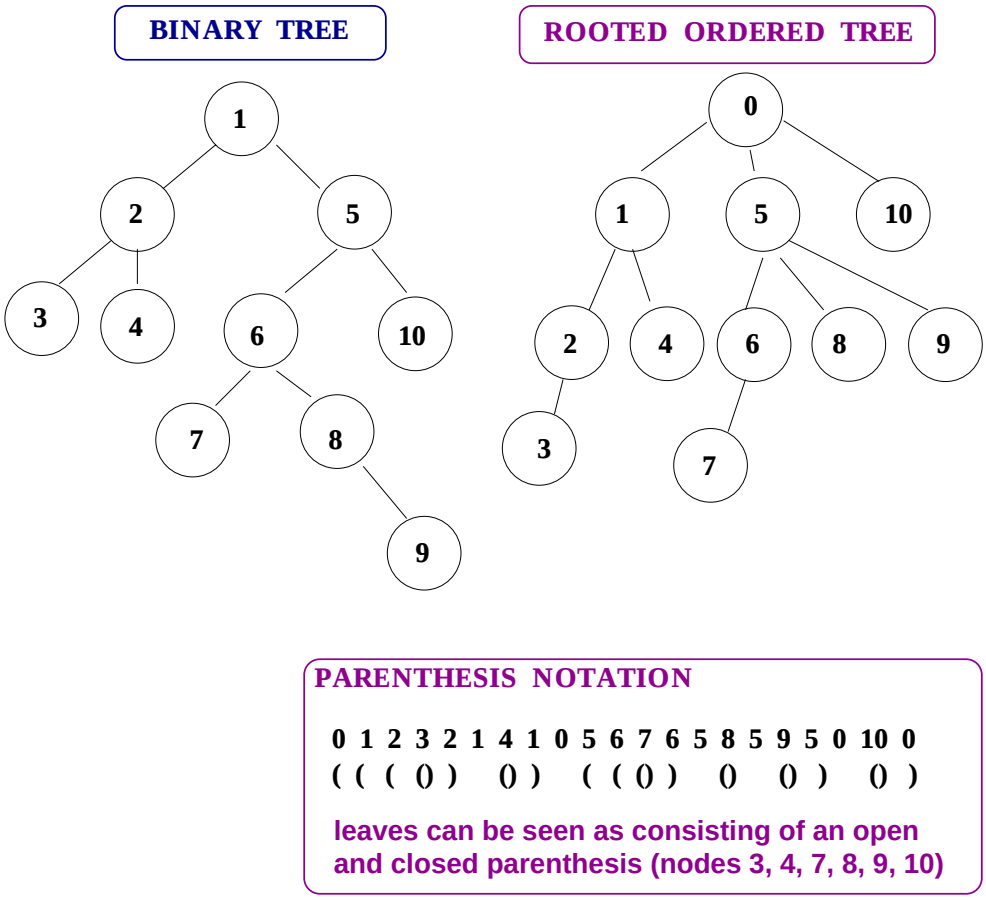


Figure 4.17: An illustration of the isomorphism between a binary tree and a rooted ordered tree and its parenthesis encoding.

Both papers share some underlying tree representation features. In the 1996 paper, the tree representation is based on the isomorphism between the class of binary trees and the class of rooted ordered trees [163]. This mapping allows for representing nodes which may have just one child, using the notation of ordered parentheses. An example of this mapping is in Figure 4.17. Parenthesis notation is based on pre-order traversal. An open parenthesis is written when a node is first encountered while going down the tree, and a closing parenthesis is written on leaving the node to go back up the tree. In the ordered tree there is a spurious root not corresponding to any of the binary tree nodes. The left child of a binary tree node corresponds to the leftmost child of the corresponding node in the ordered tree, and the right child corresponds to the next sibling to the right in the ordered tree.

In this encoding, additional information needs to be stored, to enable navigation and to

determine the size of the subtree rooted at that node. This encoding (1996 paper) is related to the schema used in LC-tries where skip values are used to encode the number of children. The details of the tree operations are intricate, and based on maximum space economy possible, but theoretically constant time access to parent and child nodes is guaranteed. Like a suffix tree node, each node stores a string offset of the first bit where the prefixes of any 2 suffixes first differ. Since there are only 2 children at any point, bit 0 will branch to the left and bit 1 to the right. Leaves store the offsets of the appropriate suffixes.

Searching in the tree described in the earlier paper is described by the authors as a blind search with skipping, so that when a "matching suffix" is identified, it has to be verified if it actually matches (this is similar to the String B-tree implementation, discussed above). Because for each node the first character is only known (0 if branching left, 1 if branching right), only that character is compared with the query. When the comparison is complete, and potential matching suffixes have been identified, starting from the suffix number(s) indicated by the leaves, a full string comparison has to be performed to confirm a match. In several tree versions described in the later paper the nodes are stored as suffix arrays, and searching follows a slightly different pattern.

The 1996 paper reports testing results, however the exact data sizes are not given. The largest example, Oxford English Dictionary, seems to occupy 18,409 disk pages of 100 K each, and corresponds to 0.5 billion index points. It is not clear here if index points refer to words or the size of the binary string indexed. The index size is 1.8 MB and index creation needed 5 to 6 hours a fast machine with 32 MB of RAM (in 1996). The recent paper does not report empirical work.

4.2.4 Reflection on data structures

Two major options for indexing very long strings seem to be viable. One is based on q-grams and the other on different derivatives of suffix indexes. Q-grams implemented as suffix arrays pointing to buckets of string pointers are one alternative. They are more space-efficient than suffix trees and seem to offer good performance for close to exact matching tasks. They have been tested with large genomic strings, but only in the context of high similarity matching. On the other hand, suffix tree derivatives offer theoretically faster searches for exact matching, but are require much further optimisation. Our current implementation of suffix trees allows for building trees of any size, and offers fast approximate matching at a lower degree of similarity. A final judgement on the appropriateness of using the alternative data structures will not emerge for a while, until all the possible implementations are tested with large data sets. Before we move on to the empirical evaluation of both approaches, we turn now to the discussion of exact and approximate matching algorithms.

4.3 Exact matching algorithms

Good overviews of exact string matching algorithms can be found on the web[8], in Gusfield [99], or in Apostolico and Galil [14]. Our interest lies in the area of searching on pre-processed text, to use the terminology of the algorithmics community. This approach creates an index to the text which is independent of the query string to be used. Beside several data

[8] http://www-igm.univ-mlv.fr/∼lecroq/string/

structures with high storage requirements (*directed acyclic word graphs (DAWGs)* [35]), following efficient data structures are used in this area:

- the suffix tree,
- the suffix array,
- the suffix binary search tree (SBST),
- the q-gram.

We discussed the four data structures in previous sections. We recapitulate here the main differences with respect to searching.

- Suffix tree exact searching times are proportional to the sum of the query length and the number of hits. For short strings, the number of hits reported is extremely high and will dominate query cost.
- The suffix array can be searched in time proportional to the sum of the query length and the logarithm of text length. For queries with many hits the number of results to be reported may influence the speed in a way similar to that seen in the suffix tree. We do not know how to guarantee the locality of disk access in a suffix array and storage schemes and memory buffering arrangements may be required for large structures of this type.
- The SBST can be searched in time proportional to the sum of tree height, query length, and the number of query occurrences. Tree height for a balanced tree would correspond to the logarithm of string length, and experience shows that is the case. We present results in Chapter 5 which show that the SBSTs we tested deliver exact matches faster than suffix trees.
- Q-gram searching times depend on the actual searching technique used. In the most common scenario, the query is first changed into a $q-gram$ by sliding a window of size q over the text, and creating an index (for up to $m-q+1$ terms). Subsequently, hashed retrieval of matching cells from the suffix array follows. The cost of querying will be proportional to the number of distinct q-grams in the query (which is proportional to the query length) and the average size of buckets (if buckets occupy more than one disk page), which is proportional to string length.

4.4 Approximate matching algorithms

Baeza-Yates and Navarro offer good overviews of theoretical and experimental properties of many approximate string searching methods [27, 168]. We follow their classification of approximate pattern matching methods and place those methods in the context of biological text searching. Only a high-level view is presented here, and more detail is available in Chapter 6, in the context of experimental work.

The problem is usually stated as follows: given a short pattern P of length m and a long text T of length n, and a measure of similarity (for instance the number of errors allowed k), find all the text positions where the pattern occurs with at most k errors (or all positions where the similarity between the pattern and the text is above a given minimum). Most

theoretical work centres around the Levenstein distance [142] which for any two strings A and B is defined as follows.

> Given the three basic string operations of character deletion, insertion and substitution, find the minimum number of such transformations which change string A into string B.

In biological applications the distance model is complex, and often instead of the distance, similarity between two sequences is of interest. In most cases the expert biologist user reserves the right to judge the "goodness" of the match produced by the matching software.

The taxonomy of approximate matching approaches consists of four species: dynamic programming, automata, bit-parallelism, and filtering. In their presentation Baeza-Yates and Navarro leave out the methods based on indexing, as in their opinion those are still in need of much more research.

4.4.1 Dynamic programming

Dynamic programming is based on a matrix calculation where one dimension is the text and the other the query. There are several ways of defining the goodness of an approximate match calculated using a DP matrix. One of the approaches uses the edit distance between the pattern and the text. Computing science theoreticians look at the matrix from this point of view, and try to identify cells in the matrix which are close to the bottom right-hand corner and have a low edit cost. In the biological context, however, a different model prevails. In biology not the cost but the *similarity* of two sequences is of interest. The cell with the maximum similarity value can be found anywhere within the matrix. In particular, partial matches on short sequence fragments are also of interest. Often in biology the similarity of *real* interest is when the two sequences which code for proteins will have similar or identical functions in the biological context of interest. Finding such similarities is beyond both our biological understanding and our computational capacity. We compare both approaches in Figure 4.18. The formulae used to calculate the edit distance and the similarity are related. In the edit cost approach, there is a cost function which maps from every pair of characters to an integer or real cost:

$$EditCost : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}$$

In most theoretical work, edit cost for unequal character comparison is assumed to be a constant 1 (the same for insert, delete or mismatch), and 0 for a match. Under those assumptions, values in the edit cost matrix are calculated as follows:

```
DP[0,j] = 0
DP[i,0] = i
DP[i,j] = if char[i] = char[j], then DP[i-1,j-1]
                else (1 + min (DP[i-1,j], DP[i,j-1], DP[i-1,j-1])).
```

The cost calculation for every cell involves a look-up in the cells directly above, directly to the left and diagonally up-left. For the edit distance the formula always compares the current pair of characters, and if they are the same, it copies the cost value from the cell diagonally up-left. If the currently compared characters are different, the minimum cost from the look-up in 3 cells is incremented using the cost function (here equal to 1).

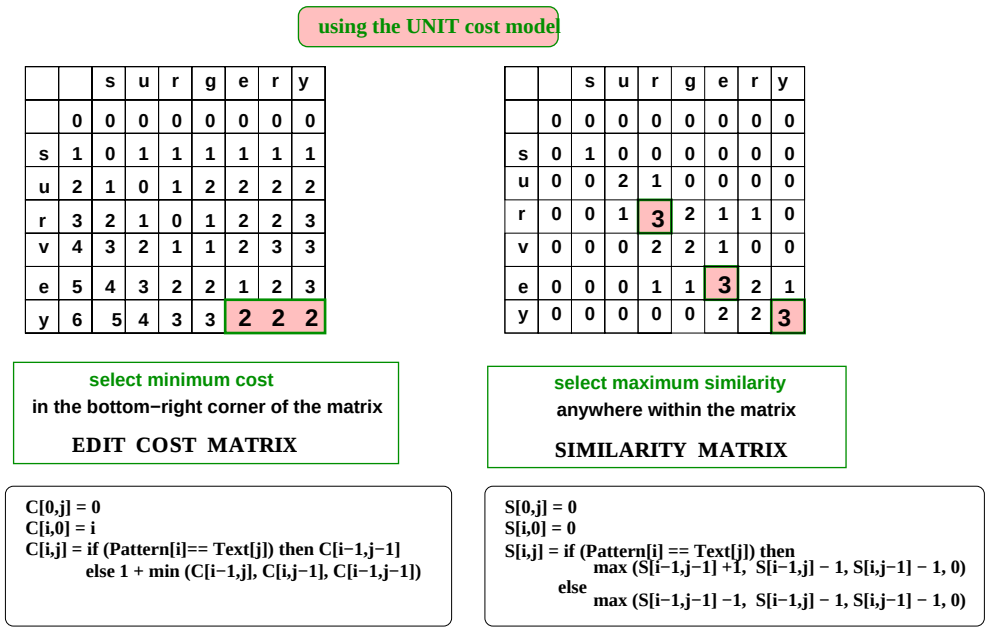


Figure 4.18: Edit cost and similarity matrices for the comparison of the pattern *survey* with the text *surgery*, using unit costs.

The similarity function, to be found for instance in Smith and Waterman ' [203], works in an analogous manner. For each pair of letters a similarity function Sim is defined, analogous to the edit cost function, and a gap cost d is also assumed. In Smith and Waterman for DNA Sim is $+4$ for a match, and -5 for a mismatch of any two letters. Gaps are scored as cost -5 for starting a gap and -2 for gap extension. Several models of gap costs are in use [94, 69]. We ignore the gap costs in our work. In a simplified model, with a gap cost d, the calculation of similarity is as follows:

```
DP[0,j] = 0
DP[i,0] = 0
DP[i,j] = max (0,
                Sim(char[i], char[j]) + DP[i-1,j-1],
                DP[i-1,j]+d,
                DP[i,j-1]+d).
```

In setting of initial similarity values to 0, we achieve a situation where partial matches can be computed, which correspond to local alignments. We can define a local alignment as any alignment of the pattern to a part of the text, where the similarity score exceeds a given threshold.

There are many possible ways of computing the matrix. A full row-by-row or column-by-column evaluation of all cells will have the time complexity of the product of both dimensions. Several optimisations have been developed. Some are based on traversing the diagonals, some on splitting the matrix into submatrices, and some on the observation that in a unit edit cost model, next value will differ by maximum of 1 [168]. Those optimisations lead to $O(kn)$ complexity, where k is the number of mismatches allowed. Some of them

are suitable for biological use, but some are not, in particular where more complex models of the cost of gaps in the alignment are required [94].

4.4.2 Automata for approximate matching

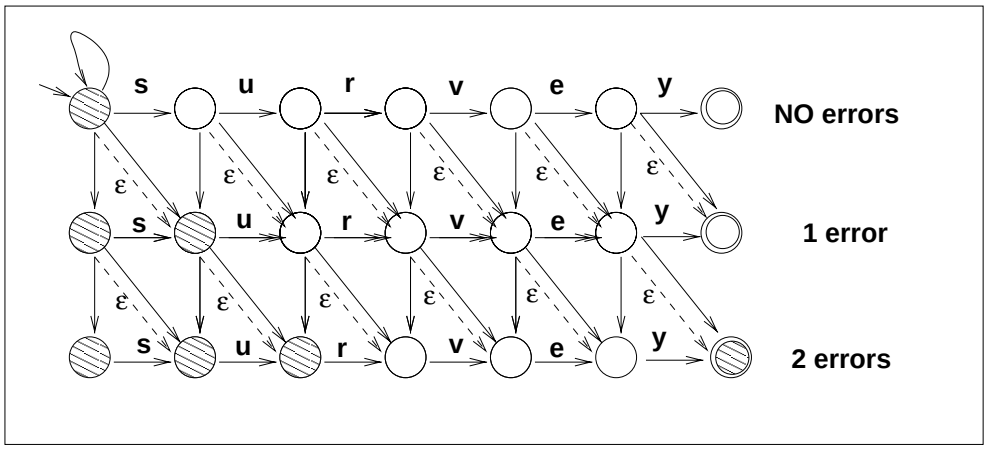


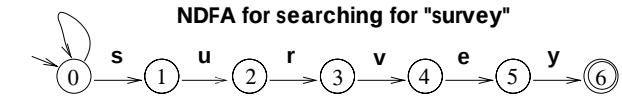
Figure 4.19: An automaton recognising the pattern *survey* with up to two errors. After reading the text *surgery*, the states which are active are shaded, one of them being a final state, reproduced from [168].

Non-deterministic finite automata (NDFAs) may be used to model the approximate matching problem. An example automaton which recognises the pattern *survey* with up to two errors is shown in Figure 4.19. In this automaton, each row denotes the number of errors seen (an automaton recognising low similarity would be large). Every column represents matching the pattern up to a given pattern position. As text is fed through this automaton, transitions are made, and, if a final state is reached, the text has a match with the pattern. Horizontal rows represent exact matching, vertical rows stand for insertion into the pattern, solid diagonal arrows represent replacements, and dashed diagonal arrows represent deletions in the pattern (ϵ transitions). There are several approaches to transforming this NDFA to a DFA (deterministic finite automaton) to enable a computational evaluation of the automaton states. A DFA will generally be very large, so alternative solutions based on bit-parallelism have been developed.

4.4.3 Bit-parallelism

Bit-parallelism can be used to represent each position in the matrix as a bit, or to parallelise the computation of the NDFA (without converting it to a DFA). The average complexity of such a computation remains in the region of $O(kn/w)$, where w is the word length, with $O(n)$ search time for short patterns. The algorithms based on bit arithmetic are limited to finding appropriate bit manipulation schemas which can be computed fast. In case of protein matching, designing such a scheme may well be impossible, and even with DNA it is not easy, as a general schema to cover different possible edit or similarity costs, as well as gap scores, would have to be devised. It would be a challenge to make such an implementation

parametric, to allow the necessary flexibility in the program execution, as is the case with BLAST [7].



BITMAP for each pattern character

with 1 set to signify the occurrence of a letter within the pattern

s	1	0	0	0	0	0
u	0	1	0	0	0	0
r	0	0	1	0	0	0
v	0	0	0	1	0	0
e	0	0	0	0	1	0
y	0	0	0	0	0	1

NEED m/w words where m is text length and w is word length

SEARCH state is kept in a machine word where a bit is set to 1 whenever the pattern up to that bit matches the end of the text read up to now (state for a particular pattern char is active). A match is reported whenever the first state is 1.

1. set D to 1^m

2. for each new text character update D using

$D' = ((D >> 1) \mid 10^{m-1}) \& BitMap[txtChar]$

Figure 4.20: Principles of the SHIFT-OR algorithm.

We show here the underlying principles of a bit-parallel SHIFT-OR automaton [28], see Figure 4.20. The automaton is based on bit arithmetic. For each distinct character in a pattern a bitmap is stored with one or more bits initialised to one. Ones are the positions where this character appears in the pattern. The text bits are initialised all to 1 (one bit per text letter). Text characters are provided as input to the initial array of ones, and appropriate bit arithmetic is carried out, as shown. This results in matches whenever the first bit of the text bitmap D is set to one. This solution to exact matching is extended to approximate matching by grouping characters into classes, and each class having one bitmap vector used to AND with the outcome of the OR comparison. Some limitations of this approach arise when the text is very long, which is the case with biological texts. The answer here is to break the text into manageable chunks which can be compared using bit-parallelism efficiently.

4.4.4 Filtering

The filtering approach relies on selecting parts of text which look similar to the pattern, and then carrying out the dynamic programming matrix computation on those parts of the text. For low error ratios filtering works well. Since it is based on the fact that some portions of

the pattern must appear with no errors in the text, matches with lower similarity will not be reported.

This approach is the same as in the q-grams, reported in a previous section. There are several variants of the filtering approach. Some of the work is based on the fact that if the pattern is split into $k + 1$ pieces (in the k errors approach), any approximate occurrence must contain at least one of the pieces with no error, since k errors cannot alter the $k + 1$ pieces. This leads to different variants of hierarchical searching, where exact hits are found and then extended with neighbouring strings to get longer approximate matches. BLAST and FASTA [7, 176] use different versions of the filtering approach, based on starting with exact matches and extending them to include areas of high string similarity.

4.5 Closing

This chapter presented a selection of most important theoretical foundations underlying our work. Those include orthogonal persistence, sequence indexing data structures, and matching algorithms. We did not cover the details of approximate searching using suffix trees. This work is presented in next chapter which is devoted to the experimental work with very large suffix trees.

Chapter 5

Experimental work with data structures and exact matching

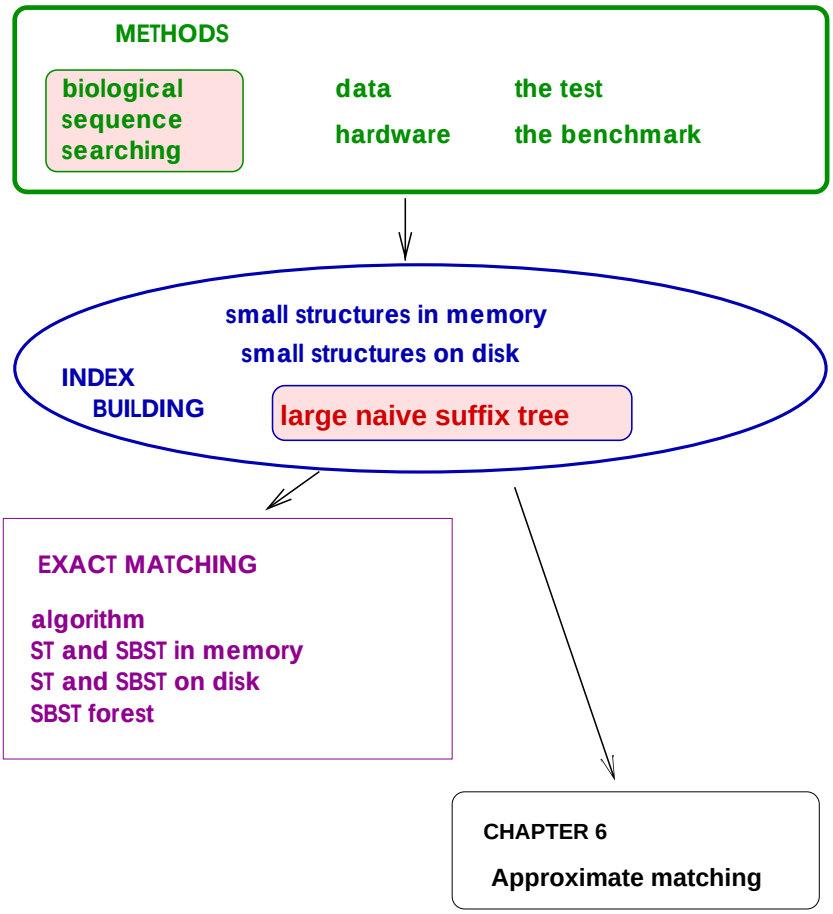


Figure 5.1: Overview of Chapter 5.

This chapter presents one of the main contributions of our research — the construction of very large naive suffix trees, described in Section 5.4. A graphical overview of this chapter

is in Figure 5.1. We start by introducing our materials - the specific features of biological sequence analysis, the data, hardware and data structures studied. We then discuss the creation of index structures both in memory and on disk, including the construction of very large suffix trees. Subsequently, we provide an overview of exact matching, and thus prepare the ground for Chapter 6 which deals with approximate matching using a large suffix tree.

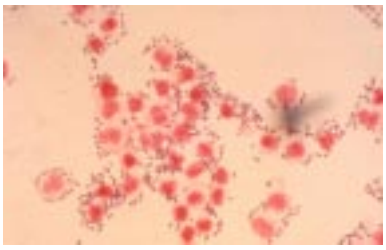
5.1 Methods and materials

Our work aims to speed up large-scale sequence comparison tasks carried out by biologists. Our methodology combines an examination of the current tools in use, with experimentation based on algorithmic techniques and persistence technologies. This combination of algorithmic techniques with persistence is novel, and allows us to extend indexing techniques to encompass new data types.

We worked with 3 groups of life-science researchers to understand the scope and requirements of sequence comparison in their work, and to focus our testing methods on the relevant data and testing scenarios. It became clear that a different focus of biological research makes different demands on the computing resources, and a variety of searching approaches may be needed within one project. We face the situation where sequence analysis tasks require significant amounts of computing power, often surpassing our computing resources. This provides additional motivation for our research into faster string searching. We briefly introduce the three different sets of requirements we studied, and possible solutions which could be applied in those contexts.

5.1.1 Possible biological tests

Bacterial genetics



With the appearance of many fully sequenced genomes, the possibility of subtractive genome analysis arises. The idea is to compare two related genomes, one of them the focus of research and the other closely related, in order to identify the genes responsible for differences between the two. This type of activity was discussed with Professor Tim Mitchell. Our initial work was in providing a global view of sequence similarities between *Streptococcus pneumoniae*[1], and *Streptococcus pyogenes*. We supervised the research of an MRes student [87] who used a variety of computing tools to generate a comparison of genes in the two genomes. Further work could involve a comparison with *Lactobacillus lactis* or other related bacteria, with 3-way gene comparisons or even more complex scenarios. Bacterial genes would be compared using BLAST, or entire genomes using MUMmer [66]. Because of the relatively small size of bacterial genomes, this type of analysis is possible using our computing resources (described in Section 5.1.4), but the management and visualisation of the data sets and possible analyses require additional software support. A data and program execution management and automation system would be useful in this context [206]. Such a system would require a combination of a database and a powerful map viewer, as a minimum.

[1] image reproduced from http://www.meddean.luc.edu/lumen/DeptWebs/microbio/med/gram/slides.htm

Protozoan genetics

We investigated the use of BLAST in collaboration with Doctor Lorenza Putignani who researches possible gene transfer from bacteria to a parasite *Cryptosporidium parvum*. The hypothesis under investigation is that *Cryptosporidium* contains parts of smaller bacterial genomes. The full genome of this organism is not available yet, but small contigs totalling some 7.7 Mbp of DNA sequence have been released. BLAST is the algorithm of choice, as it is capable of comparing a collection of contigs with a complete genome[2]. However, performing BLAST against a full bacterial genome consumes significant amounts of CPU. On our hardware configuration, see Section 5.1.4, a full BLAST run for the comparison of 7.7 Mbp of *Cryptosporidium* contigs against a database of *Rickettsia prowazekii* totalling 1.2 Mbp took in excess of 6 days and produced over 2 GB of results data which after post-processing with MSPcrunch [205] produced 50 high scoring hits which we attempted to visualise using ACT [15, 190]. As it turns out this visualisation is not satisfactory, due to the fact that *Cryptosporidium* contigs are not yet assembled. As further comparisons with other genomes are required (some 10 other genomes will be involved), the management of computation and data requires techniques which call for a complex data analysis and computation management system. Such a system would consist of a database holding the genomes, results of genome comparisons, genomic maps, and annotations of gene functions. It would also have to organise the computation involved in sequence comparison and provide data analysis and visualisation support for a rich knowledge domain. Additionally, it would have to be capable of integrating external data sources from several distributed databases.

Mammalian genetics

We identified a problem which we could not solve using our computing resources. Professor Keith Johnson's research group investigating a mouse with a genetic abnormality (mutation producing a phenotype reminiscent of human multiple sclerosis) wanted to map the mouse gene responsible for this phenotypic variation. The human sequence is now almost complete and there is sufficient mouse sequence (available as sequence fragments) to possibly find the gene. Because of the special breeding strategy used with the laboratory mouse[3], the area of interest was narrowed down to 2 Mbp. Since the approximate map for the human genome in that area is known, it should be possible to build a map for the equivalent area in mouse, based on gene similarity between species. The only way to perform such a comparison, without having access to an assembled mouse genome, would be to perform sequence comparison of all available mouse data (now over 17 million sequence fragments) against the 2 MB of the human sequence. We realised that we did not have adequate resources to support this investigation, as it would require computing power and data management which lie outside the scope of our research.

Defining the test

None of the investigations presented above furnish a test case which could be easily interpreted as a benchmark of our technology. BLAST and other sequence comparison packages

[2]MUMmer [66] could be used as well [226] but is less suited to the analysis of genome fragments.

[3]Image reproduced from http://members.aol.com/MusBeMice/LabMouse.jpg.

produce data which can only be evaluated using expert biological knowledge. We therefore propose a different benchmark which can be interpreted independently of the biological context. We will use the same dynamic programming matrix calculation for the same set of queries, in two contexts, without an index and with an index. Moreover, as well as reporting the actual time required to execute a query, we will show the number of columns of the DP matrix calculated during the partial scan of the tree. This will measure the gain from using an index rather than executing a full matrix calculation and will study the interplay of the data set and the parameters of our approximate matching algorithm. While designing a benchmark, we need to keep in mind the main requirements of biological sequence searching, and we introduce those now.

5.1.2 Biological sequence analysis

This introduction will not cover the field in depth, but concentrate on the salient features of this kind of pattern matching, in particular on the difference between the idealised view of pattern matching, as embodied in most theoretical work, and the implementation of matching in the three commonly used biological sequence search algorithms: the Smith-Waterman algorithm [203], FASTA [176] and BLAST [7, 8]. For a full account of this field of research the reader may want to consult any of the following texts [230, 99, 69, 162]. We only attempt to distill the commonalities between the three most commonly used packages in order to decide how flexible our implementation has to be, so that in the end it might lead to a valid implementation of biological sequence matching.

We now list the main features of biological sequence searching which make this field distinct.

- The use of the similarity matrix. As illustrated in Chapter 3, biological sequence searching uses the similarity matrix, not the edit distance matrix, in most cases. The edit distance matrix is used in the construction of phylogenetic trees [162]. We did not explore phylogeny construction as it requires multiple sequence alignment which is outside the scope of our work, and is performed on small data sets.

- The use of non-unit costs. For DNA, for instance, BLAST currently scores every match as $+1$, and mismatch as -3 (Stephen Altschul, personal communication), but those defaults can be changed on demand. Smith-Waterman algorithm for DNA alignment was first designed with a match DNA score of $+4$ and mismatch score of -5. For protein scoring, a whole variety of approaches are used, we present one of the scoring matrices, BLOSUM62 [104], in Table 5.1. Protein scoring matrices are based on the examination of alignments of related proteins and they reflect the likelihood of substitution of one amino acid by another.

- The use of gap costs. A variety of approaches exist [69]. For instance in DNA BLAST gap existence is scored as -5 and gap extension as -2, so that a gap of length k costs $-(5 + 2k)$.

- Use of statistical measures of the significance of a match. The scoring system used in BLAST relies on the use of statistics which are relative to the size of the database used for querying. The interpretation of results returned by BLAST is subjective, and biologists use their intuitive understanding to evaluate different results returned by BLAST. Automated approaches to the problem of filtering of the results are also

known [205] and they choose best hits based on similarity scores delivered by BLAST. Miller [158], however, calls for sequence analysis software which has a more rigorous statistical foundation (different form BLAST).

- The size of text and pattern under consideration. It is now common to ask for genomic comparisons, for instance a comparison of all human genes (between 25,000 and 40,000 sequences of differing lengths, some of which exceed 1,000 amino acids (AAs)) against all proteins (currently 200 Mb). For DNA comparisons the query sequence is longer than for proteins (even 100 Kbp) and the underlying text can be as large as several Gbp.

These requirements diverge significantly from the idealised view of pattern matching with k differences, as embodied in most computing science theoretical research. Solutions presented by the available theory address idealised problems, and further work is required to turn the "idealised" scenarios into efficient search tools.

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

Table 5.1: The Blosum62 matrix.

The special features of the problem we are addressing present us with a picture of great complexity. We currently abstract from the statistical dimension, and are planning to develop an understanding of this area in the near future. We also abstract from the fact that protein and DNA similarities can be evaluated in different ways, and alternative sets of scoring matrices as well as alternative ways of scoring gaps in sequence alignments have to be accommodated. Therefore, we limit our interest to methods which can be used with arbitrary scoring schemes [187]. This precludes the use of bit-based DP matrix evaluation techniques for instance, and forces us to focus on the simplest methods initially, that is the execution of a DP matrix calculation on top of a suffix tree.

5.1.3 Data sources

We had a discussion about our work on suffix trees with a European Bioinformatics Institute researcher (Jaak Vilo, personal communication, April 2001) in whose experience searches on proteins are currently central to most biological research, and protein searching constitutes most of the load for sequence similarity searching. We decided to focus on proteins from the SWISSPROT resource[4], and the predicted gene sequences (proteins) from the Ensembl web site[5]. In the research leading up to final tests we also used other data sets which we describe next.

Caenorhabditis elegans

This data set is available from the Sanger Centre[6], and consists of 6 chromosome sequences of total length 97 Mbp. Chromosomes range from 12.4 to 20.5 Mbp. As queries we used cDNA sequences from the same web site, which have now been replaced with another file[7]. General information on *C. elegans* (see picture) and a bibliography are available on the web[8].

Ensembl dataset and the human genome

The Ensembl dataset[9] contains up-to-date human DNA sequence and mapping data, including predicted genes. All data produced by the International Human Genome Sequencing Consortium are published and available via this website. We imported genetic sequences for chromosomes 1, 21 and 22, totalling some 300 Mbp (300 MB) and the gene set of some 26,000 genes, totalling 34 MB. Genes are available from ftp://ftp.ensembl.org/current/data/fasta/dna/, and chromosomes are available from the US National Centre for Biotechnology Information (NCBI) as merged files at ftp://ncbi.nlm.nih.gov/genomes/H_sapiens/CHR_01/hs_chr1.fa.gz and in the other chromosome directories. The Sanger Centre FTP site used by Ensembl does not offer merged chromosome data, but presents lists of individual clones instead.

Plant and insect data

We also used DNA sequences of plant origin, *Arabidopsis thaliana* (fragments totalling 253 Mbp)[10], and the fruit fly sequence of *Drosophila melanogaster* (130 Mbp)[11]. We used these data sets to carry out an initial investigation of the shape and size of the suffix trees we were producing.

[4] http://www.expasy.org
[5] http://www.ensembl.org
[6] ftp://ftp.sanger.ac.uk/pub/C.elegans_sequences/CHROMOSOMES/CURRENT_RELEASE/
[7] ftp://ftp.sanger.ac.uk/pub/C.elegans_sequences/ESTS/
[8] http://elegans.swmed.edu/, http://www.biotech.missouri.edu/Dauer-World/Wormintro.html
[9] http://www.ensembl.org/
[10] ftp://ftp.tigr.org/pub/data/a_thaliana/ath1/SEQUENCES/
[11] ftp://ftp.fruitfly.org/pub/genomic/fasta/na_arms.dros.RELEASE2.Z

Protein data set

Protein sequence data consist of three files available from the SWISS-PROT FTP server[12]. All known proteins are referred to as SWISS-PROT, and stored under the name sprot.fas.Z. Predicted proteins form the TREMBL dataset available as trembl.fas.Z, and predicted proteins which are awaiting classification are in the file called trembl_new.fas.Z. Total volume of AA sequence in the three files is 200 Mb.

Data acquisition

Data were retrieved using FTP. Genetic sequences which were to be indexed were preprocessed, using a Perl [228] script to remove headers, carriage returns and any symbols outside the DNA and protein alphabets. Individual sequences were concatenated using a single ***** symbol.

5.1.4 Computing methods

Research Methodology

A new research methodology, combining algorithmic and persistence techniques, was developed and successfully applied to solve the problem of the size limitation which characterised the construction of disk-resident suffix trees. The underlying theoretical foundations were discussed in Chapter 4.

Hardware and software platform

Tests were run under the Solaris operating system version 5.7, on Sun Enterprise 450 servers, with 2 GB RAM. Transient tree tests were initially run using Production Java for Solaris, i.e. Java 1.2 with JIT using options modifying the GC behaviour [178], and subsequently with Java 1.3 (the so-called Java 2). Initial versions of persistent suffix tree were tested in 1999 with PJama based on Java 1.1. This version of PJama software was still too unreliable to support consistent experimentation with large data sets. PJama 1.6.5, based on Java 1.2 became available in Spring 2000, and allowed for the execution of a substantial body of tests. Further improvements to the logging component of PJama in the summer of 2000 [155] allowed for longer running transactions, and led to the creation of large persistent data structures.

Initial tests

The tests we carried out reflected the gradual growth of our understanding of suffix trees and the research into tree structures and tree creation algorithms. The tests stretched the technology, and we used the data structures of the maximum size possible within the limits of hardware and software that we were using.

An initial version of *ST* code was developed in the autumn of 1999, based on code translation from Ada to Java by Sarah Cox in the summer of 1999 [60]. This code was optimised and re-written – partly to accommodate the inefficiencies of stack management in Java 1.1 (an explicit stack had to be used in tree traversal). Our first large-scale testing

[12] ftp://ftp.expasy.org/databases/sp_tr_nrdb/fasta/

was carried out in April 2000. These tests investigated transient suffix trees for up to 26 Mbp of DNA, and persistent suffix trees for up to 15 Mbp.

The *SBST* code was translated from Ada to Java in the summer of 2000 by Brian Young [237], and after cosmetic changes was used in a large set of tests in September 2000. These tests compared suffix trees and suffix binary search trees. Persistent suffix trees and suffix binary search trees for up to 20.5 Mbp of DNA were built. Additionally, a forest of *SBSTs* containing all of *C. elegans* genome (97 Mbp) was built and tested.

Tests supporting the thesis statement

In January 2001, after work with different versions of the suffix tree, we discovered how to build trees exceeding the size of RAM. We carried out tests in tree building and exact querying for a DNA dataset of 263 Mbp, corresponding to the sum of human chromosomes 1, 21 and 22 as available at that time. A paper describing this work was accepted for publication at the VLDB 2001 conference [119], and we were invited to extend this paper for inclusion in the VLDB Journal.

Further tests with protein trees for 200 Mb (SWISSPROT) and tests with approximate matching algorithms followed. We also tested BLAST [7, 8] and SIM4 [84] to gain experience with the data and package interfaces. Our tests made it extremely clear that sequence analysis is very CPU intensive, and we currently do not have resources to perform it on a large scale. We also experimented with the post-processing of BLAST results using MSPcrunch [205], and our own tools to post process BLAST and SIM4 results in order to filter out a subset of relevant matches. We then decided to test our hypothesis in a way which objectively tests the tradeoff between indexed and unindexed searches. We performed such tests for protein data, and later for DNA data sets. We used the same dynamic programming matrix calculation both in the context of the persistent suffix tree index and without an index. The paper summarising all tests to date has now been submitted to VLDB Journal and is reproduced in Appendix C.

BLAST as a benchmark

Sequence comparison benchmarking was initially performed using BLAST [7, 8]. The results we gathered are presented in Chapter 6, as a background and an indicator of the relative speed of this method. It is beyond the scope of our work to give justice to the field of biological sequence analysis, as most literature refers to a wide range of statistical measures which we did not research. We provide a bird's eye view of BLAST in Chapter 6, where we also introduce our own benchmark which directly measures the gains from using the indexing technology.

5.2 Building of suffix index structures

$O(n)$ suffix tree construction was researched by Weiner [232], McCreight [153], and Ukkonen [224]. Those algorithms owe their optimal theoretical construction time to an additional factor in the space domain – the existence of suffix links which traverse the tree in the direction perpendicular to the direction of child-parent links. We do not cover those construction algorithms here because they are "quite complex and only of theoretical interest" [27]. Gusfield provides a high-level view of those methods [99], but for full detail the original

papers are indispensable. Giegerich and Kurtz researched the equivalence between those algorithms [89], and most of the theoretical work uses those time-optimal algorithms.

When the size of the tree is an issue, i.e. the tree does not fit into RAM, and has to be constructed on disk, the additional space required for the suffix links, and the fact that they traverse the tree "horizontally", makes the time-optimal algorithms underperform. This has been observed by Baeza-Yates and Navarro [27, 25], and they concluded that suffix trees larger than RAM cannot be constructed. We noted the same behaviour in our tests, and we turned to naive suffix trees which allowed us to overcome this limitation. In this section we sketch the timing tests performed using the original Ukkonen's algorithm first, then a modified Ukkonen's tree, and finally the naive tree of our construction. We contrast those trees with the *SBST* and then show how these trees are built in memory and on disk.

5.2.1 Ukkonen's suffix tree - original version

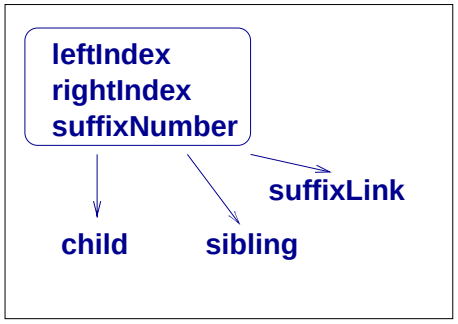


Figure 5.2: Node layout as implemented in [60].

Code translation from Ada to Java for the Ukkonen's algorithm was performed by Sarah Cox, based on code provided by Rob Irving who co-supervised the project [60]. The layout of a tree node in this implementation is as follows. A $Node$ consists of a $child$ reference, a $sibling$ reference, a $suffix\ link$ reference, a $left\ index$ into the text string, a $right\ index$ into the text string, and the $suffix\ number$, see Figure 5.2. The first three fields were implemented as object references, and the last three as integers.

One of the features of Ukkonen's algorithm is that a so-called "implicit tree" is built first, in which the right index into the text in all leaf nodes is populated with an arbitrary large value, and the suffix number is left unassigned. In a subsequent tree traversal two modifications to leaf nodes take place. First, the right text pointer is replaced with the index of the end of the indexed string (total string length). Secondly, each leaf is provided with the suffix number to which it corresponds. The resulting tree is called an "explicit tree". In the case of a tree already on disk, making the tree explicit requires a full traversal of the entire data structure, and two updates in every leaf. We observed that this was a superfluous operation, and that the size of the string could be held as a global variable for the entire index, and initialised just once, while the suffix number could be calculated at query time. This would remove the need for a traversal changing an "implicit" into an "explicit" tree. This observation led us to the development of an improved "leaner" tree using Ukkonen's algorithm, described below.

5.2.2 Leaner Ukkonen's suffix tree

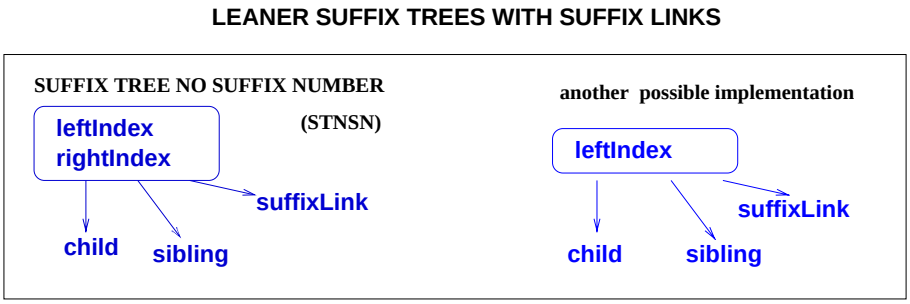


Figure 5.3: Two possible node layouts for a leaner tree with suffix links.

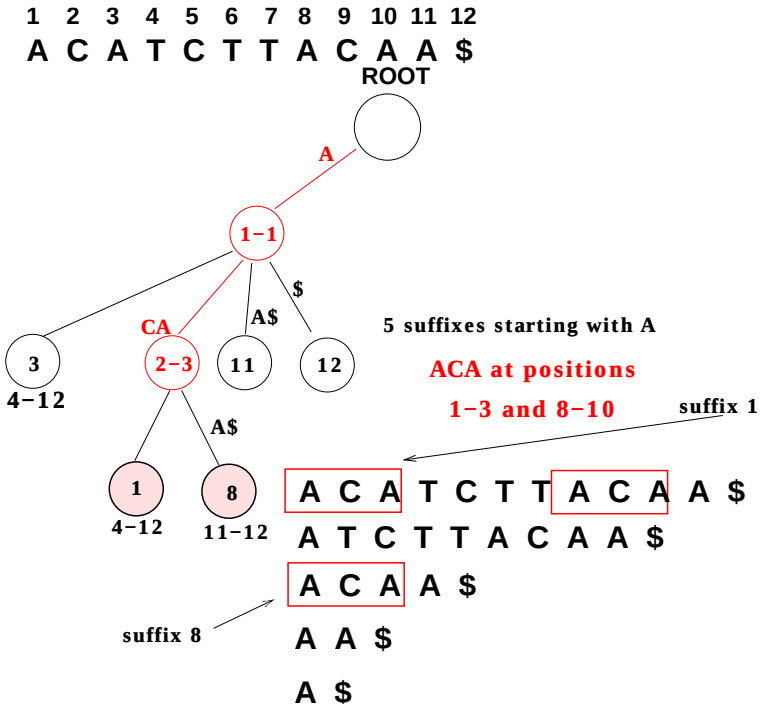


Figure 5.4: Part of a suffix tree illustrating two suffixes sharing the same prefix *ACA*, and the 2 leaves (shaded).

Several alternative implementations of lean trees with suffix links are feasible, and we implemented one lean tree version, shown on the left hand side of Figure 5.3. Other alternatives were explored by Malcolm Atkinson's students [128, 186]. After producing the naive suffix tree, see next section, we realised that the right hand variant is also possible, but we did not implement it or carry out any tests with this version, since the naive tree allowed us to build trees on disk more reliably and for larger data sets. Further analysis of the naive tree in this chapter will show why the naive tree is our preferred data structure. Removing the right index into the text from all leaves and using a global value instead is an obvious

modification[13]. Removing the right index from internal nodes is done as follows. Look up the left index in the child node and subtract 1. We now explain why the suffix number can be removed from the leaf. The suffix number represents the starting point of a particular suffix in the tree. Suffixes sharing the same prefix will share a part of the path from the root down to a point where they differ and branch into different subtrees. Let us look at Figure 5.4 which shows part of a suffix tree, together with the indexed string. *ACA* is a prefix of two suffixes, and is represented in the tree by its first occurrence from the left, that is $S_{1..3}$ (represented by two nodes). We calculate the actual suffix number on reaching the leaf, and we use the string length of the path leading towards the node. The formula used here is

$$LeafNumber = leftIndex - pathLength.$$

If we were looking for *ACA* in the tree, we scan 2 nodes, and have traversed 3 characters so far (the length of the query). Then the two shaded leaves below which have left index values of 4 and 11 have suffix numbers $4 - 3 = 1$ and $11 - 3 = 8$.

5.2.3 Naive tree

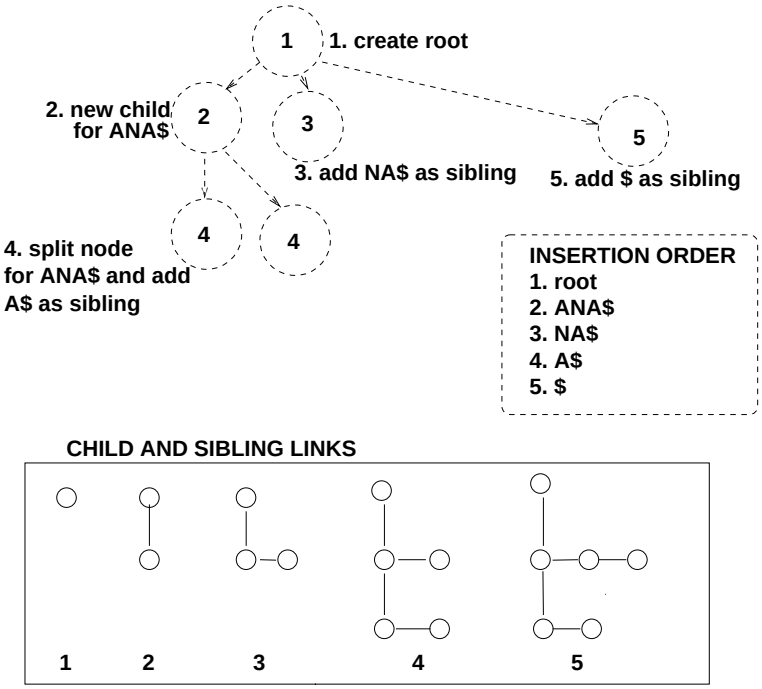


Figure 5.5: Tree creation for *ANA$*

We developed the naive tree in order to investigate possible space economies of a suffix tree. Since the algorithm to create a naive tree is easy to understand, it could also be easily used with alternative storage schemes. As it turns out, because of the lower space requirement of a naive tree and the lack of suffix links (and hence better locality of access, resulting in

[13]The right index is needed in querying so that we never try to go past the end of the substring indexed by a node.

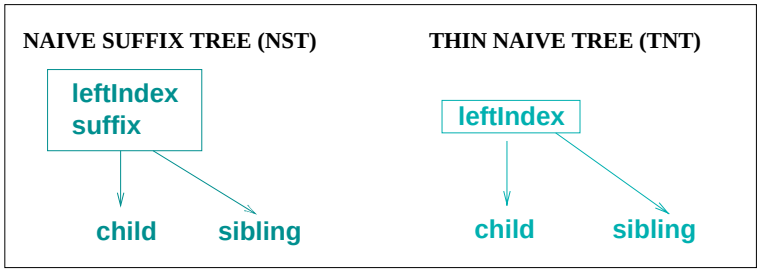


Figure 5.6: Alternative node structures for the naive suffix tree.

fewer cache misses), this tree showed good characteristics in practice, which subsequently allowed us to build much larger suffix trees, in excess of RAM size.

The process of building a suffix tree may be carried out starting from the first suffix, or from the last one. We arbitrarily selected the first method. First the root is created, then the node for the first suffix is added, and then remaining suffixes are inserted one-by-one by comparing the string indexed by the new suffix with the suffixes already in the tree. This way of creating a tree uses 2 operations: adding a new child, if a child starting with the current character does not exist yet, and splitting an existing node to accommodate a new suffix whose prefix diverges from an existing suffix. This is illustrated in Figure 5.5 on the example creation of a tree indexing string *ANA$*, where on the insertion of suffix *A$*, operation number 4, the node coding for *ANA$* has to be split into two nodes, one coding for *A*, and the other for *NA$*.

In implementing the naive tree, we used two data structures for a node, shown in Figure 5.6. The first structure (NST) has two object references ($child$ and $sibling$), a $left\ index$ into the string and the $suffix\ number$. Nodes which are to be inserted into the tree are created with the left index and suffix number initialised to the suffix number value, and as their appropriate location in the tree is being found, the left index is incremented. In this tree version we do not use the right pointer into the text. Instead, we look up the right pointer in the child node,

$$rightIndex = node.child.leftIndex - 1,$$

as explained in Section 1.2.2. This has an additional processing cost (a node lookup).

In the thin naive tree (TNT), we do not record the suffix number. A node consists of just three fields: $child$ reference, $sibling$ reference, and the $left\ index$ into the text. The right index is calculated as above, and the suffix number is calculated during the query, as already explained. Further optimisations which remove null sibling or child links were explored by Japp [128] and Riley [186] but were not considered in our work.

The worst-case complexity of naive tree building is $O(n^2)$. However, as DNA data and proteins have an almost random distribution, the average building time is of the order of $O(nlogn)$ [212][14]. Experimental behaviour of this tree is surprisingly good, see the following sections.

5.2.4 Suffix Binary Search Tree

The suffix binary search tree *SBST* was developed by Rob Irving and Lorna Love [125, 126]. The Ada code was translated into Java by Brian Young in the summer of 2000 [237]. The

[14]The actual formula used by Szpankowski is more complex, and relates to data entropy.

original code executed under Windows created a tree for 1.6 Mbp in less than 2 minutes, and the same version run under Solaris needed around 20 minutes. Structural changes were needed to improve performance under Solaris, mainly in the class and method interfaces. We experimented only with one version of the *SBST*, and the data structure used is shown in Figure 5.7. This shows a tree node consisting of two object references (one per child) and three values called $maxlcp$ (maximal common prefix, integer), $direction$ (Boolean) and $suffixNumber$ (integer) which we explained in Chapter 4. No further code optimisation was undertaken.

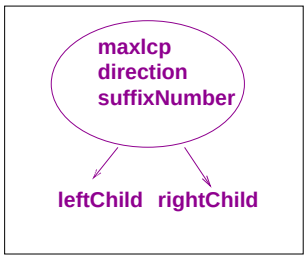


Figure 5.7: An *SBST* node.

Suffix binary search tree is a potentially interesting data structure. Because there is only one node per text character indexed, the total memory occupancy will be 2 integers + 2 references + 1 bit per character indexed. In comparison, the lean naive tree requires 3 integers per node, and each text character may give rise to up to 2 nodes (observed values for DNA and protein are in the region of 1.6 to 1.8 nodes per character indexed). As a result, the *SBST* requires less memory than a suffix tree. Recent research [126] shows that an *SBST* can be used to build a suffix array. Suffix arrays have even lower space complexity than the *SBST*, so experimental research in that direction is valuable. Baeza-Yates and Navarro [25] have researched simple approximate matching scenarios using disk-resident suffix arrays for up to 10 MB of source data, as well as optimisations of suffix arrays designed for use on disk. It would be interesting to extend their results to biological searching with larger suffix arrays, using cost functions which are relevant in biology.

5.2.5 Tree building in memory

We present now a suite of test results which were re-run recently to provide a most up-to-date comparison of the alternative structures. We used the following three data sets: *C. elegans* merged chromosomes, merged proteins from TREMBL and SWISS-PROT, and merged human DNA from chromosomes 1, 21 and 22. Tests were carried out using Java 1.3 with the following settings:

- $-server$, which indicates the VM suitable for server operation (client VM is the other option),
- $-Xms1000m$, which is the initial size of the Java heap, and
- $-Xmx1900m$ to set the maximum Java heap size.

Tests were run once for each data set, with the following sizes of input: $1Mb$, $5Mb$, $10Mb$, and then increasing by $5Mb$ up to the maximum size possible for a given data structure. For larger input data sets some runs resulted in the "out of Memory" error. For the suffix tree with suffix links the computation was not progressing and not generating an error message either, and the Java process had to be terminated after 60 minutes. Creation times were measured for the data structures presented in Figure 5.8. We measured the time to create a tree as follows. Data were read in as an array of bytes. Then the time was recorded using *System.currentTimeMillis()* method, the tree was built, and then another time stamp was taken, and the time difference calculated. The computer was not used for any other tasks,

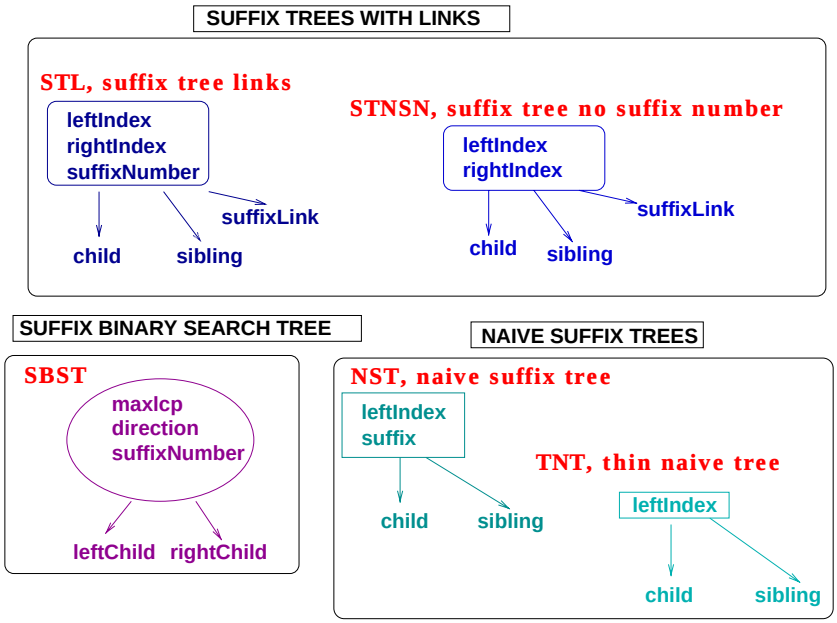


Figure 5.8: Transient indexes built in memory.

and only one processor was in use. We now present the measurement results, first ordered by index type, and then comparing different indexes.

Suffix Tree with Suffix Links (STL) creation

Data size Mb	PROTEIN	C. ELEGANS	HUMAN
1	13218	10522	10526
5	97035	70051	70505
10	254109	199419	198823
15	447873	349126	348070
20	766099	623841	630412
25	999563	815635	816330
26	1052686	897170	1075321
27	1106184	1234593	1201089
28	1152764	2519606	3203641
29	1199547		

Table 5.2: Tree creation (milliseconds) for the O(n) tree, version STL.

A summary of tree creation times of the suffix tree with suffix links (version STL) is shown in Table 5.2, page 88, and in Figure 5.9, see page 89. We make three observations here. The protein tree takes slightly longer to build than a DNA index. This is probably due to long linked lists representing siblings. As the protein alphabet is larger than the DNA alphabet, the lists will be longer. However, there is also another difference, namely that the protein tree has fewer nodes, as trees for larger data sets than for DNA are possible for protein. Thirdly, we observe here a sharp rise in the time to produce an index for trees approaching

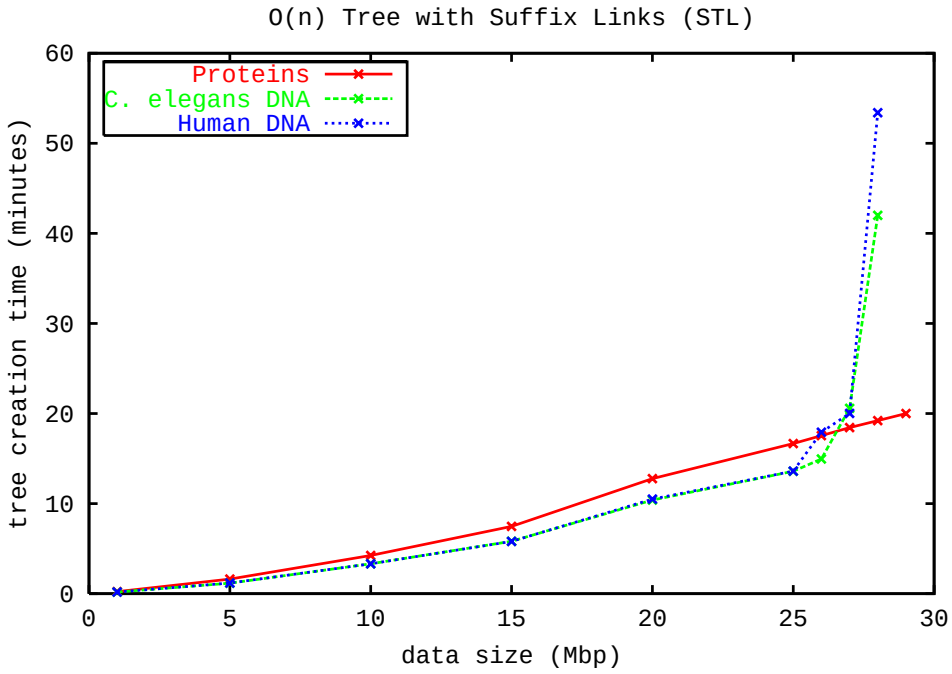


Figure 5.9: Time required to build an STL index for three data sets.

the size of the available memory (as expected). This indicates that swapping takes place, and program behaviour is hard to predict.

Suffix Tree no Suffix Number (STNSN) creation

Data size Mb	PROTEIN	C. ELEGANS	HUMAN
1	12117	9608	9378
5	87376	64746	64268
10	231932	177814	178456
15	402261	319012	305697
20	712535	568620	566887
25	925553	726142	721583
30	1410518		

Table 5.3: Tree creation (milliseconds) for the O(n) tree, version STNSN, with suffix links but no suffix number.

We present a summary of the tree creation experiment for the improved version of the $O(n)$ tree (suffix tree no suffix number, STNSN). This tree version is leaner (no explicit suffix field in the node) and it requires no tree traversal to make the tree "explicit", as described previously. Observed tree creation times are shown in Table 5.3, page 89, and summarised in Figure 5.10, page 90. Because of the lower space complexity, and lack of the final tree traversal, this tree version requires less time to build than the original ST implementation[15].

[15]for instance indexing 20 Mb of human DNA using the STL takes 630 seconds, and using the STNSN takes

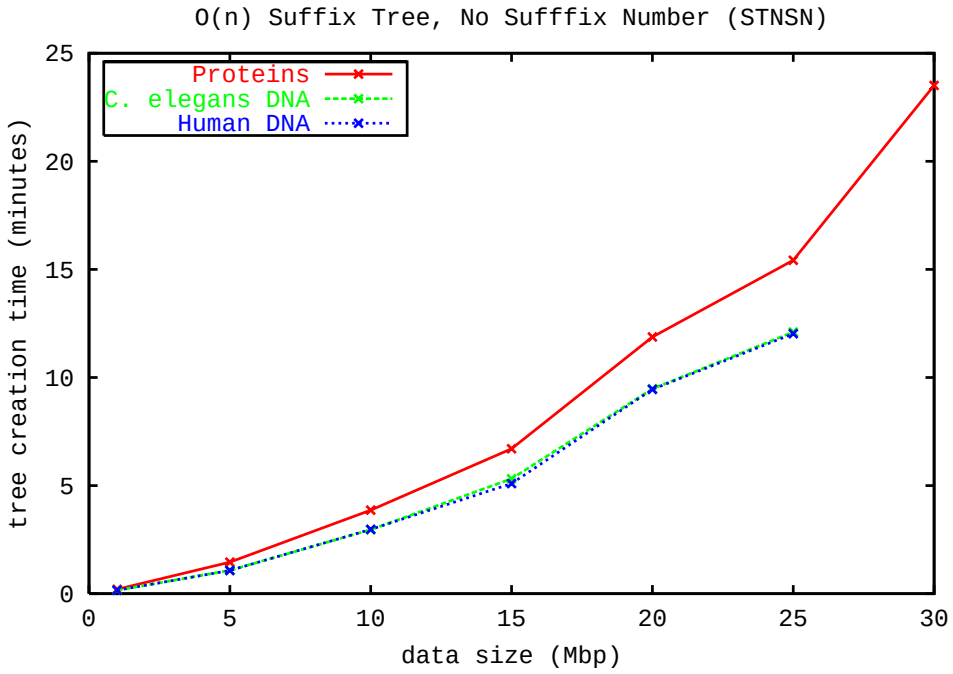


Figure 5.10: Time required to build an STNSN index for three data sets.

As previously noticed, a protein tree takes longer to build, but a larger tree is possible than with DNA datasets. The tree creation time seems to be influenced by the alphabet size, and the largest protein tree diverges from the expected linear performance, probably due to memory swapping operations needed for a tree which reaches the limit of available memory.

Naive Suffix Tree with Suffix Number (NST)

The time required to build this tree is presented in Table 5.4, page 90 and in Figure 5.11, on page 91. We observe that larger tree sizes are possible than with the original suffix tree, and

567 seconds.

Data size Mb	PROTEIN	C. ELEGANS	HUMAN
1	17101	11803	11661
5	131377	89241	80087
10	319035	194674	190593
15	559151	332667	332039
20	802824	482517	486011
25	1060027	649744	653457
30	1434817	939714	932250
35	1737997	1093467	1087255
40	2023708	1295463	1278753
45	2357526	1757964	1792716

Table 5.4: Tree creation (milliseconds) for NST.

that the protein tree takes longer to build.

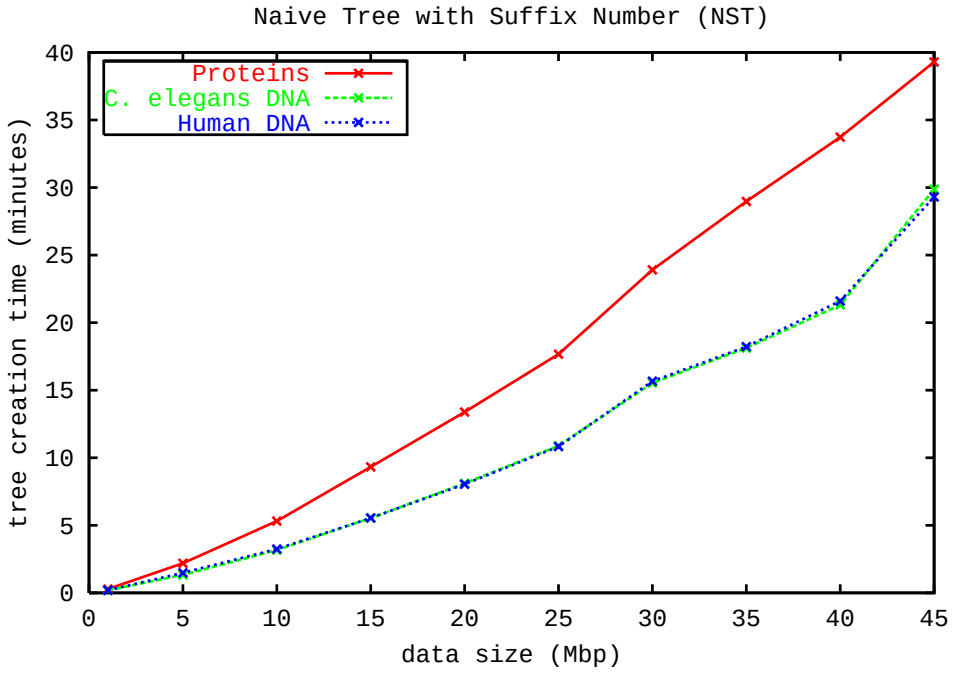


Figure 5.11: Time required to build an NST index.

Thin Naive Tree (TNT)

The time required to build this tree is presented in Table 5.5, page 92 and in Figure 5.12, page 93. DNA trees for *C. elegans* and human data show similar characteristics while the protein tree is more compact (larger tree can be built within the available memory) and takes longer to build.

SBST creation

The time required to build this tree is presented in Table 5.6, see page 92 and in Figure 5.13, page 94. The code for SBST building has not been optimised, and we used the simple version of the *SBST* algorithm[16]. We observe here the same shape of the graph and the relationship between protein and DNA trees as for suffix trees. The irregular shape of the plot for *C. elegans* DNA was probably caused by some network or disk activity which subsided subsequently. We are currently unable to explain adequately why the protein tree takes longer to build than the DNA tree. For a suffix tree this difference is probably due to the length of linked lists which connect the sibling nodes. To explore the issue of the tree creation time for the *SBST* we would have to measure the number of nodes visited, and

[16]A more elaborate *SBST* implementation in Java which requires additional storage is also available, and this version allows for faster tree creation in Ada than for the simple version of the tree (Rob Irving, personal communication). We did not test the faster algorithm because we wanted to reduce the tree size. It is also possible that a performance improvement in tree building could be achieved after code re-engineering.

Data size Mb	PROTEIN	C. ELEGANS	HUMAN
1	16889	11907	11707
5	133474	89629	81021
10	318229	194382	188809
15	554520	333834	333292
20	807349	487833	489799
25	1059601	655399	649900
30	1440385	929096	939092
35	1748979	1093559	1096373
40	2034271	1289894	1282010
45	2360460	1782224	1780282
50	3005667		

Table 5.5: Tree creation (milliseconds) for TNT.

Data size Mb	PROTEIN	C. ELEGANS	HUMAN
1	18341	14340	13941
5	116453	146768	89427
10	277608	275586	195453
15	528018	430138	325904
20	793786	530258	469364
25	1004392	671230	629826
30	1219113	830099	801455
35	1571219	1279580	1033430
40	1841070	1483357	1224281
45	2128671	1732245	1422216
50	2408032	1701047	1620394
55	2901096	2125339	1978192

Table 5.6: Tree creation (milliseconds) for SBST.

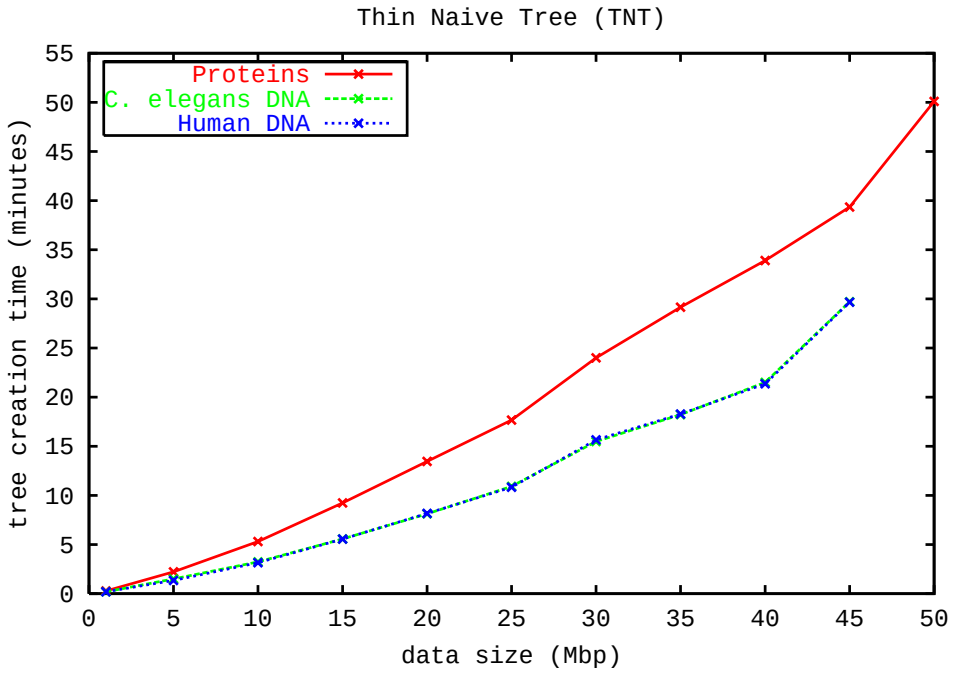


Figure 5.12: Time required to build a TNT index. The graph for worm DNA indexing is superimposed on the graph for human DNA tree construction.

the number of equal and unequal character comparisons made during the tree construction. This phenomenon of longer tree creation for proteins than for DNA was also observed for the *SBST* implemented in Ada (Rob Irving, personal communication), and is possibly attributable to the overall shape difference between DNA and protein trees.

Comparison of all five data structures

We now present a summary of tree creation times which compares all five data structures examined. Index creation for human DNA (very similar to *C. elegans* results) is shown in Figure 5.14, page 95, while Figure 5.15, page 96, shows index creation for the protein data set. Surprisingly, there appears to be no major difference in the tree creation times for alternative indexes. The major limitation in this experiment turns out to be the available RAM, and leaner data structures permit larger indexes. With respect to space, $SBST$ is the best choice, and both naive suffix trees follow closely behind. There is no visible difference in tree construction time between the naive suffix tree and the $O(n)$ suffix tree. This means that the additional space complexity provided by the suffix links, and the additional cost of traversing them cost as much time as the additional character comparisons performed in the construction of the naive suffix tree, for the full range of trees that we could construct using 2 GB RAM and the latest version of Java.

The role of garbage collection strategies

Index creation times reported are highly dependent on several system variables. Our comparison of the first SBST execution under Windows NT and Solaris has already been re-

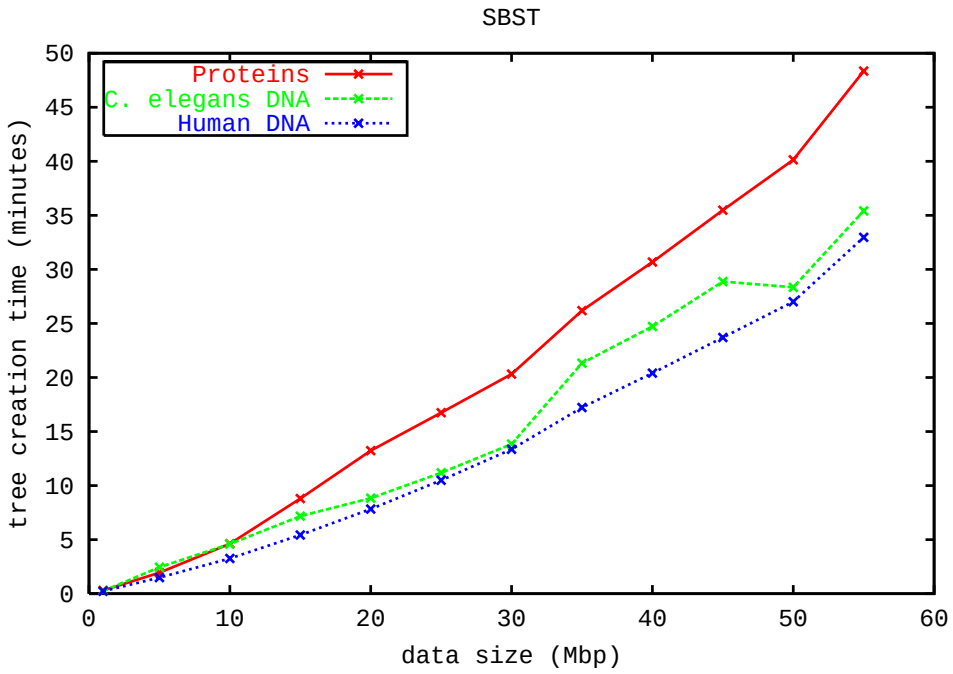


Figure 5.13: Time required to build an SBST index.

Data size (Mbp)	standard GC	No young generation
12.8	421	136
14.2	479	285
15.0	503	296
16.8	574	319
17.4	603	336
20.6	728	371

Table 5.7: Comparison of transient ST building times (in seconds), for the STL data structure, using standard GC and a GC version with no young generation.

ported (20 minutes under Solaris and 2 minutes under Windows for the same data set but different Java Virtual Machines). Another example is the garbage collection strategy which plays a significant role in the performance of tree construction. Our earlier experiments with GC options available in Java 1.2 showed that tree creation time can be halved, if no young generation is used [178]. We reproduce here this measurement which was performed for all *C. elegans* chromosomes and the *SLT* data structure. Table 5.7, see page 94, shows that the copying of object references from the young to the old generation while creating a data structure which produces little garbage has a $significant$ impact on the tree creation time. In fact by not using the young generation, we can halve the time needed for tree creation.

Timing Kurtz's tree

We summarise the tests carried out using Kurtz's suffix tree implementation [138] which we discussed in Chapter 4. The purpose here is not to compare with our implementation, but to

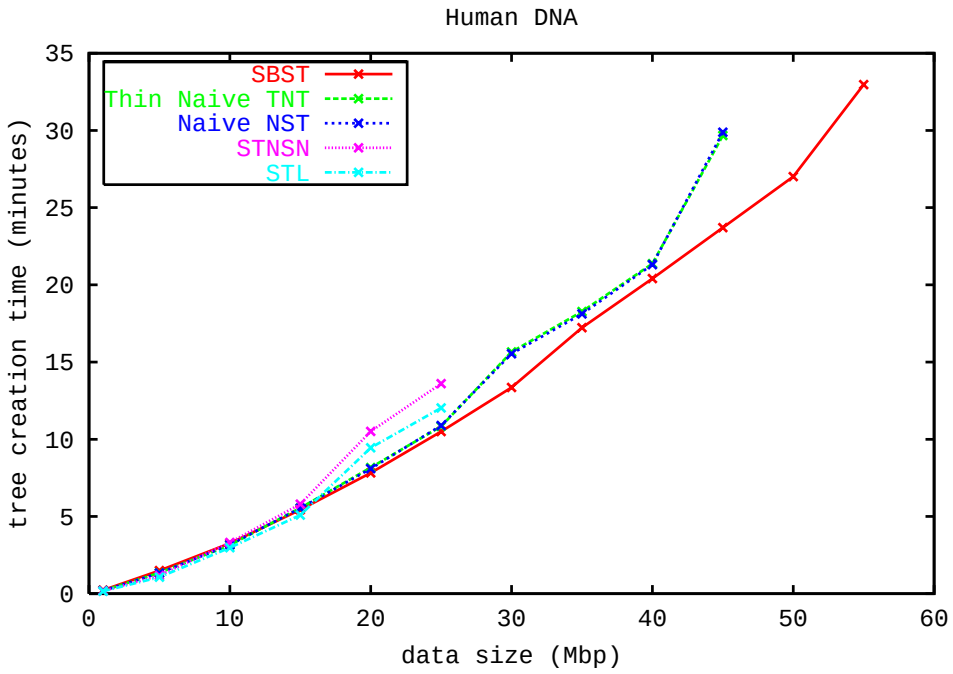


Figure 5.14: Time required to build an index for human DNA, the graph for TNT is superimposed on the graph for NST.

provide background on the fastest algorithm available which is faster and more efficient than the suffix tree used by Celera for human genome analysis [66, 226] (Stefan Kurtz, personal communication, 2001). Kurtz's lean tree structure is implemented in C. We obtained the code from the author and carried out a test on all of the *C. elegans* genome, human DNA and on proteins, and tested two data structures he provides - the improved linked list and the improved hash table implementation. The time required to build a linked-list based tree for the size of the *C. elegans* genome (96,934,461 bp) differed widely between the 3 data types we used. We report the minimum time measured over two runs on a machine that was otherwise idle. For *C. elegans* the tree construction needed 12 min 28.14 sec, for human DNA the time recorded was 9 min 20.50 sec, and for proteins 20 min 18.19 sec. We then tried to test the hash table based implementation. It turned out that the maximum data size allowed was below 9 Mb. For larger datasets we encountered error messages "Sorry, textlen = 9693447 is larger than maximal textlen = 8388605", and "Can't find primetab larger than 100485343". The time needed for tree creation for 8 Mb was 32.84 sec for *C. elegans*, 33.09 sec for human DNA, and 29.93 sec for proteins. We discovered that the C code provided by Kurtz does not separate the time required to read in the data file from the time needed for tree construction, and we decided not to modify the code. To overcome this problem we simply created appropriately sized input files for this test. We recorded the time to run the program using the Unix *time* command. We notice that the protein tree requires longer to build using the linked list structure, but slightly shorter using the hash table implementation. The difference in construction time between DNA and proteins for the linked list implementation is very significant. This difference parallels the tests we carried out with our data structures, where protein index construction took longer. A

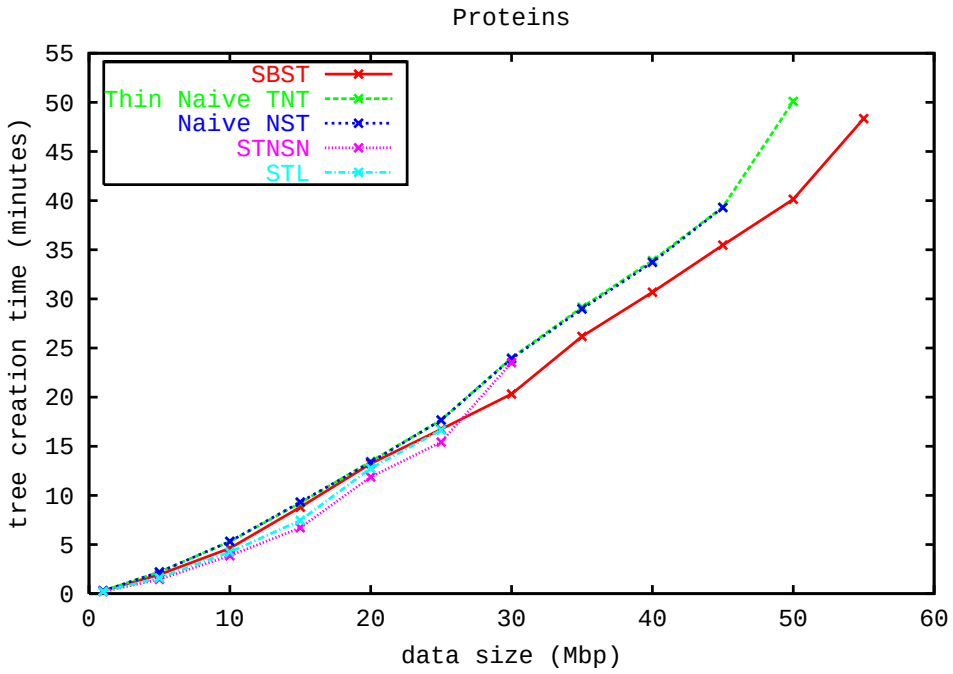


Figure 5.15: Time required to build a protein index.

direct comparison of tree construction times is in favour of Kurtz's tree. Java data structures require more memory and therefore take longer to build. For this reason we are not capable of building memory-resident indexes for 100 MB of source data, which is possible using Kurtz's code. Based on these tests we assume that the time needed to construct a tree for the human genome using Kurtz's tree can lie somewhere in the region of 5-9 hours (this confirms our previous estimate of 9 hours, based on our measurement of tree construction for the worm data in around 18 minutes, using Kurtz's REPuter [139] software, which did not allow for separating tree construction from tree traversal needed to calculate repeating motifs).

5.3 Small persistent trees

Our work is concerned with prototyping an indexing solution using available persistence mechanisms. We used PJama which provides a generic mechanism of persistence, and is designed to provide several database features which may interfere with fast creation of objects on disk. For instance transaction logging [100] consumes a lot of resources and slows down the process of tree creation. As a result, the time needed to make data structures persist on disk is much longer than the actual time needed for disk writing of objects outside the database context. This apparent inefficiency is a practical impediment, as more time is needed for testing. On the other hand, the fact that PJama offers recovery facilities is of practical benefit once the tree is built. This is needed for testing, as some tests may cause a system crash and some may have to be aborted, if they take too long to execute. We believe that with advances in persistence, solutions to the problem of write-once indexes will appear

and they will be efficient enough to make write-once read-many-times indexes viable.

We used two distinct versions of PJama in this work, as described previously. Since we are using a prototype implementation of persistence, and PJama is a general purpose persistence mechanism, we do not focus here on the actual time needed for tree creation on disk, but investigate the feasibility of indexes of various sizes and their performance within the constraints of the technology we are using. From that point of view the maximum size of the data structure possible is of great interest, as the disk-resident data structures we created index more text than any of the approaches used by other researchers who investigated suffix trees and suffix arrays.

5.3.1 Persistent *STL* tests

A suffix tree with suffix links, using Ukkonen's algorithm, was constructed first. It was limited by the size of RAM (2 GB). Small trees (up to 12 Mbp of sequence) required just an invocation of PJama runtime environment, and no explicit checkpoints (data being commited to disk by invoking the method $OPRuntime.checkpoint()$). For larger trees, checkpoints during the tree creation had to be added. Trees for datasets smaller than 20 Mbp were the largest that could be built until the Autumn of 2000, as the log kept by PJama was not large enough to keep all the records required for recovery. Since the new version of PJama appeared in September 2000 [155], it has become possible to create a tree for 20.5 Mbp of DNA, using *C. elegans* Chromosome 5 data, taking up 2 GB of disk. This version of the suffix tree initially used checkpoint granularity of 300,000 new suffixes, and it needed 43 hours to build on disk. Further runs explicitly called the GC after each checkpoint, and with checkpoint granularity of 0.5 million, the same tree was build in 8 hours.

Trees larger than 20.5 Mbp, i.e. the size of available RAM, could not be built using this data structure, which is also reflected in the observations made by Navarro and Baeza-Yates [25].

5.3.2 Persistent *SBST* tests

In September 2000 tests on suffix binary search trees were carried out. Initially an SBST index for 20.5 Mbp of *C. elegans* DNA was created. The store containing the index occupied 1 GB of disk, and fitted in memory (2 GB RAM), which enabled a straightforward store creation. Subsequently we built a store for all of worm data (97 Mbp), as 6 trees in one store, occupying 4.7 GB of disk. The full store creation took some 8 hours, and after each tree was added a checkpoint took place. This contrasts with our unsuccessful attempt to build a forest of 6 STL trees using a similar method. We also tried to build larger *SBSTs* but found that trees over 45 Mbp of DNA were not possible, even with partial checkpoints. This was due to the fact that after each checkpoint we had to modify the structure which was already disk-resident, and the persistence mechanism found the management of large updates pertaining to the addition of large numbers of leaves throughout the data structure too hard to orchestrate (most of the tree had to be kept in memory as updates were randomly distributed, and the amount of memory at our disposal was limited). We conjecture that with the method of tree building in partitions presented in the next section we will be able to build larger *SBSTs*. Further work could investigate the potential of this technique.

5.4 Naive suffix tree for an arbitrarily large index

5.4.1 The memory bottleneck

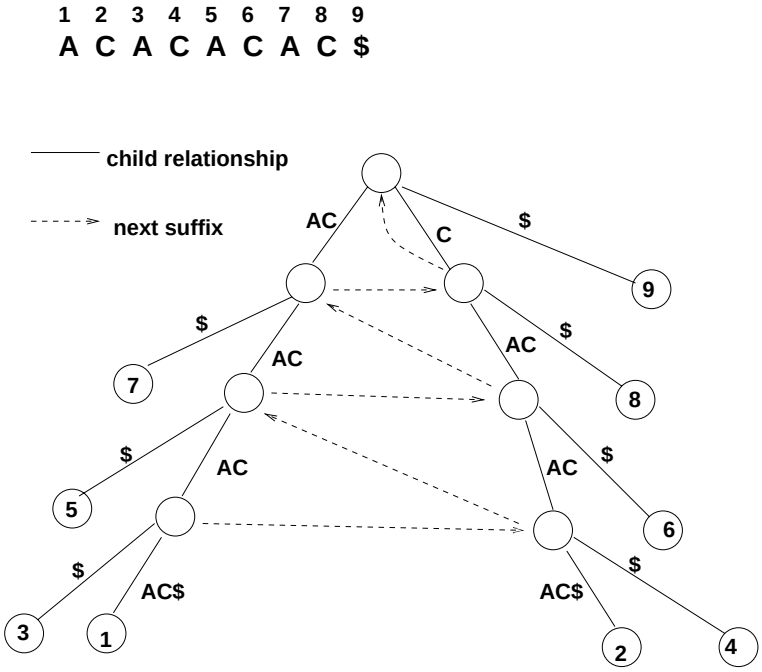


Figure 5.16: Suffix tree with suffix links for **ACACACAC$**.

After the development of the naive tree code it turned out that this tree could be built on disk more easily than the suffix tree with suffix links. We now present an explanation of why that is the case. First of all, there is the space factor, as the naive tree does not require suffix links, but this factor is of minor importance. What is significant in a naive suffix tree is its simple structure of child and sibling links in the direction from the root towards a leaf. In contrast to the naive tree, in the suffix tree with links, links traverse the tree horizontally, see Figure 5.16, and contribute to tree traversal in two directions (from the root down, and horizontally from suffix to suffix).

Suffix links [224] are defined as follows. Given an internal node indexing aw where length of a is one, there is always a suffix link to a node indexing w. These suffix links, necessary for the $O(n)$ tree construction, are an impediment in the disk-based tree construction. This effect has been observed before and is referred to as a "memory bottleneck" by Farach and co-authors [79]. In Farach's work this bottleneck is attributed to the difficulty of string sorting, and his efforts focus on overcoming the difficulty by proposing a new theoretical approach to tree building. In our opinion, the problem is caused by suffix links, and removing those leads to practically applicable algorithms, as shown in our work.

5.4.2 Tree construction

Our algorithm allowing us to build arbitrarily large trees is based on the naive tree construction. It first pre-groups the suffixes down to a given prefix length and then builds a

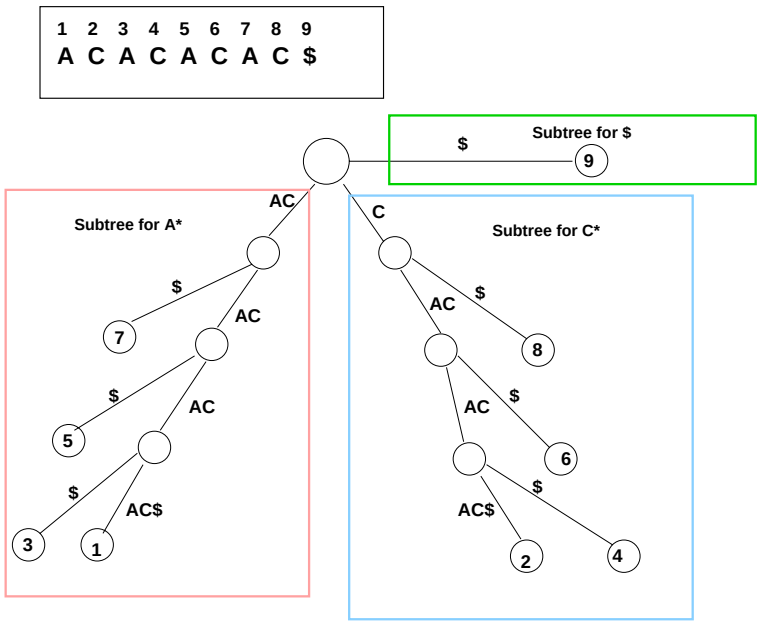


Figure 5.17: Suffix tree with suffix links removed, consisting of 3 subtrees, based on prefix length of 1.

tree for groups of suffixes which are lexicographically close together, shown in Figure 5.17. This grouping allows us to partition the tree construction into distinct phases. In each phase a subtree is built and committed to disk, and the subtrees create distinct tree partitions. Once a partition is completed, it is never revisited. We illustrate this fan-like structure for strings over a 3-letter alphabet (ignoring the terminator), which results in 9 partitions, see Figure 5.18. The root has three main subtrees, one for all suffixes starting with A, one for suffixes starting with B, and the third one for all suffixes starting with C . Those subtrees are further subdivided, based on the second character of each suffix. We first build the subtree for all suffixes starting with AA, commit, and then proceed with the AB prefix, commit, and so on, until the last CC prefix is built and committed to disk. In a naive suffix tree there are no suffix links, and those subtrees are built independently. For the same reason, because subtrees can be made to fit entirely in memory, post-processing of subtrees to achieve data compression and optimal placement on disk is now possible.

In other words, we base our partitions on the prefixes of each suffix. In this section we will use the DNA alphabet for simplicity of presentation, however the same tree construction mechanism applies to protein and has been tested in practice on both alphabets. In the DNA alphabet of *A, C, G, T* the suffixes that have the prefix **AA** fall in a different subtree from those starting with **AC**, **AG** or **AT**. The number of partitions and hence the length of the prefix to be used is determined by the expected size of the tree and the available main memory. It may be the case that smaller partitions (for instance based on the first three, four or five characters of each suffix) would be better because their impact on disk clustering would accelerate lookups, but this has yet to be investigated.

The number of partitions required can be computed by estimating the size of a main-memory instantiation S_{mm}, available for tree construction, and the number of partitions, p,

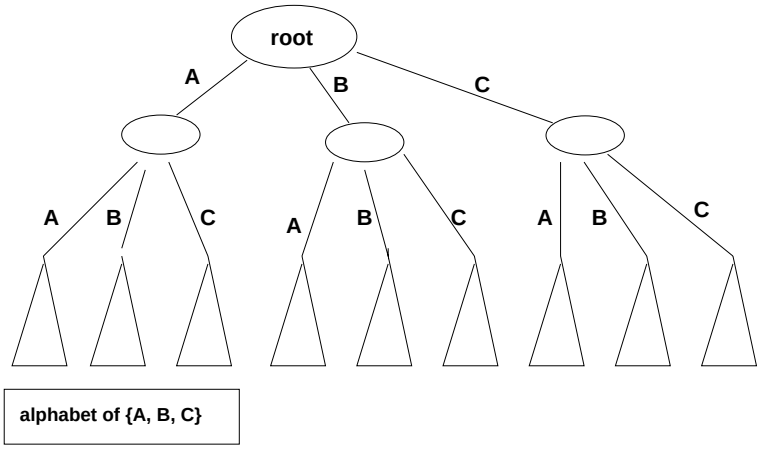


Figure 5.18: A fan-like structure of the partitioned suffix tree, with 9 partitions, using prefix length 2.

is

$$\left\lceil \frac{S_{mm}}{A_{mm}} \right\rceil,$$

where A_{mm} is the available main memory. The actual partitioning can be carried out using either of the two approaches we outline. One way is to scan the sequence once, for instance using a window of size 3 (sufficient for 263 Mbp of DNA and 2 GB RAM), count the number of occurrences of each 3-letter pattern, and then pack each partition with different prefixes, using a bin-packing algorithm [59]. Alternatively, we can assume that, given the pseudo-random nature of DNA, the tree is uniformly populated. To uniformly partition, we calculate a prefix code, P_i, for each prefix of sufficient length, l, using the formula:

$$P_i = \sum_{j=0}^{l-1} c_{i+j} a^{l-j-1},$$

where c_k is the code for letter k of the sequence, and a is the number of characters in the alphabet. The code of a letter is its position in the alphabet, i.e. for DNA **A** codes as 0, **C** codes as 1, etc. The minimum value for P_i is 0 and its maximum is $a^l - 1$. So the range of codes for each partition, r, is defined by:

$$r = \left\lceil \frac{a^l - 1}{p} \right\rceil.$$

The suffixes that are indexed during the jth pass of the sequence have

$$jr \leq P_i < (j+1)r.$$

The structure of the complete algorithm is given as pseudocode below:

```
for j in partitions do
  for i in 0..totalLength do
   if suffix i is in partition j
    new Node(i);
    insert Node(i);
   endif
  endfor
  checkpoint; (write to disk)
endfor
```

5.4.3 Space requirement of the thin naive tree

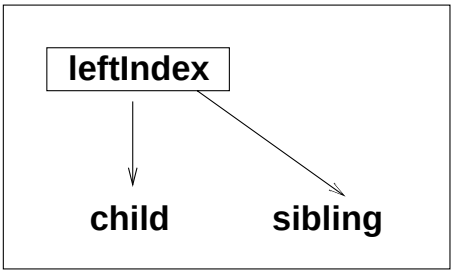


Figure 5.19: Node of a Thin Naive Tree (TNT).

Our TNT implementation disposes of suffix links, see Figure 5.17. A single tree node is shown in Figure 5.19. We reduce storage by not storing the suffix number and the right index into the string for each node. The suffix number is calculated during the search. The right pointer into the string is looked up in the child node, or, in the case of leaves, is equal to the size of the indexed string. Each tree node consists of two object references requiring 4 B each (child, sibling), one integer taking up 4 B (leftIndex) and the object header (8 B for the header in a typical implementation of the Java Virtual Machine). The observed space is some 28 B per node in memory in the persistent context. The difference is due to PJama's housekeeping structures, such as the resident object table [143] which account for around 8 B per object.

PJama's structure on disk adds another 8 B per object over Java, i.e. 36 B per node. The actual disk occupancy of our tree is around 65 B per letter indexed, close to that expected. The observed number of nodes for DNA remains between $1.6n$ and $1.8n$, where n is the length of the DNA, giving an expectation of between 58 and 65 bytes per node. Some of this space may well be free space in partitions, and some is used for housekeeping [178]. If we encoded the naive suffix tree without making each node an object (using arrays which grow as the tree is being built which involves array copying), we would then require at least 12 B per node, that is around 21 B per character indexed. Further compression could be obtained by using techniques similar to those proposed by Kurtz [138], described in Chapter 34.

5.4.4 Persistent indexes for large data sets

We constructed large trees for two datasets. One indexed the merged content of human chromosomes 1, 21, and 22 (263 Mbp in January 2001, corresponding to the full length of chromosome 1), the other indexed the merged protein sequences from TREMBL, SWISS-PROT and newTREMBL, totalling 200Mb, using data which we previously described. The tree for 263Mbp is roughly 13 times larger than the largest tree with suffix links we managed to build, i.e. for 20.5 Mbp.

We did not concentrate on possible refinements to the tree construction. Our running times represent unoptimised tree builds. The first run to construct a DNA tree for human DNA for 263 Mbp in January 2001 took 19 hours. A log file of 2 GB was used and the disk requirement of that tree was 18 GB. A protein tree for 200Mb took around 8 hours elapsed time to create on disk, and required a log file of 2 GB and 12.5 GB of store files.

The strategy for the DNA tree was to proceed alphabetically from a tree indexing all suffixes starting with AAA, to AAC, AAG, and all character permutations through to TTT, and then adding any suffixes which had the concatenation character ***** in the first three characters of the suffix. After every triplet of prefix letters a checkpoint was made. The strategy for the protein tree was slightly different, because by that time we decided to focus on approximate matching which terminates on reaching a separator symbol. In an application focusing on motif discovery which we initially had in mind a full tree would be required. We took all two-letter prefixes over the protein alphabet in lexicographic order, and we accumulated the number of nodes added. When that number reached 4 million the program performed an explicit checkpoint. We decided not to add any suffixes starting with ***** or having ***** as a second character, as those suffixes are not useful in searching (the search stops whenever ***** is encountered). Further space savings in the tree could be achieved by extending this technique for instance by excluding suffixes which have ***** as their third or fourth character, depending on the tree size, alphabet, query length, and allowed error margin. This exclusion of some suffixes will have a negative impact on our ability to find repeating words in a tree, but is sufficient for approximate matching.

5.5 Exact matching with indexes

We explore exact matching using suffix trees and suffix binary search trees. For information about exact matching without the use of indexes the reader may consult Gusfield [99] or an online publication by Charras and Lecroq [48]. We did not explore approaches which are appropriate in the indexing of linguistic text either, and information on those is widely accessible [27, 234]. Our work does not focus on exact matching but on an approximate matching algorithm which calculates a similarity matrix. Therefore we report only the results we produced during the initial work on tree indexes, and do not report on all the data sets.

5.5.1 Exact matching using a suffix tree

Exact pattern matching in a suffix tree involves a traversal of one path down the tree and subsequent traversal of all nodes below the last matching character. From the root we trace the query until either a mismatch occurs, or the query is fully traced. In the second case, we then traverse all children and gather suffix numbers representing matches. If we ignore the

alphabet size, the complexity of a suffix tree search is $O(k+m)$ where k is the query length and m the number of matches in the index. This means that in a balanced tree looking for queries of length q brings back a $\frac{1}{a^q}$ fraction of the whole tree, on average, where a is the size of the alphabet. For example, a query of length 4 using a DNA index might retrieve $\frac{1}{256}$ of all tree nodes. The threshold at which indexing begins to show an advantage over a linear scan of DNA depends on the precise data structure used, on the query pattern, and on the size of the sequence. For short exact queries, an alternative q-gram index may be more appropriate. In a tree indexing proteins searches for short substrings are expected to deliver fewer hits, and query performance will be different. Our experiments with approximate queries on a protein tree are reported in Chapter 6 and they support this conjecture.

The exact matching algorithm

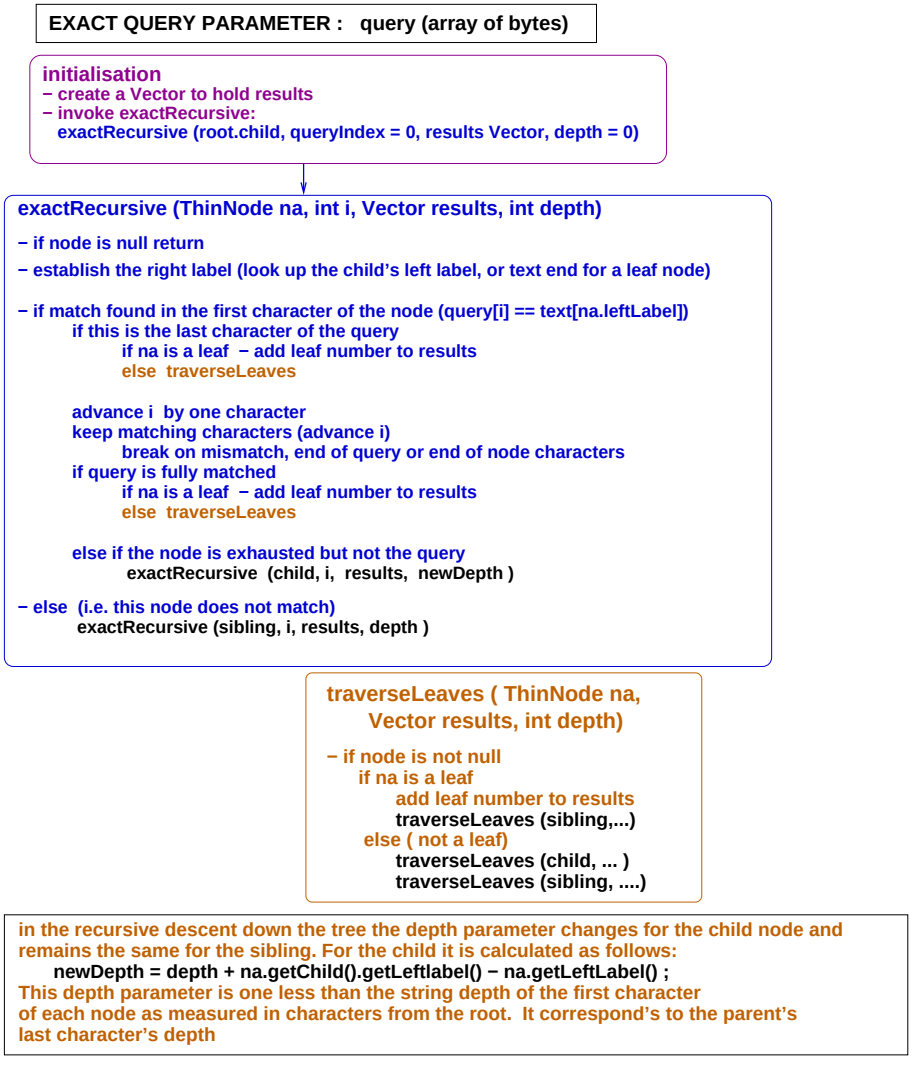


Figure 5.20: The algorithm for exact string matching using a suffix tree.

The exact matching algorithm implemented by Cox [60] was discarded because we found

query length	Number of hits	SBST avg time (ms)	ST avg time (ms)
8	not measured	.0503	3.7215
9	12997843	.0437	1.1123
10	3887138	.0450	0.3566
15	19341	.0460	.0387
20	not measured	.0449	.0377
50	1831	.0448	.0375
100	945	.0456	.0415
200	413	.0485	.0367

Table 5.8: Comparison of query response times while using transient ST and SBST indexes for 20.5 Mbp of *C. elegans* DNA sequence, averaged over 10,000 queries.

it hard to reason about its correctness. We implemented a recursive tree traversal for exact matching. We summarise the technique we used in Figure 5.20, and refer here to the naive suffix tree, as this simplifies the presentation. During the tree traversal an index into the query string is used, called i, and another integer, called $depth$, allows the tracing of the string depth of any node, as counted from the root. We refer to the $depth$ of the last character of the parent node. To record matches, we use the $java.util.Vector$ class. We believe our early measurements of exact query performance which we report here are distorted because of the use of this construct. It has later come to our notice that using Java Collections slows down the execution of Java code, and a handcrafted data structure which can expand more efficiently than $Vector$ would be more appropriate.

We tested the correctness of our implementation of this algorithm by direct comparison of the result set returned by the suffix binary search tree and our suffix tree for the same indexed sequence and the same set of queries.

Exact matching in a transient tree

We carried out tests on memory-resident indexes, using the least space efficient version of the suffix tree (STL) and an SBST in order to gain an appreciation of the difference between those data structures. Tests were carried out on chromosome 5 of *C. elegans* (20.5 Mbp), and query response was measured for different query lengths and different batch sizes. We present a summary of data gathered from the run of 10,000 queries in Table 5.8. For longer queries the performance of both memory-resident trees is similar, however, short queries over SBSTs are faster. This may be due to different tree topologies. In a suffix tree a search for a string of length 9 traverses a maximum of $9 * alphabetSize$ nodes down the tree, but then has to scan all the nodes below the ninth node, including all the intermediate nodes leading towards the leaves, to find the relevant suffix numbers. In an *SBST*, on the other hand, the path leading to the identification of the $9th$ matching character may be considerably longer, but a smaller subtree needs to be scanned completely in search for matches. The other contributing performance factor is that an *SBST* occupies only half of the memory required for a suffix tree, and could therefore be inherently faster.

In a further investigation it might be worth timing the finding of the first string occurrence (traversal down the tree) and then tracing of other occurrences (subtree traversal). The big performance difference between the ST and the SBST seems to point to the fact that SBST might be more suitable for exact searching especially with short queries. Since

Query length	Number of hits	SBST avg (ms)	ST avg (ms)
9	12997843	38	331
10	3887138	19	107
15	19341	12	14
50	1831	13	12
100	945	13	12
200	413	13	12

Table 5.9: Comparison of average query response times using 10,000 queries running over a persistent ST and a persistent SBST, based on 20.5 Mbp of *C. elegans* DNA.

exact matching is not the focus of this research, we did not carry out measurements with different variants of the suffix tree. Future work might include executing exact query tests for both proteins and DNA using all of the data structures we implemented.

Exact matching using a disk-resident index

Using chromosome 5 of *C. elegans* (20.5 Mbp) we constructed a suffix tree and an *SBST*, each in a separate PJama store. The *SBST* store measured 1 GB, and the suffix tree store measured 2 GB. The exact matching test was then carried out, and we report the results in Table 5.9. For this dataset the SBST fits completely in memory, and the ST is mostly memory-resident. The *SBST* is faster than the suffix tree for short queries and these results are analogous to the results recorded on transient trees, with a significant proportional decrease in speed due to the loading of all tree parts from disk into memory structures, and to the larger memory image of a persistent data structure. A transient tree performs, on average, 2 to 3 orders of magnitude faster than a persistent tree, and for longer queries the *SBST* seems to be equivalent to an *ST*.

A disk-resident forest of SBSTs

Subsequently, we attempted to create a forest of chromosome trees for *C. elegans* within a single store. We did not succeed in creating a forest of suffix trees, but created and tested a forest of six *SBSTs*.

The purpose of this test was to investigate the performance of PJama under multithreading, and to understand the possible gains from a multi-processor environment. We did not follow this line of enquiry further, however after this test we are certain that the use of multi-threading will be beneficial, and further work might examine how to achieve maximum benefit from a multi-processor machine. For all previous tests the index structure fitted in the available RAM. For 6 trees held in a 4.7 GB store this was no longer the case, and we wanted to see how changing the number of threads used would influence query response. We wanted to exclude artifacts resulting from the use of a synchronised request queue, and make sure that there were always threads ready to execute a query.

A nursery of 6 SBSTs was created, a tree per each of the six chromosomes of *C. elegans*. Chromosome sizes were as follows (in Mbp): 14.3, 15, 12.8, 16.8, 20.5, 17.4. The trees were built within one PJama store. The full run for tree creation took 8 hours. A batch of 1000 queries of equal length was prepared, broken into smaller sub-batches, and placed in a queue where batches circled over the trees, and each tree would receive all 1000 queries.

Query length	threads	batch1 avg (ms)	batch2 avg (ms)	batch3 avg (ms)	batch4 avg (ms)
10	1	149	529	180	494
	2	48	511	507	521
	4	37	321	335	321
	8	36	276	279	
	16	33			
	32	35			
15	1	374	346	340	342
	2	401	340	382	340
	4	249	223	224	225
	8	201	179	213	183
	16	172	172	175	168
	32	179	179	162	169
20	1	319	348	304	333
	2	349	302	323	306
	4	225	197	195	204
	8	202	163	205	166
	16	173	208	152	162
	32	163	163	156	143
100	1	309	351	358	326
	2	349	342	302	334
	4	225	202	213	198
	8	188	180	172	178
	16	153	170	152	152
	32	147	147	158	167
200	1	310	289	286	290
	2	291	261	238	259
	4	185	167	154	160
	8	137	138	137	140
	16	127	127	131	130
	32	134	134	129	140

Table 5.10: Average time per query in ms, using the exact search algorithm ran on a forest of persistent $SBSTs$ for *C. elegans*.

A number of threads was initialised, and the total execution time from batch submission to completion was measured. We observed some recoverable crashes during this test. We assume that thread management within PJama is still imperfect, and thread interactions must have caused the system collapse. We present the data resulting from four runs, where we measured the elapsed time in ms for 1000 queries. We show *average time per query in milliseconds*, in each of the four runs, see Table 5.10.
We calculate the average query time over the four batches, and produce a graph summarising the observed query performance, see Figure 5.21. We have no plausible explanation for the differences in query performance we observed between queries of length 10 and longer queries. The timings we recorded seem to vary a lot, and we think that system activities influenced the query response times considerably. In all measurements, however, it seems that the optimum system behaviour was reached with 8-32 threads, and using more threads sometimes lead to system instability, which is a possible artifact of the lack of investment

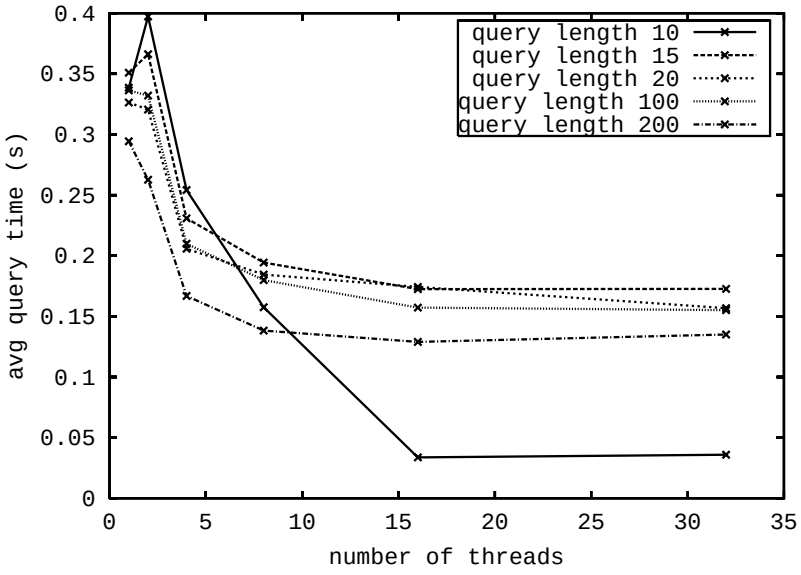


Figure 5.21: Average query times over 97 Mbp in 6 *SBSTs* (4.7 GB PJama store), using a 4-processor Enterprise 450 server.

in the persistent platform.

5.5.2 Exact queries over a large DNA tree

To find out if the increased tree size has an impact on query performance, we carried out tests on the large naive tree for 263 Mb of human DNA. Batches of queries were submitted and the elapsed time for a batch of queries measured. Summary data are presented in Figure 5.22 and in Table 5.11. Analogous summary data for a small transient tree were presented in Table 5.8. It appears that short exact queries which return large numbers of results take longer to process because large regions of disk have to be scanned to find the relevant leaves. For longer queries performance is very good, as either a mismatch generally occurs or just a few matches are retrieved from the tree leaves. We notice that for all queries longer than 15 characters the average time reported is below 0 ms. This is due to the measuring technique which could only provide resolution down to 1 ms, i.e. the call to $System.currentTimeMillis()$. As exact matching was not the focus of our research, we decided not to pursue this measurement at a higher level of accuracy. It appears that larger indexes (for 263 Mbp) offer similar performance to smaller indexes (for 20.5 Mbp). For instance for queries of length 10, and a batch of 2000 queries using a large tree, the average query time was 6 and 9 ms, while in our previous tests with a small tree we recorded a time of 107 ms. This "improvement" of the larger tree over the small one is due to the difference in tree structure and their lay-out on disk. The small tree (STL) had suffix links and was constructed using Ukkonen's algorithm, with intermediate commits which were triggered by the need for free memory, rather than respecting the shape of the index. This resulted in an index with very poor locality of reference. The very large tree (TNT), on the other hand, has no suffix links and higher locality, as subtrees are committed separately and never updated.

batch size	query length	avg Time (ms)	total hits / batch	avg hits / query
500	10	13	945785	1891.57
		14	802808	1605.62
	15	1	56196	112.39
		2	50202	100.40
	50	0	211	0.42
		0	89	0.18
	100	0	59	0.12
		0	60	0.12
	200	0	13	0.026
		0	14	0.028
	300	0	4	0.008
		0	4	0.008
1500	10	10	1633481	1088.99
		8	1346706	897.80
	15	0	100952	67.30
		0	65829	43.89
	50	0	222	0.148
		0	543	0.362
	100	0	60	0.04
		0	61	0.041
	200	0	13	0.009
		0	14	0.009
	300	0	4	0.003
		0	4	0.003
2000	10	6	1628307	814.15
		9	2344782	1172.39
	15	0	103615	51.81
		0	105297	52.65
	50	0	222	0.111
		0	543	0.272
	100	0	60	0.03
		0	61	0.03
	200	0	14	0.007
		0	15	0.007
	300	0	4	0.002
		0	4	0.002

Table 5.11: Exact queries on a DNA tree for 263 Mbp where two batches of queries were submitted for each combination of query length and batch size.

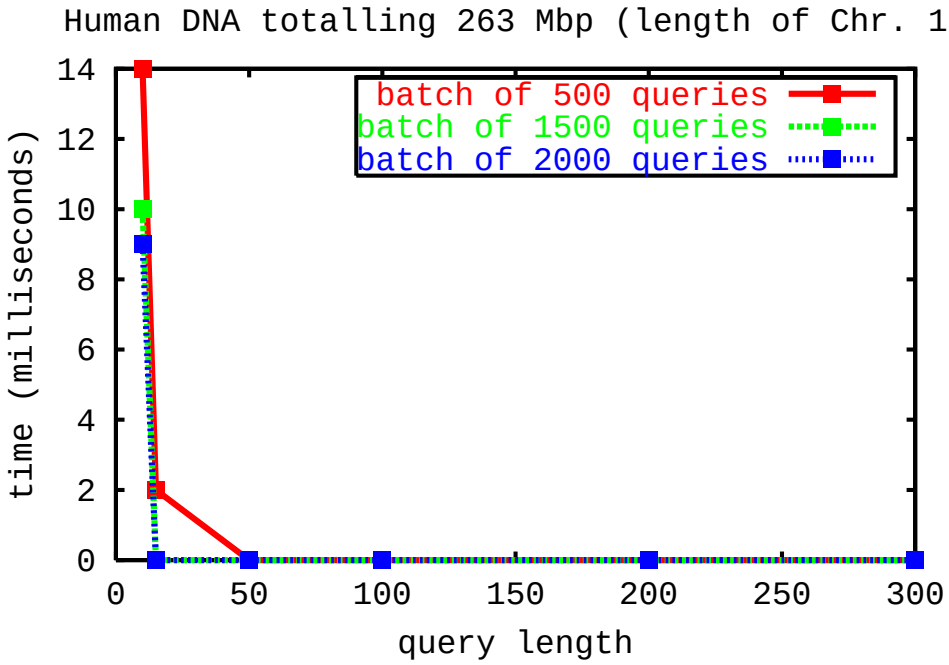


Figure 5.22: Average query times for a warm store.

5.5.3 Discussion of the exact matching tests

Tests on transient and persistent suffix trees show that the unoptimised implementation of *SBSTs* is faster than the unoptimised suffix tree with links. We believe that this is partly due to *SBST's* smaller space requirement. A forest of *SBSTs* indexing 97 Mbp of DNA in 6 trees took up 4.7 GB of disk, using PJama 1.6.5, and a forest of equivalent suffix trees would have needed 9.5 GB, had we been able to complete the forest creation.

These tests force us to consider the following issues in any future work that will follow on from this investigation, some of which have already been taken up by other researchers [128, 186].

- The size of the index needs to be reduced. We have taken up this line of work, and produced the thin naive tree (TNT). Further work at the data structure level and database level is needed.
- As short exact queries return many hits, a suffix tree which summarises leaves below a certain level might be useful. In our measurement we pay the penalty of using $java.Vector$ to store hits, and this increases the reporting cost considerably. Measurements with a more efficient data structure to store the matches are needed.
- DNA and protein strings should be compressed to reduce the size of data which is looked up during the query evaluation. This improvement can be carried out easily.
- The exploitation of parallelism in any future work is a very promising avenue of research and will reduce the time needed to evaluate a query.

- The SBST needs further investigation and exact examination of the performance difference between the $SBST$ and the TNT should be made.
- A more lightweight approach to persistence which does not require a large object overhead on each node would be very useful.
- Tree creation without logging or with more efficient logging should be experimented with, to achieve faster tree creation.

5.6 Summary

This chapter presented our investigation of a variety of alternative suffix indexing structures against the background of biological work that this research is aiming to support. We described the materials and data used, and showed how our work on time and space optimal suffix trees has led to the discovery of the naive suffix tree which in practice behaves as well as an optimal suffix tree, as shown in our experiments. We described in detail the algorithm for the creation of very large suffix trees, based on the naive tree, and showed the performance of the exact matching algorithm on a tree for 263 Mbp of human DNA. The tree we used in our experiments is the largest suffix tree ever reported in literature and contradicts previous statements that suffix trees in excess of the RAM size were not possible. This data structure opens new perspectives in the database field, as it will allow for future indexing of very large sequence repositories.

We now move on to the algorithmic and experimental research we carried out in implementing the approximate matching using a suffix tree index. This work constitutes the second major contribution of our research and is presented in Chapter 6.

Chapter 6

Approximate string matching using a naive suffix tree

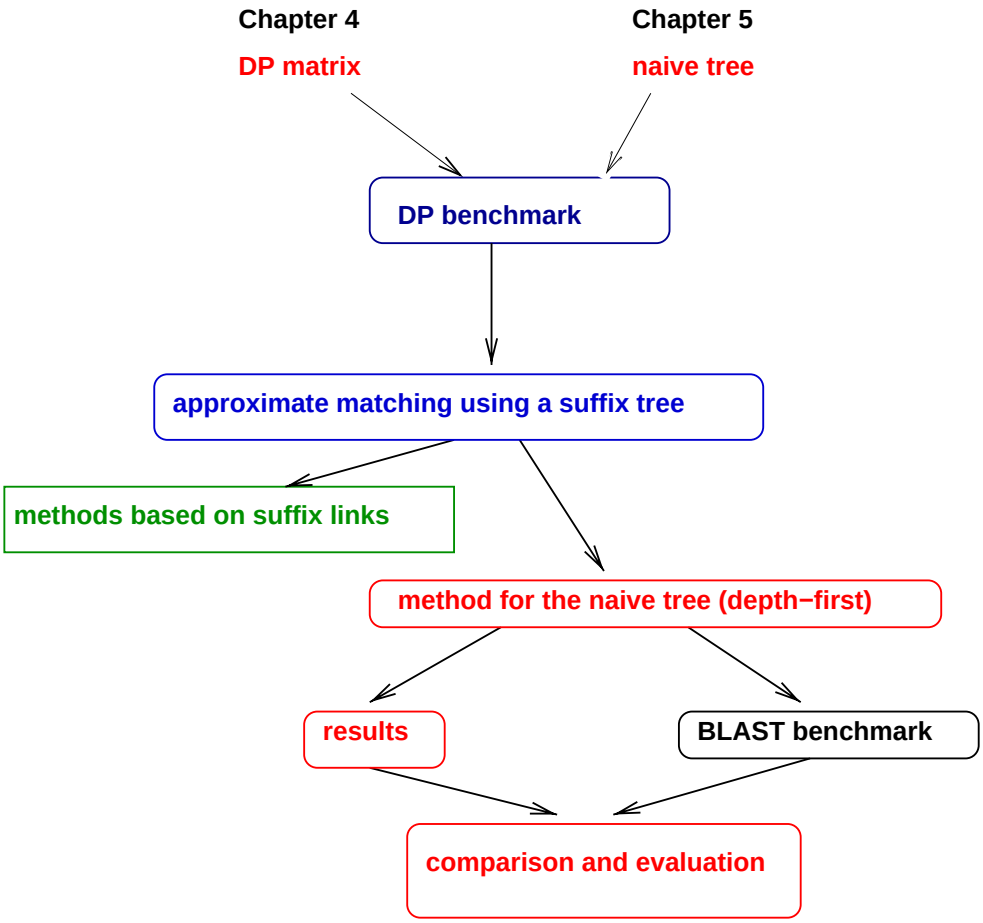


Figure 6.1: Overview of Chapter 6.

We present one of the three main contributions of our research. In this chapter we discuss our new methodology which defines the indexing gain of a suffix index, and use it to ex-

amine the indexing gain for two protein indexes, indexing 36 Mb and 200 Mb of sequence. We describe the implementation details of our approximate matching algorithm which uses the suffix tree, and analyse the performance of the index. This chapter builds on Chapter 4 and Chapter 5. In Chapter 4 the theoretical foundations of our work were presented, including persistence, suffix trees, and approximate matching techniques. Chapter 5 detailed the construction of a very large suffix tree which is the index structure used in the research presented in this chapter.

We describe an approximate matching algorithm which executes the DP matrix calculation on a naive suffix tree. We characterise the benefits of using the suffix tree index to carry out a DP calculation, and argue that using a suffix tree considerably reduces the size of the DP matrix which needs to be evaluated. Our contribution is threefold.

- We implement an algorithm based on [25] and adapt it to reflect the needs of similarity searching in the biological context.
- We build a suffix tree on the largest publicly available protein data set containing 200 Mb of sequence[1], and test the approximate matching algorithm with human protein data[2].
- We present a new methodology for evaluating the indexing gain. We distinguish clearly between the algorithm speed up resulting from indexing, and other possible sources of efficiency, for instance following on from the use of an automaton. We demonstrate that the suffix tree for 200 Mb reduces the size of the DP matrix to be calculated 10 to 100 times, and that the indexing gain grows with the size of the data structure.

A graphical overview of this chapter can be found in Figure 6.1. Our argument is structured as follows. We first describe a DP benchmark which measures the speed of unoptimised matrix calculation in Java. We then detail two possible approaches to using the suffix tree as an index. The first family of algorithms uses a suffix tree with suffix links, and the second one does not use suffix links, and can therefore be executed on the naive tree. We then describe the implementation of the approximate matching algorithm based on the naive tree, and present the experimental results we generated, using protein data. Finally, a comparison with BLAST is presented, and an argument for further work on the suffix tree index is put forward. Additional testing results, based on work with large DNA indexes are presented in Appendix C, an invited paper we submitted recently to the VLDB Journal.

To find out if the combination of a naive suffix tree with the DP matrix is beneficial, we devised a benchmark testing the performance difference between indexed and unindexed approximate matching. We now present this benchmark.

6.1 Dynamic Programming benchmark

This benchmark enables us to measure directly a possible gain from using a suffix tree index. We use the same unoptimised dynamic programming matrix calculation both in the context of a suffix tree and without it. The calculation we carry out measures sequence similarity using a simple metric (unary costs of match, mismatch, deletion or insertion),

[1] http://www.expasy.org
[2] http://www.ensembl.org

and this lets us abstract from the complexity we find in the software which is currently used by biologists. This provides a fair comparison of both approaches without introducing complexity. Extending this calculation to use protein similarity matrices used in biological sequence searching is always possible, and the additional cost of using such matrices and different gap costs should be small, as it would be limited to a look-up in a small array.

Our initial approach to this benchmark was to construct a rectangular matrix indexed by the text and the pattern. We used `short` as the data type for each matrix cell in order to reduce the storage requirement of the matrix. However, this optimisation is not sufficient. Comparing 300 Mbp of DNA to a pattern of length 100 exceeds our available RAM. This led us to the design of a circular buffer holding the matrix. We initialise the buffer to twice the query length, and wrap round after reaching the array end. We do not report the hits, but count them instead to reduce the reporting component and its possible influence on performance (many matches slow down reporting).

The test for this benchmark is carried out with the minimum size of Java heap set to 1 GB ($-Xms1000m$), and maximum to 1.9 GB ($-Xmx1900m$), using Java 1.3.0 and the *-server* option. We are unable to perform large-scale testing in this case, due to the heavy computational demand of this test. A single program execution on the protein dataset of 200 MB and a query of 245 characters took over 15 hours to complete. Therefore our test uses smaller data sets. In this case we expect a linear relationship between the matrix size and computation time. A full matrix is computed in each case for 20 Mb of sequence and a query length between 245 to 688 characters. The same computation is carried out twice for each query. Queries come from the Ensembl file of predicted human proteins[3]. We reproduce the first sequence from this data set:

```
>ENSP00000003603 Gene:ENSG00000000003 Clone:AL035608
     Contig:AL035608.00001 Chr:chrX basepair:97278090

MASPSRRLQTKPVITCFKSVLLIYTFIFWITGVILLAVGIWGKVSLENYFSLLNEKATNV
PFVLIATGTVIILLGTFGCFATCRASAWMLKLYAMFLTLVFLVELVAAIVGFVFRHEIKN
SFKNNYEKALKQYNSTGDYRSHAVDKIQNTLHCCGVTDYRDWTDTNYYSEKGFPKSCCKL
EDCTPQRDADKVNNEGCFIKVMTIIESEMGVVAGISFGVACFQLIGIFLAYCLSRAITNN
QYEIV
```

The file of queries was pre-processed to remove headers and carriage returns. The SWISSPROT and TREMBL sequence file was preprocessed in the same way as for the suffix tree, i.e. sequence headers and carriage returns were removed, and sequences were concatenated using a *****.

The reporting was done as follows. We recorded the length of text, the length of the query and the time needed to perform the matrix calculation (starting the timer after the source data and query were read in). A summary of query execution times recorded is presented in Table 6.1 and in Figure 6.2. The full matrix calculation is expected to have the $O(nm)$ time complexity. The relationship between the size of the array and the computation time is linear and we can calculate a formula to be used in estimating predicted running times for any combination of text and query lengths. An estimation of the parameters can be done using the least-squares method. Before calculation, we converted text size to Mb,

[3] ftp://ftp.ensembl.org/current/data/fasta/pep/

Text size Mb	Query size	text * query (Mb)	time (s)
20	245	4900	5501 5514
20	250	5000	5617 5615
20	317	6340	7167 7148
20	688	13760	16482
20	742	14840	17857 17808
200	245	49000	56340

Table 6.1: DP matrix calculation for a selection of text and query sizes (merged TREMBL and SWISS-PROT) and 5 human genes from the ENSEMBL dataset. The last entry shows one calculation for the entire dataset of 200 Mb which took some 15 hours to produce and other rows use 10% of that data.

multiplied the text size by query size, and converted time to seconds. We used a web-based implementation of that calculation[4] which produced the following:

```
QuickFit: Results

The results of a QuickFit performed at 07:22 on 14-JUL-2001

10 data pairs (x,y):
( 4.900E+03, 5.501E+03); ( 4.900E+03, 5.514E+03);
( 5.000E+03, 5.617E+03); ( 5.000E+03, 5.615E+03);
( 6.340E+03, 7.167E+03); ( 6.340E+03, 7.147E+03);
( 1.376E+04, 1.648E+04); ( 1.484E+04, 1.786E+04);
( 1.484E+04, 1.781E+04); ( 4.900E+04, 5.634E+04);

y = a + bx where:
```

a= 65.7 ($\sigma\ a$ = 1.37E+02)
b= 1.16 ($\sigma\ b$ = 8.49E-03)

```
degrees of freedom = 8
r = 1.000 (p = 0.000),
```

We also used Excel functions *SLOPE* and *INTERCEPT* to get an independent calculation. This produced:

```
slope                1.150278591
intercept            66.05964582.
```

We will therefore use the formula

$$time(seconds) = 1.16 * matrixSize(Mb) + 66.0$$

to produce approximate running time of any matrix calculation.

[4] http://www.physics.csbsju.edu/stats/QF_NROW_form.html

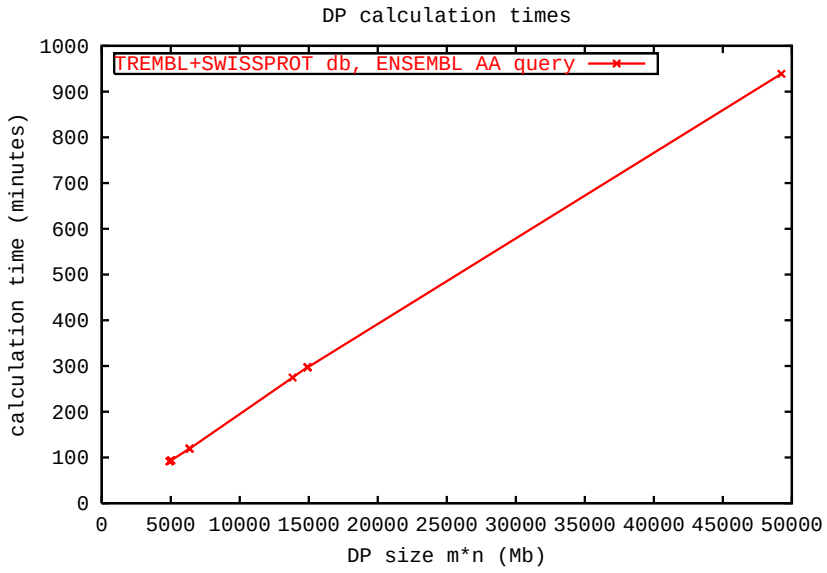


Figure 6.2: Time required to carry out a full DP calculation is proportional to the product of text and query sizes.

6.1.1 Approximate matching using a suffix tree

The DP matrix calculation can be carried out using a suffix tree. In the simplest form of such a calculation, a top-down traversal of all nodes down to a predefined depth is used. In more advanced versions of this approach, presented by Ukkonen [223] and Cobbs [56], suffix links are used to reduce the number of nodes for which the matrix values have to be calculated. Cobbs' algorithm is an improvement on Ukkonen's, whereas Baeza-Yates and Navarro [25] develop a hybrid approximate matching method implemented on top of a suffix array and compare its performance to Cobbs' implementation.

Our work follows the work of Navarro and Baeza-Yates and uses the simple traversal method which does not require suffix links. We proceed now to outline the known methods, and then present our own implementation and testing results.

6.2 Suffix-link based methods

We present a high-level view of those methods, comparing the approach they take with the simple traversals on trees without suffix links. For a detailed description [56, 223] are to be consulted.

Ukkonen presents three algorithms with running times

- $O(mq + n)$,
- $O(mq \log q + outputSize)$,
- $O(m^2 q + outputSize)$.

Here n is the text length, m pattern length, k the number of errors, Σ the alphabet, and q varies depending on the problem instance between 0 and n. For unit cost edit distance it is

shown that

$$q = O(min(n, m^{k+1} \mid \Sigma \mid^k).$$

The purpose of Ukkonen's algorithm is to keep the number of columns evaluated in the matrix calculation $\leq n$, so that overall the computation is faster than the $O(nm)$ needed for a full DP matrix evaluation. Cobbs reduces the running time to

$$O(mq + outputSize).$$

Both algorithms are complex and use suffix links. The main idea is to avoid unnecessary computation of columns which have been computed already, and suffix links enable this optimisation. If a column of the DP matrix corresponding to a particular suffix has already been evaluated, the next shorter suffix need not be evaluated. By using suffix links we find references to nodes which need not be calculated. However, there is an associated space cost which is significant. This optimisation requires keeping a stack or a hash table of nodes and the relevant DP columns, and therefore has a significant space overhead. The fact that suffix links are needed, currently precludes the use of these algorithms in our work. In the future one might consider adding suffix links to an existing tree down to the maximum depth of matrix calculation, after the initial tree build. The space and time trade-offs of such a tree and matrix calculation would then have to be re-examined.

6.3 Depth-first search

This is the name that Baeza-Yates and Navarro use to describe their algorithm which traverses the top of the suffix tree in a depth-first fashion [25]. A traversal down every tree branch is made, up to a predefined maximum string depth, and possibly aborted before the full depth is reached. String depth here is the string length as measured from the root down every existing branch of the tree. The idea of using this kind of traversal is attributed to Gonnet and Baeza-Yates [30], and this technique is used in a sequence analysis package Darwin[5] which dates back to 1992.

6.3.1 Suffix trie simulation

The algorithm looks at the suffix tree as if it was a suffix trie (where one suffix character is represented by one node) and goes down every possible branch to a certain depth (counted in nodes or characters). The depth of traversal has to be kept to a minimum, to guarantee good performance, as the deeper we descend into the tree, the longer the DP matrix we have to calculate becomes and more tree branches need to be explored. In the unit cost model adopted by the authors the maximum depth required for any comparison is the sum of the query length m and the number of errors allowed k. It is also easy to see that minimum depth of the matrix calculation needed to find a match under the same assumptions will be the difference between m and k (after $m - k$ text characters the DP matrix calculation can produce a match with the minimum score $m - k$, and further computation can be abandoned). This condition will become more complicated to evaluate for non-unit costs, and the authors confine their interest to unit costs, which also justifies the use of fast bit-based arithmetic.

[5] http://www.inf.ethz.ch/personal/gonnet/DarwinManual/DarwinManual.html

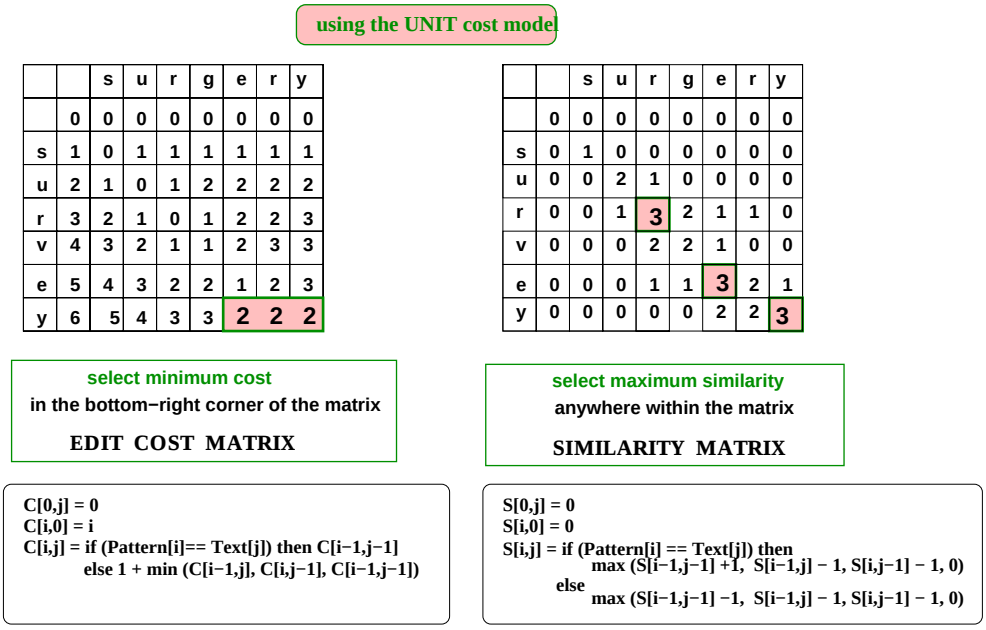


Figure 6.3: Edit cost and similarity matrices for the comparison of the pattern *survey* with the text *surgery*.

Baeza-Yates and Navarro do not use the similarity function [203] but the distance function [142]. A high level view of the difference between the edit and the similarity function for strings is shown in Figure 6.3. The approximate matching algorithm traverses the tree, and extends the DP matrix by one character (one trie node) at a time. After a DP column is evaluated, the algorithm performs two checks. If the edit distance between the entire pattern and the current suffix position is less than k, the whole subtree below this node is traversed, and the leaf numbers of the leaves are reported. If, on the other hand, the algorithm determines that the edit cost at this point is so high that it can never lead to a match with a maximum number of errors k, this tree branch is immediately abandoned. We reproduce the pseudocode as presented by the authors. The function $Update$ is the extension of the currently considered suffix with the letter a encoded by a child trie node. The $SearchState$ is the authors' designation of the stack which holds the DP matrix.

Search (SuffixTreeNode N, SearchState S)

if (S implies a match between P and N)
Report all the leaves below N
else if (S implies that N can be extended to match P)
for each edge tree edge $N \rightarrow N'$ labelled a
Search (N', *Update*(S,a))

The if part of this algorithm is obvious, but the $else\ if$ needs elucidation. The authors say that extending the currently evaluated string with any further characters is impossible when all the values of the last DP column are $> k$. This condition applies to the edit cost matrix. We will adjust this condition to fit the similarity matrix calculation context.

6.3.2 Filtering

This basic tree traversal is enriched by the authors with the addition of a filtration technique. Filtration can be based on the fact that each approximate match contains partial exact matches between the pattern and the text. The suffix tree can be used to locate quickly the $exact$ occurrences of selected pattern pieces, and then some other mechanism can be used to verify the surrounding text areas for approximate occurrences of the whole pattern. This approach is *not* adopted in this algorithm. The authors present a lemma which helps them design the partitions and use the approximate matching in their filter.

> *LEMMA 3.1*
> Let A and B be two strings such that $ed(A, B) \leq k$.
> Let $A = A_1 x_1 A_2 x_2 \ldots x_{k+s-1} A_{k+s}$ for strings A_i and x_i and for any $s \geq 1$.
> Then, at least s strings $A_{i_1} \ldots A_{i_s}$ appear in B. Moreover, their relative distances inside B cannot differ from those in A by more than k.

This lemma is the foundation of several different versions of filtration algorithms. It can be relaxed to permit the presence of some errors in the pieces.

> *LEMMA 3.2*
> Let A and B be two strings such that $ed(A, B) \leq k$.
> Let $A = A_1 x_1 A_2 x_2 .. x_{j-1} A_j$, for strings A_i and x_i and for any $j \geq 1$.
> Then, at least one string A_i appears in B with at most $\lfloor k/j \rfloor$ errors.

Using this lemma the authors partition the pattern into less than $k + 1$ pieces, so that there is no guarantee that any of the pieces will be free of errors. However, they assume that one of the pieces will be present with a reduced number of errors. Their filtration approach corresponds to $P = A$, $x_i = \epsilon$ and $B = T'$, where T' is an occurrence of P in T. The pattern P is split in j pieces and those are searched allowing $\lfloor k/j \rfloor$ errors in the text.

This approach consists of partitioning the pattern, searching each piece, and then checking all the positions found for a match spanning the partitions. The DP matrix is evaluated using a bit parallel NDFA (described in Chapter 4 and reproduced in the next section).

6.3.3 The NDFA simulation

We reproduce the automaton here for ease of reference, see Figure 6.4. Following modifications to the automaton are required in the context of the suffix tree use. The initial self-loop of the automaton is removed, so that no text is skipped before the evaluation starts. This follows from the structure of the suffix tree where each suffix is represented and matching starts at the start of every suffix without any skipping. The first full diagonal (starting at top left) is active at the start. The states in the lower left triangle need not be represented (other suffixes will be explored to find those initial matches). To represent this automaton, the full diagonals are kept. This DP matrix simulation will need $(m - k + 1)(k + 2)$ bits. Taking a computer word of size w, the states are split into as many words as necessary, and the calculation is performed using bit arithmetic on the matrix of states and the matrix of

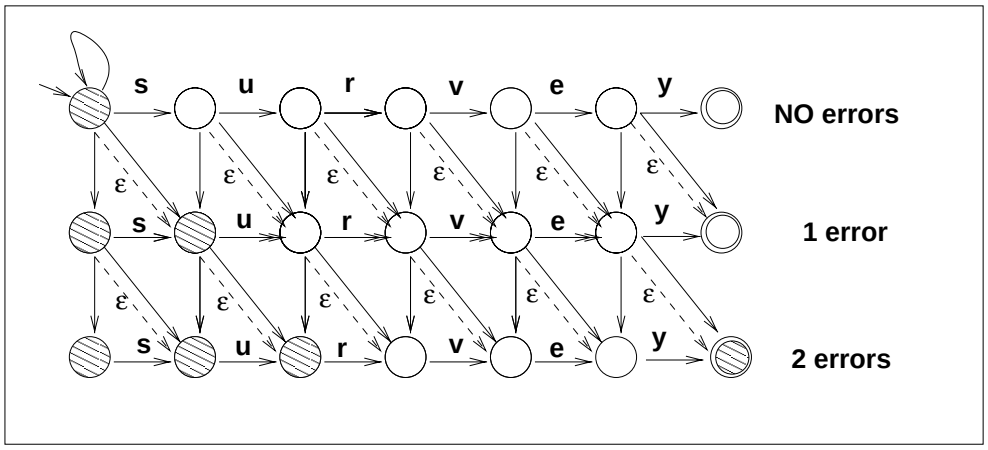


Figure 6.4: A non-deterministic automaton recognising the pattern *survey* with up to two errors. After reading the text *surgery*, the states which are active are shaded, one of them being a final state (reproduced from [25], horizontal arrows represent an exact match, vertical arrows represent a skip in the pattern, solid diagonal arrows correspond to a skip in the text, and dashed diagonals are ϵ-transitions).

masks for the text which is being processed. For the full details of the algorithm the reader should consult [29] where optimised variants for shorter and longer patterns are discussed.

A possible additional optimisation suggested by Navarro and Baeza-Yates is to observe that every approximate occurrence of the pattern in the text must start with one of the first $k + 1$ pattern characters, since otherwise a match is not possible. This limits the number of nodes entered at the top of the tree to at most $k + 1$ different characters.

6.4 Our results

6.4.1 Matching in the protein tree

We performed tests on both transient and persistent trees. For the transient tree we selected the SWISSPROT[6] data (currently 36 Mb), and for database tests we created a tree for the 200 Mb of available protein sequence (SWISSPROT and TREMBL joined). The transient tree and the associated data structures for approximate matching occupy around 65% of available RAM so that no paging should take place during the program execution. The persistent tree uses a 2 GB log file and 12.5 GB of disk storage in 7 files. Tree creation takes between 7 and 8 hours (given 9 GB RAM the tree could be created in memory in 3 hours, based on our measurements for transient trees). Partitions based on two-letter prefixes of each suffix were built one by one, with checkpointing after a partition was finished, if the sum of new nodes added was greater than 4 million. This tree does not include any suffixes which start with a star, or have a star symbol as their second character.

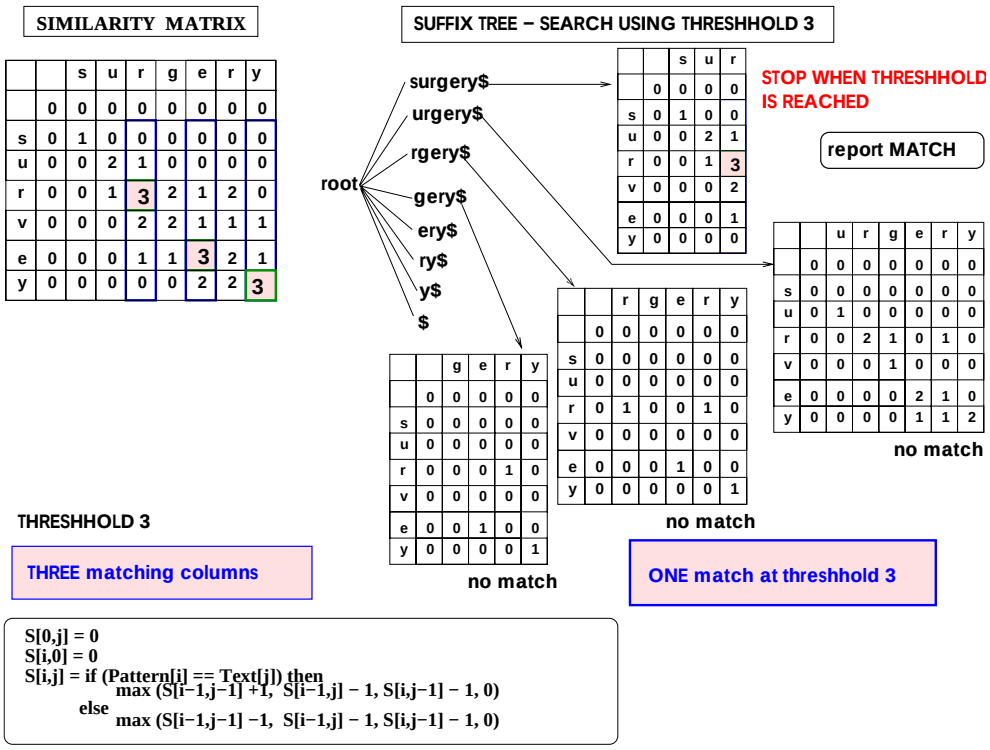


Figure 6.5: The difference between DP matrix evaluation using the text matrix and using the suffix tree. In the suffix tree, on reaching the threshold the calculation stops. For the text $surgery$ and the pattern $survey$ the suffix tree reports one match, while the DP matrix reports three matches, where the second and the third one are extensions of the first.

6.4.2 DP evaluation using a suffix tree

We implemented a version of Navarro and Baeza-Yates's algorithm but kept only the parts of algorithm which fit the sequence comparison context. We first describe the differences between our approach and theirs.

- We index larger data sets (36 Mb and 200 Mb of protein) while Baeza-Yates and Navarro index 10 Mbp of DNA.
- We use a suffix tree, not a suffix array. Navarro and Baeza-Yates were not capable of constructing a large suffix tree and they used a suffix array which is more space-efficient. However, the search time complexity in a suffix array has an additional $\log n$ factor, and is theoretically inferior to that of a suffix tree.
- We use data as currently used by biologists. Navarro and Baeza-Yates used a small bacterial DNA dataset which they concatenated with itself to get sufficiently large data (10 Mb sequence).
- Our tree does not index suffixes which have a concatenation symbol * as a first or second character. This makes the tree slightly leaner, and will not have negative

[6] http://www.expasy.org

impact on the similarity calculations.

- We did not consider optimised bit-based calculations. Those are currently inappropriate in the context of protein cost matrices and gap costs as used in biology.
- We did not consider filtration at this point, because filtration techniques would need to be adapted to varying cost matrices, and further research is required to clarify this area.
- We use the similarity matrix and not the edit distance matrix, and therefore need to derive a new formula to exclude the unnecessary matrix calculations.

Two techniques are used to limit the extent of DP matrix computations needed to compute sequence similarity. The first one is to break the calculation whenever the required similarity threshold is reached, as shown in Figure 6.5, top right matrix, report the match and return to the calling node. If needed, extensions of this match could be carried out if the current state of the matrix and current node reference were preserved. The second optimisation is to stop calculating the matrix whenever we determine that the current calculation will never reach the threshold. The first condition is easy to evaluate. For the second condition, Navarro and Baeza-Yates state that whenever all cells in the *edit distance* matrix are above the threshold, the computation on a given tree branch can be abandoned. We derive a similar test for the *similarity* matrix calculation. We are using unit costs. For non-unit costs, more complex conditions will have to be derived, however this should have limited impact on the speed of calculation. The additional cost will consist of an indexed look-up in an array of 200 short integers for each position in the array (as there are 20 different amino-acids, if the matrix is symmetrical, 200 cells are needed, or 400 cells for an asymmetrical replacement cost matrix). Adding gap costs (which minimise the penalty for a mismatch stretching over a number of characters) will require calculating the distance between two array positions which will have a constant cost per array position. In fact, because we will have to split the query into smaller fragments, accommodation of gaps at this stage may not be necessary, and gap costs may be considered at the post-processing stage.

6.4.3 Our contribution

Our model of similarity searching does not use the notion of error. We reformulate the question as follows.

> Given a query of length q and a threshold t, find all positions in the text where the threshold is first reached in each text suffix.

This means that in a given suffix, extensions of a scoring prefix will not be counted, but only the first score at threshold level will be reported. Such extensions could be calculated after the query has been executed, as performing extensions within the tree index may lead to exponential growth of the number of calculations needed.

To define the cut-off points which minimise the depth of tree probing for possible matches, we now derive the appropriate condition. We start with an example and show possible scores in Figure 6.6. Take a text of length 6, and consider different thresholds that

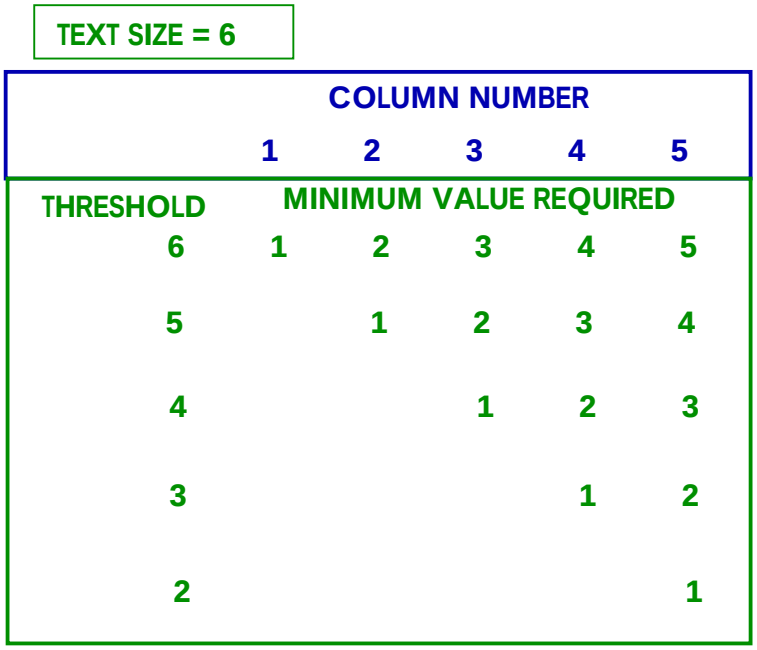


Figure 6.6: Derivation of the minimum value required for the alignment to reach a threshold.

are to be reached as the result of the matching. For threshold 6, there has to be an entry in the column 6 with the value 6. This is exact matching, and we do not show this column. For threshold 5, this value will be 5, for threshold 4 it will be 4, etc. To reach that threshold, the preceding column will have to have the minimum value smaller by 1. And the previous column's minimum will be smaller by 1 again. Therefore to reach value 6 in column 6, column 5 has to be 5, column 4 has to be 4 and column 1 has to be minimum 1. So the first column to check has the index

$$firstColumn = textLength - threshold + 1.$$

We now identify the minimum value which has to be reached in a particular column. The maximum value max for each column is known after the column has been calculated. We continue with the previous example of text length 6 and threshold 6. We stop the calculation under the following condition

$$(col = 1 \wedge max < 1) \vee (col = 2 \wedge max < 2) \vee \cdots \vee (col = 5 \wedge max < 5).$$

The condition to be evaluated starting with the $firstColumn$ is then

$$max < colIndex - textLength + threshold.$$

If this evaluates to $true$, we stop the calculation for this node and return *TERMINATION* condition. This will return control to the parent method, unless there is a sibling node which has not been evaluated yet.

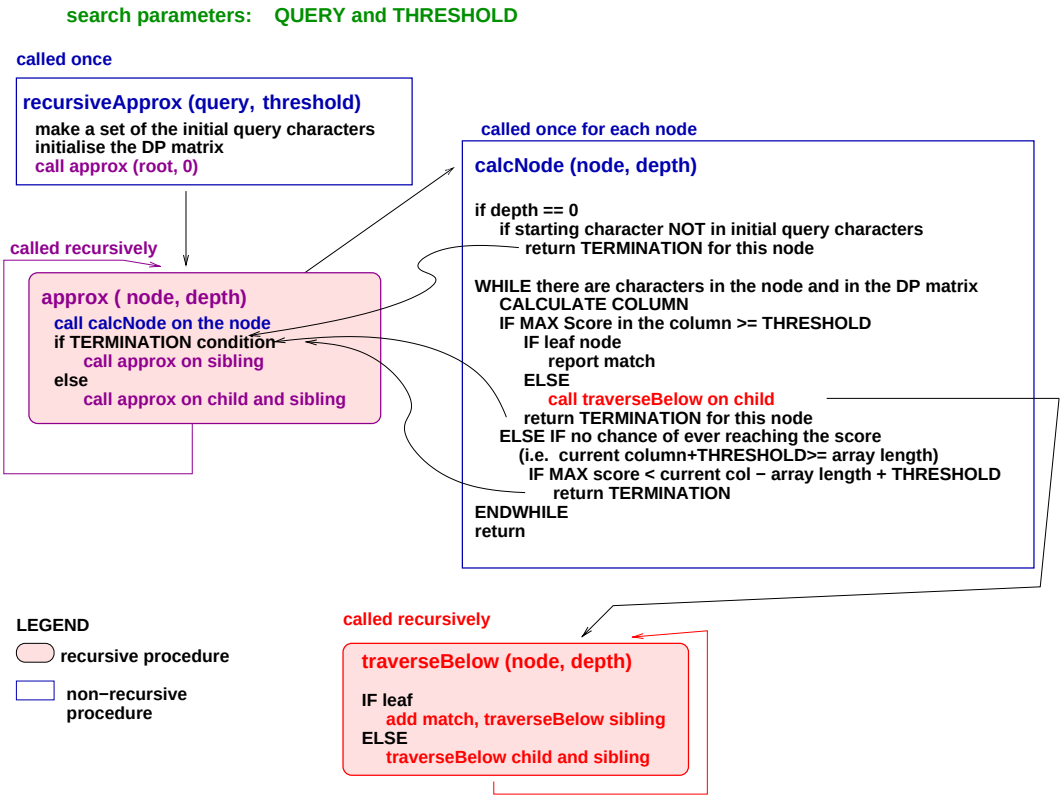


Figure 6.7: The approximate search algorithm.

6.4.4 Summary of implementation

A query of length k with the threshold value t is submitted as an array of bytes to be matched against the text stored as an array of bytes and indexed by the tree. Using the thresholds value, the length of the text array needed to reach the threshold is calculated as

$$size = 2 * k - t + 1.$$

This array is then initialised as a rectangle of

$$size * (k + 1)$$

cells of type $short$. Row and column zero are set to zero (as we are interested in local alignments which can start anywhere in the text and query), and the similarity function set to 1 for match and -1 for mismatch, insert or delete[7]. Approximate matching starts at the root, and the depth-first traversal of the tree follows. Recursive descent of the tree is initiated with the depth value of 0, and as nodes are traversed the depth value is increased to reflect the string depth of the last character in the node's parent. The formula for the calculation of

[7]If a protein similarity matrix were to be used, it would be implemented as an array, and an array look up would be made to find the cost of mismatch between any two amino acids. The look up cost would be constant, and should not contribute significantly to the overall cost of the calculation. Experiments combining the use of a similarity matrix with the suffix tree were carried out this summer [101].

depth values on transition from a parent node to child is

$$newDepth = depth + rightLabel - thisNode.getLeftLabel() + 1.$$

For each node the procedure calculating the DP matrix is called. When the node calculations are done (completely, or incompletely, if a terminating condition occurs), depending on the outcome of the calculations, either the sibling is considered next (if this node terminates the descent down this branch) or both child and sibling nodes are entered. If the node indexes h characters and no terminating condition occurs, then h columns of the DP matrix will be calculated, otherwise fewer calculations will take place.

There are two conditions for termination, as outlined in the previous section (score equals threshold or score is too low to reach the threshold), and two additional conditions which result from reaching the end of the DP matrix or reaching a string separator symbol *. Before a column is calculated, we check whether the end of the text matrix is reached or the current symbol is a *. It this is the case, the calculation stops and returns to a sibling (if a sibling exists) or to the parent, otherwise. After each column is evaluated a test is performed. If the threshold has been reached, all leaves originating from the node are traversed and the matching text positions are output. If the score is below threshold, and the conditions for termination due to a low score are met, the calculation on this node stops.

An additional terminating condition can also been added. This condition, stated by Navarro and Baeza-Yates, says that any match at a maximum cost k (assuming unit costs) must start with one of the first $k + 1$ characters of the pattern. In our matching scenario this is rephrased. We consider such children of the root which start with the first

$$queryLength - threshold + 1$$

characters of the query. This is justified for a tree over a large alphabet, like protein, but may not be useful for the DNA alphabet. To test for this condition the initial characters are scanned, and matching takes place only for the children of the root which start with any of those initial characters.

We now present the algorithm in detail. It consists of four methods, shown in Figure 6.7. We discuss the methods, starting with the top entry method, `recursiveApprox`, shown in Method 6.1, and follow on with the method `approx` shown in Method 6.2.
Before we discuss the next procedure `calcNode`, we explain the mechanism of DP matrix evaluation. Our simplified schema leaves out similarity matrices and gaps, but those can be easily incorporated, with some programming effort, and some impact on performance. Further work is necessary, however, to establish what termination criteria have to be used with different similarity matrices, and for different similarity scores used in DNA alignments. The calculation currently used is as follows.

- Look up the value in the cell above, and increment it using the similarity function.
- Look up the value in the cell to the left, and increment it.
- Look up the value in the cell diagonally to the left and above. Compare the current text and pattern characters.
- Take the maximum value of the three values derived above and 0.

recursiveApprox (byte [] x, int threshold)

This procedure is called once, with the appropriate query presented as an array of bytes, and the threshold that has to be reached during the calculation. It performs the following actions.

- It scans the first
$$queryLength - Threshold + 1$$
characters of the query to deliver a set of characters present in this part of the query. This set is returned as an unordered array of bytes.
- The text dimension $textDim$ of the DP array is calculated as
$$textDim = 2 * queryLength - threshold,$$
and the DP array is constructed with size
$$(textDim + 1) * (querylength + 1).$$
- Method `approx` is called, `approx(root,0)`, i.e. with depth equal 0.

Method 6.1: The top level method `recursiveApprox` used in approximate matching.

approx (ThinNode na, int depth)

This is a recursive procedure which descends the tree down to the maximum string depth defined by the DP matrix. The following actions are carried out.

- A comparison is made between the current tree depth and the length of the DP array. If the last column of the array equals tree depth, this terminates this invocation.
- Method `calcNode` is called on the node na. This method returns a Boolean. $True$ is returned if `calcNode` established a termination condition. $False$ is returned if further descent down the tree is warranted.
- If termination condition has been reached, `approx` is invoked on the sibling.
- Otherwise, we check if the end of the DP matrix has been reached. If it has not, `approx` is invoked on the child (using a newly calculated depth) and on the sibling (using the same depth).

Method 6.2: The approx method which recursively traverses the tree

While calculating a DP column we keep track of the maximum score in the column reached so far, and use that value in checking for the termination condition. We also increment the total depth of DP matrix calculated so far, for reporting purposes, so that we can find out how many columns were actually computed. The DP calculation mechanism presented here is used in the method `calcNode`, shown in Method 6.3.

calcNode (ThinNode na, int rightIndex, int depth)

This method is called once for each node visited. It calculates the columns of the DP matrix corresponding to the text characters indexed by the node, and investigates possible termination conditions which may speed up the query evaluation. It delivers matches (either directly or by calling a procedure which traverses leaves `traverseBelow`). Following actions are performed.

- Check if the current node descends directly from the root. If that is the case, terminate if the first character encoded by the node is not in the first initial characters of the query, as previously defined. If the first character encoded by this node is not in the initial query characters, return a termination condition.
- Advance the calculation in the direction of the text, by calculating the matrix for each successive column. Return termination condition, if end of DP array is reached, or a separator or terminator character is encountered. Return termination condition if the threshold is reached (return all matches first by gathering leaves, via calling `traverseBelow`). Return termination condition if there are not enough positions left in the DP matrix to reach the threshold.

Method 6.3: Matrix calculation and the determination of termination conditions.

The method `calcNode` may indentify sequence matches which need to be reported. If the node where a match is found is a leaf, `calcNode` adds the end index of the match to the list of results. If the node is not a leaf, the method `traverseBelow` is called, to gather leaves which descend from the current node, as shown in Method 6.4.

traverseBelow (ThinNode na, int depth, int iIndex)

This recursive method gathers the leaves which match the query.
$iIndex$ records the column of the DP matrix where the match was observed.
On reaching a leaf, the end position of the match is calculated using:

$$matchEnd = thisNode.getLeftLabel() - depth + iIndex.$$

Recursive call on a sibling node is initiated using the same depth value, while for the child the new depth is calculated using

$$newDepth = depth + thisNode.getChild().getLeftLabel() - thisNode.getLeftLabel().$$

Method 6.4: Recursive descent of the tree to gather all matching leaves.

In our tests we simplify the management of results, and just count the hits, in the same way as in our DP benchmark. As is customary in theoretical work on approximate matching, we report the index of the match end and not the starting index for a match. We stop the DP matrix calculation as soon as it reaches the threshold (so we do not know if a match better

than the threshold exists).

6.4.5 Correctness of implementation

We have not constructed a full test showing the exact equivalence between the matrix-based and the suffix tree-based matrix calculation. One of the difficulties of finding the equivalence between the two approaches was already illustrated in Figure 6.5 where we showed that the suffix tree calculation does not reach all of the extensions of a matching suffix which might lead to a score equal to the threshold or greater.

Beside this observation we make here another point. This refers to an opposite scenario where the suffix tree produces more hits than the DP matrix. We illustrate this now. If we calculate the DP matrix for the text $MMSARGDFLN$ and the query $MMSAR$, we observe the score 4 just once, on reaching the text character R, as shown in Table 6.2. In

	M	M	S	A	R	G	D	F	L	N
	0	0	0	0	0	0				
M	0	1	1	0	0	0				
M	0	1	2	0	0	0				
S	0	0	1	3	0	0				
A	0	0	0	2	**4**	0				
R	0	0	0	1	3	5				

Table 6.2: DP calculation for the global alignment of *MMSARGDFLN* and *MMSAR*.

a suffix tree the calculation is different. We get two matches at threshold 4 because we calculate separately for the first and the second suffix (and other remaining suffixes). For instance the second suffix $MSARGD...$, starting at the second character of the text, is used in matching starting with zero values in the zero column of the matrix. As a result, we get local alignments which start anywhere within the text. The first alignment is the same as for the DP matrix, but stops at the fourth text character at threshold 4. An additional match ending at text position 5 is also delivered, based on the second suffix. This situation is portrayed in Table 6.3.

To construct a test of correctness, we would have to evaluate the DP matrix for all of the suffixes which exactly reach the threshold, and then compare that output with what the suffix tree produces. Additionally, suffix extensions would have to be taken into account. Those could be generated by modifying the tree traversal algorithm so that the computation does not stop when the threshold is reached, but carries on to the full depth of $m + k$ (the sum of the query length and the error).

We did not construct a fully automated test but investigated a number of alignments produced by the suffix tree and the DP matrix, and reached the conclusion about the correctness of our code, based on those tests. For instance, we run the comparison of words *surgery* and *survey* and the matches reported by the matrix were the text positions 3, 5, and 7. On running the suffix tree on the same data one match was reported at text position 3 only, as expected, as shown in Table 6.4. The same test run for the text $surgery$ concatenated 4 times, i.e. $surgerysurgerysurgerysurgery$ using the DP matrix reports 12 matches (3 matches on each $surgery$), while the same text compared over the suffix tree reports 4 matches at positions 3, 10, 17 and 24, each relating to the first three letters of the word $surgery$.

		M	M	S	A	R	G
	0	0	0	0	0		
M	0	1	1	0	0		
M	0	1	2	1	0		
S	0	0	1	3	2		
A	0	0	0	2	**4**		
R	0	0	0	1	3		

			M	S	A	R	G
		0	0	0	0	0	
	M	0	1	0	0	0	
	M	0	1	0	0	0	
	S	0	0	2	0	0	
	A	0	0	1	3	2	
	R	0	0	0	2	**4**	

Table 6.3: DP calculation for the local alignment of two text suffixes *MMSAR* and *MSARG* with query *MMSAR*.

		1	2	3	4	5	6	7
		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	0	1	0	0	0	0	0	0
u	0	0	2	1	0	0	0	0
r	0	0	1	**3**	2	1	1	0
v	0	0	0	2	2	1	0	0
e	0	0	0	1	1	**3**	2	1
y	0	0	0	0	0	2	2	**3**

Table 6.4: DP calculation for the local alignment of two suffixes *MMSAR* and *MSARG* with *MMSAR*.

6.4.6 Approximate searching - test overview

A naive approach to using the suffix tree would be to submit each query in turn and wait for the results. Our approximate matching algorithm accepts the query and a threshold. With real queries as used in biology this approach is not feasible, as for a query length of 250 characters, we would probably traverse most of the tree and perform the number of calculations which increases exponentially with the traversed depth, so that the upper limit on the calculation would be some power of 250 (for the protein alphabet 250^{20} is the maximum). To avoid excessive computations we need to break the query into shorter substrings and search for each of them using a predefined threshold. In our scenario we decided to start with query length 5, and threshold 3, and increase those until the number of reported calculated DP columns is larger than if the same calculation were to be performed using a DP matrix.

We report the query, the threshold, the size of the DP matrix used, the number of matches observed, the number of nodes visited, the number of columns calculated, and the time in ms needed to perform each query. A full report of the data produced is reproduced in Appendix D. The output has the format shown in Table 6.5.

query col 1	thresh col 2	DPsize cols 3-5	MatchesReported col 6	nodesSeen col 7	cols col 8	timeMs col 9
MASPS	4	5 x 6	1354	9188	45865	185

Table 6.5: Output format for approximate matching tests.

The semantics of this table are as follows.

1. Column 1 is the query itself.
2. Column 2 is the threshold.
3. Columns 3-5 consist of 2 numbers and the times symbol. They represent the size of the DP matrix calculated for this query. For instance 11 x 19 signifies that the query length was 11 and the text dimension of the array was 19. The formula we used was

$$textDimension = 2 * queryDimension - threshold$$

4. Column 6 is the number of matches recorded.
5. Column 7 is the number of nodes visited in this traversal. The contents of this column may be used in database performance tuning in future work but are not analysed further.
6. Column 8 is the number of matrix columns for which a DP calculation was carried out.
7. Column 9 is the time in ms needed for this traversal. This time measurement is the total elapsed time for the entire query evaluation, and for the persistent tree it contains within it a considerable component of garbage collection time. We had to call the garbage collector using $System.gc()$ Java call after each part of the traversal which found matching leaves. Without this call we were not able to carry out any queries on the large PJama store.

Our data analysis concentrates on three issues.

1. Comparison between the number of DP columns calculated in comparison to the number of column calculations needed for the full matrix evaluation, for text length n. We take the $maximum$ number of columns and calculate what fraction of n this represents.
2. Comparison of the number of results returned for different query lengths and thresholds. If the number of results returned is large, this means that very large portions of disk are scanned, which is inefficient, and that further post-processing of results may not be sufficiently fast. This measurement is based on $average$ values.
3. Comparison of time required for tree traversal with the time needed for the full matrix calculation. We base this on $average$ time reported and calculate the ratio of time in seconds to Mb of sequence in the same way as reported for the DP benchmark.

query	thresh	min	max	avg	max/36mln
5	4	21405	53535	42959	.0003
6	4	3181482	8960148	6045120	.0448
6	5	30180	79446	62425	.0004
7	4	33492781	90720238	60637158	.4536
7	5	3640049	11597537	7548249	.058
7	6	35371	119357	84012	.0006
8	4	108495304	251024912	178571276	1.2551
8	5	39192752	102245968	75437310	.5112
8	6	2922120	13570984	9841472	.0679
8	7	74808	139040	108745	.0007
9	4	191805228	475824366	343272622	2.3791
9	5	95033853	276993117	193006027	1.385
9	6	41098536	105980760	78171406	.5299
9	7	6615036	15241887	10962888	.0762
9	8	83538	185526	128722	.0009
10	4	167366310	769337230	562480802	3.8467
10	5	124155670	540604130	371848620	2.703
10	6	69759770	303568470	210464641	1.5178
10	7	34430920	124350020	89022147	.6218
10	8	5280720	19574720	12701967	.0979
10	9	55320	222570	160751	.0011
11	3	883686199	1506901275	1228614912	7.5345
11	4	588209424	1282432382	927304596	6.4122
11	5	337938843	889693266	643200278	4.4485
11	6	252268302	577024657	419560898	2.8851
11	7	168544189	312397712	242837030	1.562
11	8	79405403	137446474	114782868	.6872
11	9	15113879	20830898	17682355	.1042
11	10	193424	214797	207629	.0011

Table 6.6: The number of matrix columns calculated in the 36 Mb transient tree.

6.4.7 A transient tree for 36 Mb of protein

We constructed a transient suffix tree for the SWISSPROT dataset of 36 Mb. We carried out tests on the Sun E450 machine, as for all other tests described previously, and using the latest version of Java, Java 1.3.2. A set of 40—50 queries was used for each combination of query and threshold, and data summaries are based on a sample of 1425 queries of length between 5 and 11 characters, and thresholds between 4 and 10.

We focus first on the fraction of the DP matrix calculated for different combinations of query and threshold. We report minimum, maximum and average values as well as the ratio of maximum columns to 36 million in Table 6.6.

The picture emerging from this measurement is that by splitting the original query into strings of length 5, 6, 7 or 8, we can efficiently query using error ratios of up to 28% (taking threshold 7 and query length 5, for instance). It appears that low error ratios (one or two letters in a query of 9-12 characters) are handled well by this size of index, but high error ratios are not. These figures characterise a small protein index. We predict that for other alphabets and larger indexes the ratios will be different.

We now turn our attention to the same testing scenario applied to a larger data set (200 Mb, the merged SWISSPROT and TREMBL data).

6.4.8 Approximate matching using a large persistent tree

The same test, but with a smaller batch size was carried out for the persistent tree indexing 200 Mb of protein. The batch size had to be reduced to 20, and several testing runs had to be aborted, as the query processing on this data set was very time consuming. Overall, the total processing time required for this test spanned over a week. We present a summary of the data in Table 6.7.

We observe that for a query of length m and error ratios of $m-1$ and $m-2$ there is always a gain in using the index. In particular for the threshold of $m-1$ ratios of 1/1000 and 1/2000 prevail and those promise a significant speed up in sequence searching. Although these thresholds guarantee good performance, they may not be competitive enough in terms of query sensitivity. For queries of length 5 and 6 the threshold $m-2$ reduced the computation to between 5 and 7% of the text matrix and this might deliver good sensitivity as well as a speed up.

6.4.9 Performance and practicality of this approach

We proceed to analyse our results from two other perspectives. One is the number of matches returned in each case, and the other the time required to query the tree. The number of matches is important because after splitting a long query into many parts we need to process the matches retrieved for each substring query, and the manipulation of those results will constitute a significant overhead. On the other hand, the time needed to query the data will decide if this approach is competitive. Our measurement data reflect directly on the performance of PJama, and are indicative of the overhead which we will incur by using a persistent data structure as compared with a transient one which fits completely in RAM.

The measurements we produced were taken using CPUs which are now 3 years' old. As the DP matrix calculation is CPU bound in the *transient context*, the speed of matrix calculation will improve from the one we report if a faster processor is used. On the other

query	thresh	min	max	avg	max/200mln
5	3	6436675	9481975	8249977	.0474
5	4	42950	51310	47598	.0003
6	4	8956248	13943562	11641847	.0697
6	5	62592	87888	77719	.0004
7	5	9659566	16140761	14263059	.0807
7	6	81907	122206	107137	.0006
8	6	10457160	21760960	18001259	.1088
8	7	76776	146472	124644	.0007
9	6	190771353	283692348	246872682	1.4185
9	7	13089069	25485219	19755152	.1274
9	8	77760	192555	150067	.001
10	7	204715200	337762230	283422668	1.6888
10	8	15573770	32118120	23200178	.1606
10	9	161180	243360	199979	.0012
11	7	564852574	808146713	723471613	4.0407
11	8	232200210	371462817	301077707	1.8573
11	9	17369022	36604612	28906699	.183
11	10	126247	268433	230633	.0013
12	9	197337984	409506576	336096871	2.0475
12	10	28309116	40709484	34184200	.2035
12	11	234804	326556	285869	.0016
13	9	758829344	1321559070	974136247	6.6078
13	10	238134416	481226798	363482077	2.4061
13	11	25080224	45218784	35699910	.2261
13	12	174850	435175	316965	.0022
14	12	34398056	52736376	42116508	.2637
14	13	291256	485884	378641	.0024
15	12	546638490	546638490	546638490	2.7332
15	13	36644295	60233355	50801498	.3012
15	14	371685	517530	448736	.0026
16	13	415554432	622350800	510104411	3.1118
16	14	40067808	63316144	54075435	.3166
16	15	262656	581600	448268	.0029

Table 6.7: The number of matrix columns calculated in the 200 Mb persistent tree.

hand, the retrieval of tree nodes in a *persistent suffix tree* is disk bound, and in this context a leaner tree structure and more effective database technology could improve performance.

6.4.10 Performance - number of matches reported

We present the number of matches reported for the transient tree and the persistent tree. Last column in Table 6.8 shows a calculated value of the expected number of results returned for a query of length 300, based on the *average* number of results reported. This was calculated as follows:

$$matches\ per\ query\ of\ 300\ chars = avg * (300/length\ of query).$$

Our data show clearly that some combinations of query length and threshold in the index for 36 Mb return very large results sets. In particular, for a query of length 11 and threshold 3 the number of results returned, 14 mln, corresponds to 39% of the queried data set. Even after taking into account the fact that the suffix tree returns the same suffix number several times, it clearly does not make sense to use this threshold with longer queries. Similarly for the threshold 4, query lengths 8 to 11 produce very large result sets. To get a better understanding of the relationship between the query length, threshold and the result set, we now present the same measurement for the index to 200 Mb of protein. Table 6.9 summarises the results.
It appears that the combinations of query length 5 and threshold 3 and query length 6 and threshold 4 return too many results to be practical. The suffix tree probably returns the same result up to three times in both cases (depending on how many of the direct children of the root are considered in the traversal), but even after adjustment for possible triple reporting the result set appears to be too large to handle efficiently. Other combinations of query length and threshold return manageable result sets and seem to be appropriate for our purposes. As the query becomes longer, higher thresholds lower the similarity between the text and the query that can be detected using this method.

6.4.11 Performance - timing

We now focus on the time needed to execute the query. Similarly to the average result set expected for a query of 300 characters, we use *average* query time to predict typical evaluation times. An average execution time calculated in this way for a query of length 300 is shown in column *avg 300 chars*. We also show the ratio of time (secs) to DP matrix size (Mb) calculated based on the product of query length and text length and the *average* time. We first present a data summary for the transient index for 36 Mb, see Table 6.10.
In this measurement for the transient suffix tree index it appears that all combinations of query length and threshold offer an improvement over the calculation of the full DP matrix. Our algorithm does not filter results to remove duplicates or build full alignments, so additional time is required to carry out those operations. On the other hand, the DP matrix does not output local alignments but only the global ones, so that additional work is needed to produce those, as detailed in [231]. Clearly, thresholds of $m-1$ offer excellent performance in all cases.

We present our results for the large persistent tree in Table 6.11. In the case of a persistent tree, as expected, performance is slower. This is due to some extent to the larger size of the index, and more importantly to the fact that data are retrieved from disk. It appears

query length	thresh	min hits	max hits	avg hits	expected matches for a query of 300 chars
5	4	28	3403	896	53759
5	5	0	151	24	1465
6	4	167	6647	2048	102398
6	5	0	239	53	2642
6	6	0	11	2	101
7	4	420	12600	3888	166649
7	5	7	688	143	6141
7	6	0	50	7	288
7	7	0	5	1	47
8	4	532	22973	6035	226300
8	5	18	1481	244	9165
8	6	0	119	12	449
8	7	0	12	2	73
9	4	1732	32131	9354	311799
9	5	16	2716	408	13584
9	6	0	212	20	666
9	7	0	26	3	109
9	8	0	6	1	46
9	9	0	2	0	16
10	4	1446	43000	13488	404628
10	5	40	2744	633	18982
10	6	0	277	33	990
10	7	0	27	5	143
10	8	0	10	2	67
10	9	0	4	1	33
11	3	279711	732887	516487	14086021
11	4	13353	44715	27134	740014
11	5	542	2506	1377	37555
11	6	30	116	62	1677
11	7	9	31	15	400
11	8	6	16	10	268
11	9	4	8	6	164
11	10	1	4	3	86

Table 6.8: Number of matches observed in the 36 Mb transient tree.

query length	thresh	min hits	max hits	avg hits	expected matches for a query of 300 chars
5	3	86871	551218	260176	15610540
5	4	218	17526	7449	446953
6	4	1444	31015	13585	679267
6	5	9	739	363	18150
7	5	64	3798	1232	52781
7	6	5	169	40	1719
8	6	16	108	56	2088
8	7	0	19	7	258
9	6	26	112	63	2089
9	7	4	47	12	406
9	8	2	9	4	136
10	7	7	61	23	687
10	8	3	18	9	273
10	9	1	12	5	147
11	7	7	31	18	491
11	8	4	14	9	245
11	9	2	10	6	164
11	10	0	6	2	66
11	11	0	3	1	36
12	9	6	33	17	436
12	10	2	22	10	244
12	11	1	11	5	128
12	12	0	3	2	44
13	9	12	41	21	490
13	10	9	26	15	341
13	11	3	14	8	185
13	12	1	9	4	87
13	13	1	3	2	46
14	12	1	12	6	133
14	13	0	6	3	62
15	12	9	9	9	180
15	13	2	12	7	136
15	14	1	6	3	60
16	13	2	19	11	198
16	14	0	16	7	135
16	15	0	6	3	65

Table 6.9: Number of matches reported in the 200 Mb persistent tree.

query	thresh	min	max	avg	avg 300 chars	time (sec)/DP size (Mb)
5	4	42	175	89	5346	.0001
5	5	1	19	2	117	0
6	4	5732	16116	10869	543447	.0091
6	5	54	143	111	5537	.0001
6	6	0	2	1	65	0
7	4	47112	124548	83872	3594513	.0599
7	5	5996	19783	12734	545756	.0091
7	6	60	197	141	6036	.0001
7	7	1	3	2	70	0
8	4	121255	283049	200358	7513423	.1252
8	5	51330	134168	98365	3688685	.0615
8	6	4439	21566	15459	579707	.0097
8	7	117	214	168	6309	.0001
9	4	196316	523115	354320	11810657	.1968
9	5	103649	308100	211009	7033630	.1172
9	6	52051	134134	98294	3276467	.0546
9	7	9759	22828	16402	546735	.0091
9	8	121	272	189	6307	.0001
9	9	1	5	2	77	0
10	4	163226	835883	563602	16908063	.2818
10	5	124304	557265	379482	11384457	.1897
10	6	73123	356037	227065	6811953	.1135
10	7	41526	170960	108653	3259584	.0543
10	8	7385	27516	17774	533213	.0089
10	9	95	317	230	6885	.0001
11	3	825973	1412062	1151716	31410444	.5235
11	4	555693	1211841	874742	23856595	.3976
11	5	325368	855454	617572	16842868	.2807
11	6	248479	569079	413156	11267891	.1878
11	7	173910	323956	250632	6835405	.1139
11	8	92343	158147	132290	3607914	.0601
11	9	22927	28407	24405	665582	.0111
11	10	261	392	297	8100	.0001

Table 6.10: Query times (ms) in the transient index for 36 Mb.

query	thresh	min	max	avg	avg 300 chars	time (sec)/DP size (Mb)
5	3	1450560	2583183	1853556	111213340	1.8536
5	4	4151	33372	18650	1119000	.0187
6	4	517083	2070834	1541020	77051017	1.2842
6	5	1685	67656	21380	1069000	.0178
7	5	1139873	1701941	1393505	59721652	.9954
7	6	2052	63256	25098	1075629	.0179
8	6	660330	2266510	1563657	58637154	.9773
8	7	1798	42883	16450	**616887**	**.0103**
9	6	3362900	4306590	3874534	129151122	2.1525
9	7	1201663	2016994	1551996	51733192	.8622
9	8	3047	59940	24716	823856	.0137
10	7	3194328	5986231	4574149	137224470	2.2871
10	8	1140265	2570789	1957363	58720897	.9787
10	9	44954	185122	94972	2849157	.0475
11	7	4675691	6276299	5629039	153519245	2.5587
11	8	3346721	4975578	4300113	117275800	1.9546
11	9	1267862	2629181	1768264	48225389	.8038
11	10	3078	185881	49093	1338902	.0223
11	11	2	126	58	1594	0
12	9	2379287	5506253	4480121	112003033	1.8667
12	10	1312396	2793171	2047009	51175214	.8529
12	11	4866	185079	84540	2113500	.0352
12	12	5	302	90	2253	0
13	9	5511472	10047181	6975204	160966254	2.6828
13	10	3773371	6913588	4742562	109443736	1.8241
13	11	1472837	2795986	2058922	47513587	.7919
13	12	71293	150758	104202	2404672	.0401
13	13	49	103	72	1667	0
14	12	947869	1891285	1524030	32657793	.5443
14	13	88479	184132	141211	3025950	.0504
15	12	5111573	5111573	5111573	102231460	1.7039
15	13	1325548	2176277	1704810	34096198	.5683
15	14	68562	188278	140532	2810648	.0468
16	13	4517261	7173091	5225156	97971683	1.6329
16	14	1635917	2724093	2181954	40911642	.6819
16	15	115815	289287	193278	3623954	.0604

Table 6.11: Query times (ms) in the persistent index for 200 Mb.

that using this combination of technology and our algorithm, our method offers a speed up for all thresholds of $m-1$ within the range of queries we investigated. Limited speed up is also achieved for some thresholds equal $m-2$, but the performance is not satisfactory. Further work at the level of database performance and tree structure is needed to improve performance, so that using the threshold of $m-2$ becomes viable.

Some of the values we present compare favourably with the values calculated for the DP matrix where the equation for the graph of time against matrix size was

$$time\,(seconds) = 1.16 * matrixSize\,(Mb) + 66.0$$

For instance taking queries of length 8 at threshold 7, shown in bold, and using this combination to evaluate a query of 600 AAs, we expect the suffix tree to deliver the hits within

$$617 * 2 = 1234\,(seconds)$$

while the full matrix calculation would require

$$1.16 * 600 * 200 + 66 = 139266\,(seconds).$$

This means that the suffix tree is around 100 times faster.

6.5 BLAST benchmark

We describe one algorithm which is specifically tuned to biological use. A direct comparison of our method with known sequence comparison tools does not provide a useful benchmark yet. To achieve the functionality presented by the software we tested requires further research and engineering.

6.5.1 BLAST

Basic local alignment search tool BLAST [7] is a sequence comparison tool which computes alignments faster than the exhaustive-search algorithm of Smith and Waterman [203] or the heuristics-based FASTA program designed by Pearson and Lipman [176]. A further paper explaining an improved version of BLAST (with application to proteins) was published in 1997 [8] and a related faster version for DNA comparison called MegaBlast [239] is also available now. BLAST can be downloaded from ftp://ncbi.nlm.nih.gov/blast/executable/ and installed locally, but then a local mirror of data has to be kept up to date. A full discussion of BLAST is not possible here, but we want to single out some of the features which are relevant in this context.

BLAST assumes that all sequence data reside in memory during program execution, and that additional memory is available to hold the query and associated data structures needed for computation. Currently Compaq Alphas [226, 57] are the preferred hardware platform used by both NCBI in the US http://www.ncbi.nlm.nih.gov/, and by the Sanger Centre in the UK, http://www.sanger.ac.uk. Farms of computers (400 at the Sanger Centre) serve the incoming BLAST requests, and load sharing software is used to serve the requests coming form the web. Current volumes of sequence data can still be accommodated by the existing computer farms, but the speed of query processing becomes low around noon when American users become active, and subsides with the end of the American working day

(Keith Johnson, personal communication). Our collaborators in genetics have developed work strategies to cope with those supply fluctuations, as prolonged waiting for query results has a bad impact on the efficiency of their work.

BLAST keeps sequence data in a compressed form and concatenates multiple sequences into one file. A program $formatdb$ is run to format the sequences which are concatenated, compressed, and filtered. A set of indexes is created, which indexes sequence identifiers, repetitive sequences, and sequence end/start positions in the so-called database file. This "database" can be queried with either a single sequence or a file containing many sequences in the FASTA format [176] (containing a sequence identifier in the first line, starting with the $>$ sign, and remaining lines containing plain sequence).
The query algorithm has the following components which we outline below.

- The query string is compressed, both in the case of DNA and proteins.
- DNA is filtered for repetitive sequences.
- Protein queries are scanned with a window of a given size to create an index of the query text. For each unique window alternative (mutated) protein strings are generated which are similar to the given window and this similarity is bounded by a threshold T which has been established using simulation techniques. Usually for each character in the query, some 50 such words are generated, so for a query of 250 AAs 12,500 words would be produced. This list can be generated in time proportional to the length of the list. A deterministic finite automaton (DFA) encoding all the words in the list is produced and used to scan the "database" text. This is a Mealy automaton which signals text acceptance on transitions between states. In 1990 this automaton could process 500,000 characters per second.
- A DNA query does not use mutated sequences but exact matching on substrings of the original query. The word length of 12 is commonly used, and all contiguous windows of length 12 are created, without mutations, producing $n - w + 1$ words where n is the length of the query and w window size. The comparison of text and query is done by byte-wise scanning. If there is a hit over the whole byte, extending this hit to the enclosing window, and further into next windows is attempted.
- All matches are then extended in both directions until the score falls a certain distance below the best score found for shorter extensions.

Additional refinements for DNA include filtering of repetitive regions and removal of such regions from the list, prior to the comparison operation. Current cost models for DNA include integers in the range of 1 to 5, where matches are scored positively, and mismatches bear a negative cost.

In protein matching several cost matrices can be used, and several models of scoring and gap costs are available. BLAST provides automatic translation between DNA and protein and different program invocations have to be made depending on the type of sequence used in the query and the target (biologists usually know which version of the BLAST program they want to use). To accompany this complexity, some 20 parameters can be set to define the behaviour of the program, for instance by reporting hits which are closer to the query. The underlying filtering of results and the stringency of matching rely on a complex framework based on statistical considerations which are beyond the scope of our work.

We now report the testing results which we produced using BLAST. We used the TREMBL dataset of proteins (146 Mb) and the human proteins from the Ensembl dataset as queries. We first formatted the "database" which took 5 min 43.33 s. We reproduce the formatdb command and the time recorded

```
mars{ela}99: time /local/pj_test_n2/BLASTsoftware/formatdb
-t trembl -i tremblFASTA -l formatTremblLog -p T -o T

213.97u 9.22s 5:43.33 65.0%
```

Subsequently we submitted a file containing all of the Ensembl genes, and aborted the computation after 21 hrs 56 min 10.82 s. In that time 4,051 queries have been processed and the resulting output file reached the size of 1,189,638,445 bytes. This corresponds to an average time per query around 30 s. We invoked BLAST as follows, and aborted it as shown:

```
neptune{ela}37: time ../BLASTsoftware/blastall
-p blastp -a4 -dtremblFASTA -i ../SEQUENCE/HUMAN/ensembl.pep
-e10 -o BLASTpRESULT

^C-172866.-23u 1783.12s 21:56:10.82 0.9%
```

The parameters were chosen as follows.

- $-a4$ instructs the software to use all 4 processors available.
- $-e$ refers to the expected random 10 hits for every query sequence submitted.
- $-pblastp$ specifies a comparison between two proteins
- $-d$ and $-o$ specify the "database" and the output file.

The total length of sequence in 4,051 queries was 2,268,482 characters. This yields the average query length of 560 characters. The total size equivalent (product of database length and sequence length) is then

$$2,268,482 * 145,703,122 = 330,524,909,600,804.$$

This is equivalent to 330,524,909,600 Mb of matrix calculation in 78,971 seconds, i.e. an approximate slope of

$$2.39 * 10^{-7}.$$

In comparison, the full DP matrix calculation which we reported, carried out using Java, had the slope of 1.16 and the intercept of 66 for the same units. The speed of BLAST is due to the careful design of the algorithm which avoids the full matrix calculation, and to its efficient implementation.

BLAST uses heuristics. The choice of the word length to be used in DNA scanning and protein query partitioning is governed by experience, and with smaller word size, performance degrades. The choice of threshold for generating mutated protein strings relies on simulation results. In practice, parameters which maximise performance have a negative impact on the sensitivity of string comparison.

Statistics used by BLAST, which are outside the scope of our research, are based on the size of the "database". Until quite recently that meant that short matches which are significant in comparison of small genomes were never reported, because those small genomes were part of large microbial "databases". Now, an additional menu option at NCBI is provided, for searching for hits against small data sets and with short strings.

Finally, user control over the size of BLAST output is limited. Different options often have to be tried to identify the appropriate level of reporting, by setting the expected number of matches which could arise randomly. In practice, in cross-genome context, parameters are set to allow for the reporting of all matches, and then post-processing using MSPcrunch [205] is done which filters for instance the top 50 matches. In our experiment with 7.7 Mbp of DNA from one species and 1.2 Mb of DNA from another species, BLAST took 6 days to evaluate the query, and at some point the file size limit of 2 GB for the output file was reached. This resulted in a truncated result file, and the postprocessing with MSPcrunch consumed another day.

6.6 Evaluation of results with respect to benchmarks

In comparison to BLAST our technology is not competitive. However, in comparison to our DP matrix benchmark which carries out a full DP matrix calculation, we observe significant speed ups. We can also provide the advantage of exhaustive searching, instead of heuristics.

We are limited by two factors. One of them is our data structure, which needs further optimisation. Secondly, the underlying database technology imposes high space overheads on every node on the index. Based on our measurement of the number of DP columns that we calculate, we believe that further refinement of the suffix tree searching mechanism is warranted. Further data which we gathered, in particular the information about the number of nodes accessed, will help in identifying appropriate optimisations of both the data structure and alternative caching strategies to be used in approximate searching.

6.7 Summary

We presented our new method of comparing the performance of indexed string matching with its counterpart which does not use indexing. Using this methodology we showed that suffix trees reduce the size of the string comparison problem significantly, and the indexing gain improves with the size of the index. We started by explaining the approximate matching algorithm, then proceeded to the analysis of results, which we then compared with the unindexed string matching and BLAST. We now move on to the presentation of Conclusions and further work.

Chapter 7

Conclusions and further work

This thesis presents three major contributions in the area of bioinformatics. The first contribution is the recognition of the need for new database techniques in biological data processing. The second contribution is a new methodology which combines persistence with the investigation of suffix indexing structures and their properties. This methodology resulted in a practical algorithm for the construction of suffix trees in excess of RAM, which has hitherto not been possible. The third contribution is a new approach to the evaluation of the indexing gain achieved by using a suffix tree in combination with a dynamic programming algorithm, as used in approximate matching of biological sequences.

7.1 Developing the scope of our research

The research into string indexing for faster sequence comparison follows on from our earlier experience in database management of genetics data [236, 213, 122], also see Appendix A, and from our research identifying the need for special support for large scale biological analysis [183, 184]. In Chapter 2 we outlined current directions in bioinformatics and the trend to do more data intensive biological investigations. In this section we close with a longer term perspective.

In 1996 genome databases and maps were beginning to appear on the internet, coinciding with plans to sequence the human genome. At that point Oracle[1] was beginning to introduce object-oriented features into the new version of its relational product, Oracle 8i. Most biological "databases" relied on ACeDB [70] and each of them had a dedicated data curator responsible for data management, including formatting the data submitted by the collaborating labs, reading in the data into the "database", preparation of queries, and other chores. Using ACeDB was not only labour intensive and error prone, but resulted in poor web access to data and genetic maps. The amount of available genomic sequence was not large initially, and local BLAST engines could cope well with the global volume of queries they were being subjected to. As it became possible to build genome maps of higher resolution, for instance by large-scale hybridisation experiments which are a pre-cursor of microarrays, ACeDB turned out *not* to provide adequate query support, and we implemented a genome mapping system using Oracle. However, we could not perform sequence comparison tasks within this system, and several thousand BLAST queries had to be submitted via email to an external BLAST server. Oracle enabled us to manipulate mapping data and

[1] http://www/oracle.com

to finally present it on the web (using OPM [51, 173, 50], see http://chr21.molgen.mpg.de), but significant human intervention was needed (software was written to submit a batch of requests and a trained biologist had to analyse the BLAST output) to select a subset of BLAST results for inclusion in the map. We found this approach to be cumbersome and hard to manage, as the BLAST searches had to be re-submitted and re-examined to find sequence matches in the constantly growing body of known DNA sequences. The database aspects of this work are described in Appendix A. Our main discovery that automation of large-scale biological research was needed remained unpublished.

Our further work in the context of linkage analysis and mutation modelling [183, 184] deepened our understanding of the need for data processing automation but our grant proposal to the Biotechnology and Biological Science Research Council (BBSRC) in this area was not understood then as being of significant potential in biology. We seem to have done work ahead of the recognised need for this research which is now coming to the fore under various guises, including the GRID [31, 85]. Subsequently, we focused on a subproblem which we identified, that is providing indexed access to sequence data within a database system. This work combined the issues of large scale data processing with the possibility of automation of query evaluation and integration with a database, forming the core of this thesis. We realise that beside this narrow field of research there are further issues in the field of large scale data processing which are gradually being addressed by many research groups worldwide [208, 80]. It is very likely that future solutions will include agent technologies, ontologies and derivatives of the abstract specification language called eXtensible Markup Language (XML) [3, 111].

We did not investigate the workflow problem further but turned to the nature of the data itself. We surveyed new biological data types and new data acquisition technologies which give rise to large scale data collections. In Chapter 2 we introduce this theme with high-level overviews of the sequencing technology, hybridisation techniques, and protein analysis techniques. These methods produce thousands of data points in parallel and are not supported with adequate data management and analysis techniques which would give the biologist full mastery of their data. With the new technologies data acquisition is now cheaper and faster, but data manipulation presents significant challenges. In particular, none of the new data types we described in Chapter 2 are directly supported by database systems, or can take advantage of indexing technologies which are known to speed up access to large data repositories. This initial investigation of data processing in the biological domain led to the focus of our work on solving the data indexing problem just for one of the new data types — the sequence data. We developed a prototype indexing solution. We developed an algorithm which breaks the previous limit on the size of a suffix tree index to sequence, which used to be the RAM size, and showed that we can now build suffix trees on disk significantly in excess of RAM size. We also implemented an approximate matching algorithm using the suffix tree index, and showed what the indexing gain is, for a dataset of 200 Mb of protein. We also showed that the indexing gain increases with index size, by comparing the indexing gain for two indexes of different sizes. We now revisit briefly the research issues which create the backdrop to our work and position our findings against this background.

7.2 The biological data processing scene

We claim that indexing technologies can be used to provide faster access to the volumes of biological data which are growing rapidly, and we demonstrate the value of indexing for protein data sets. We are aware that the requirement to describe and index all experimental data for future use, and make them available to other scientists is in most cases not being met, with the exception of few publicly funded data collections, mostly containing sequence data. The majority of data reside on hard disks of private computers, are backed up to CDROM only, and are not annotated. Data reside on electronic media which may be hard to find, as they reside in filing cabinets, and their location, the description of data, protocols used to produce it, and the way they were evaluated is recorded only in lab books. By keeping data separate from annotation, biologists may forego the opportunity of observing data relationships and discrepancies that could be examined, had all the data been available in one system which they could query using a variety of search criteria.

Current indexing technologies for biological data have a limited scope. Database indexing is available for word structured alphanumeric data. Short space delimited textual fields are stored as the VARCHAR data type, of limited length. Sequence data are not indexed, and they attracted our attention because the volume of data is increasing faster than our ability to search it[2]. To enable text searching of sequence data, the BLAST suite of programs is used [7, 8]. BLAST assumes that all sequence data are present in memory, and carries out a serial scan of all data to identify potential hits which are then aligned with the query. Because sequential scanning of the text requires large computing resources (Genbank currently provides access to 18 Gbp of DNA sequence), biologists have to wait for their queries to be processed, and cannot submit large queries which would let them compare two genomes. After sequence matches are found by BLAST, combining the results of a BLAST search with other data residing in a database system or in files is not easy, and requires significant programming effort, so in practice this is done by a minority of researchers only. A hybrid solution to this problem was proposed in our work on Chromosome 21 mapping (Appendix A) and we found that this solution had a high associated labour cost for the biologist and the programmer. The current state of the art in sequence similarity searching results in two data universes. One of them is the genomic context universe exemplified by the Ensembl web site[3], where maps of genes positioned on human chromosomes are shown, based on a relational representation of data (using MySQL). The other universe is the sequence comparison (or alignment) world, i.e. BLAST using a web or email interface, where hits may be shown in different colours based on sequence similarity, but where full contextual visualisation of the hits, or comparison with other relevant data cannot be easily made. In practice, for large-scale sequence comparison tasks this involves a two-level interaction. First, query sequences are submitted to an external BLAST server, then manual data selection is made where a subset of the results is chosen, then the selected results are fed into a spreadsheet, and possibly finally transferred to a database where a new map could be computed. Facilities for selecting subsets of BLAST matches using logical criteria do not exist yet, although the biological knowledge needed to sift the relevant hits from bad ones exists, and could be made explicit as a set of rules to be executed by a database engine. However, if indexing of sequence data becomes available as part of database technology, and therefore executed

[2] See the graphs on the NCBI web site with a picture of exponential growth in sequence data volumes, http://www.ncbi.nlm.nih.gov/Database/.

[3] http://www.ensembl.org

more efficiently than in the current scenario of scanning all of the input text, then local evaluation of sequence similarity queries can contribute to an integrated data management approach. This vision of integrating sequence searching with other data operations needed in genome map construction and protein analysis is one of our contributions. We now describe further data indexing challenges, some of which we may be able to address in the future.

We see the following data types as presenting further challenges to the database community.

- Microarray data which require solutions for data storage, annotation and analysis. The volume of data produced by microarray experiments is large, and data are stored without proper annotation which would make them usable in the future or by other researchers. This problem was investigated by a recent student project [95, 12] and remains a focus of significant research activity, with more than 1000 papers published this year. However, most of the current research refers to data analysis of small data sets and does not address data storage and annotation problems.

- Protein datasets, including 2D gels and other protein abundance and interaction assays also need an adequate database solution, if they are to be easily used along other data. A preliminary exploration of this subject was undertaken [130] and resulted in the identification of several issues including annotation, data analysis and integration with other data sets. Current practice for both microarray and protein data includes CDROM storage without annotation, and, usually, no archival or public release policy. As both types of data result in large images, the issue of database technologies to support storage and analysis of such data is a priority.

- Protein structure data. These are deposited in the Protein Data Bank (PDB)[4], in ASCII format, and are searched online, in the same way as sequence data. With the growth in the volume of structural data, algorithms for faster searching in protein structures are needed, so that partially resolved structures can be aligned with fully resolved ones, and 3D motifs can be found [227]. Current protein indexes are small, and can be held in memory. We expect that such indexes could be stored persistently on disk.

- Alignment data and associated phylogenetic trees. Only a few attempts so far have been made to explore this issue, and result in databases of alignments including SYSTERS[5], COGS[6], and HOMSTRAD[7]. With the growing interest in building "the tree of life", and existing databases of protein families, the requirement to search and adequately visualise such datasets is becoming very important. New indexing technologies will be needed here, and new visualisation techniques are required, as current solutions, particularly in the area of data visualisation, are highly unsatisfactory.

- Genome comparisons between species. Current ways of storing BLAST results in very large flat files and postprocessing them using the scan of the entire file are not efficient or flexible [205]. Database indexing of such data would provide better access to genome-level comparisons which result in very large results sets. Visualisation

[4] http://www.rcsb.org/pdb/
[5] http://systers.molgen.mpg.de/
[6] http://www.ncbi.nlm.nih.gov/COG/
[7] http://www-cryst.bioc.cam.ac.uk/data/align/

of genome comparisons for 2 genomes is inadequate, see http://www.sanger.ac.uk/-Software/ACT/, and needs tools which offer greater flexibility to the biologist. Both database and visualisation techniques can improve access to such data sets, both in terms of their management and analysis.

- Image data from biological experiments. Biological image indexing is currently limited to textual descriptions of the image. However, research in this area [174] might lead to new techniques which would allow for the indexing of a wide variety of images - including those produced in genetics research (for instance images of protein expression *in-situ* [32]) using techniques which summarise the image itself [8].

This background of unsolved database issues demonstrates that database research in the field of bioinformatics has the potential to make data more readily available and therefore more useful to researchers. With access to more varied data, the biologists will be empowered to formulate new hypotheses and develop theories supported by a wider set of experimental results than currently possible. We now return to other contributions made by this research, that is the creation of very large suffix trees and the measurement of the indexing gain produced by a suffix index.

7.3 Construction of large suffix trees

We are the first to report the construction of large suffix trees on disk, in excess of RAM size, which solves a problem identified as "insoluble" by Baeza-Yates and Navarro [25], and which focused their attention on the suffix array data structure instead.

The special property of reference sequence collections which we recognised and exploited is the slow rate of change in the datasets we encounter. Although the volume of data seems to be growing exponentially[9], changes to genome data are limited to the genomes which are currently being sequenced. As soon as a genome is finished, the data are frozen. For the organisms which are being sequenced, data are released every few months. This leads us to believe that a special variation of database technology is called for. We are looking for indexing structures which can be built once and can be replaced if a new data release is made. This implies that update efficiency is not essential, and the time to create an index is not a primary consideration. This remains in sharp contrast to the usual database scenario of concurrent updates to data which might require complex strategies to ensure transactional correctness of updates as well as speed. Our contribution in this area is to show practically how large suffix trees can be built, under those assumptions.

We developed an algorithm which allows us to build very large suffix trees, by partitioning the tree and building partitions one by one. Given a known memory size, we can partition the tree, so that a partition, or a number of partitions fit completely in memory, and are then committed to disk. A subtree or subtrees once written to disk do not need to be updated again. This opens up new mechanisms of distributed suffix tree construction and use, and allows for the distribution of a very large tree between many processors. Our work presents an alternative solution to the one proposed in theoretical terms by Galil and Apostolico [14], which required as many parallel processors as the number of tree leaves,

[8] See for instance http://www.caip.rutgers.edu/ comanici/jretrieval.html where image searching is based on a similarity metric between the image and the query

[9] http://www.ncbi.nlm.nih.gov/

and built the tree from leaves towards the root, to then reverse the pointers to point from the root towards the nodes. To the best of our knowledge, their theoretical development was never subjected to experimentation, whereas our approach has led to the construction of disk resident indexes for up to 300 Mb of string data. These are the largest suffix trees ever reported, and our results have now been published [119].

Interesting questions arise from this work. It came as a great surprise that the $O(n^2)$ worst-case construction algorithm performs in practice as well as the $O(n)$ one, for the range of index sizes that we could test on our hardware. This discovery reflects the need for caution in using the worst-case complexity as a performance predictor. It also demonstrates the need for a different analysis of index-building algorithms in terms of a more realistic computational model than a flat memory model where each memory location can be read and written equally rapidly, and space complexity does not impact time complexity, a difficult modelling challenge in its own right. The memory hierarchy that we deal with, and which consists of caches at different levels, as well as RAM and virtual memory, is hard to model using known theoretical approaches. We would like to be able to use more accurate models to predict the performance of index-building algorithms.

The actual possibility of tree building and querying in parallel is quite exciting, and might improve the performance of BLAST. BLAST uses a filtering approach followed by matrix calculation, and can be easily parallelised. The same is true for the large suffix tree which can be split into sub-parts. If we replaced BLAST filtering with suffix tree indexing, we would combine good performance with the guarantee of delivering all relevant hits. This means that we could expect speed ups typical of other parallel database technologies exploiting indexes, and may eventually achieve a faster implementation of BLAST while offering the guarantee of finding all relevant sequence matches.

Our work leads directly to several avenues of future investigation which we discuss in Section 7.5. We believe that further work might use either the large suffix tree or its derivatives to deliver more efficient sequence comparison systems than are currently in place.

7.4 Approximate string matching using a suffix tree index

Our contribution lies in adapting a known approximate matching algorithm [25] to the biological context and in rigorous measurement of the indexing gain achieved for large sequence sets, i.e. 200 Mb of protein. Our further work with DNA indexes for 286 Mbp of sequence was submitted to VLDB Journal as an invited paper, and is reproduced in Appendix B. We are the first to clearly show the indexing gain component of this algorithm, and to separate this gain from other performance factors related to the data structure, persistence mechanism, and the DP calculation method. For instance in [25], where the suffix array data structure is used, the authors selected to present an overall performance measure which combines the effects of the data structure, persistence, matrix calculation method, and the indexing gain, so that no clear statement about the significance of using an index can be made. As their index was small (10 Mbp of DNA), the indexing gain would have been less significant than the efficient DP matrix evaluation. Our research makes an important step in justifying the use of indexing for biological data, and we show that with the growth of the index the indexing gain increases.

We adapted the algorithm published by Navarro and Baeza-Yates [25] to our suffix tree, and made adjustments which reflect the similarity metric required in sequence comparison.

We argued why some of the techniques used by the authors are not suitable in the context of sequence comparison, and we presented a significant body of testing results which characterise the behaviour of a protein index for most of the currently available protein data. We tested a range of query lengths and similarity thresholds using a simplified unary cost model, and showed significant potential of a suffix tree index in this context. Using a benchmark we constructed, we showed that even within the limitations of our prototype system, we can demonstrate efficiency gains resulting from indexing. Our results indicate that indexing is beneficial. It is also clear that we need to produce a more efficient implementation of the suffix tree so that we can speed up sequence comparison for a wider range of query lengths and thresholds. This proves that further work in this direction is warranted.

The novelty of our approach in this part of the work was the clear delineation of two issues. The first one is the speed up resulting from the use of an index, and the second one the performance aspect which relates to several other implementation factors. Our findings on the usefulness of indexing are relevant in most suffix indexing contexts, and are independent of the actual suffix index used, that is they apply to a suffix array, a balanced suffix binary search tree, or a suffix tree, as the same indexing gain will apply in all cases where the top-down traversal method is used.

7.5 Further work

We draw two pictures here. The first one is an overview of a wider scope of computing research that we believe could follow on from our bioinformatics work, and the second is a focused development of our techniques which might lead to a product useful to biologists. We present those two perspectives now, starting with the overview.

7.5.1 A conceptual view

We see our work as being part of a wider trend of indexing research. In the most narrow sense, we are trying to extend database techniques to new data types. We now have a working prototype for biological sequence data. We believe that similar techniques (but possibly other indexing structures) could be tested in the context of protein structure searching, phylogenetic tree indexing, alignment indexing, microarray and protein gel indexing, and gene and protein interaction networks. Outside the biological domain, our research is related to the search for indexing structures for semistructured data [58, 211] and other data types which are hard to represent as short strings or numbers, for instance large repositories of astronomy data, see http://research.microsoft.com/ Gray/. Indexing based on explicit data content is just one possibility. Alternatives include indexing based on data distribution and clustering, i.e. statistical and data mining techniques which we did not explore, as demonstrated in [131] where a wavelet transform is used to build an index over biological sequence data.

In a broader context our research will help in providing a more unified view of data and human interaction with data. By adding persistence techniques to sequence analysis we achieve not only more efficient processing at the level of the CPU, but also on the human level. Our aim is to provide a higher level view of biological data processing and enable automation of manual tasks. This implies on the one hand an understanding of what a typical user interaction is, and on the other capturing the knowledge required to automate this interaction. We analysed the set of interactions required in sequence analysis, and see

the possibility of encoding this task in a computer-readable format so that in the future this work can be made less error-prone and require less human intervention. These concerns with the data flow on the internet combine the issues of distributed data processing, human-computer interaction, workflow and databases very closely. Our current understanding is that by using persistence we can first of all speed up the evaluation of sequence similarity queries. Secondly, we want to use persistence to record interactions with data (including data import from an external sequence database, reevaluation of the same query against an updated database of sequences, and re-examination of new matches in the context of knowledge already in our possession). By keeping all the intermediate information about data sources, the processing required and the outputs, we can then automate this interaction in the future. Additional computing techniques are then needed. We need to store both the descriptions of data and of actions performed, and this can currently be done using meta-data descriptions. Techniques for meta-data merging and provenance derivation [9, 43] are a subject of research, and no general solutions exist yet. By testing the latest theoretical work in practical context, we can improve our understanding of what issues need to be addressed next, and find out where current solutions are not satisfactory. Within this context we are investigating the use of User Interface Markup Language, UIML[10], to generate interface tools which could automate the process of sequence analysis.

We also see the role of this research in the general methodological debate. Most computing science research is very highly specialised. In our research we attempted to combine the results of algorithmic research with the latest persistence technology. It seems that experimentation with two branches of computing science research is fruitful as it allows us to see better the challenges of our field. In the algorithmic field we discover that worst-case complexity measures do not capture well the observed performance of transient suffix trees. In the field of persistence we see that an alternative model of persistence is required, one that does not follow closely the typical database model of efficient transactional updates. Our research testifies to the need for continuous re-assessment of assumptions and methods in the light of the experiments we carry out.

From yet another perspective, that of the emerging e-science effort, we see our work as contributing to the methodological discussion about the management of large data repositories. It appears that new indexing texhniques are indispensable, if this data is to be shared. Without indexing, one will not know what data are available, where they are, and how to access them. This applies equally to any of the data that will be shared via the GRID.

7.5.2 A practical view

Within the narrower scope of database support for sequence similarity searching, we see the need for further testing and refinement of our prototype solution. Following work could be done in this area. We list the main research directions which we then analyse in detail, in the sections which follow.

- Scaling up of the index.
- Data structure optimisation.
- Persistence optimisation.

[10] http://www.uiml.org/

- Optimisation of the approximate matching algorithm.
- Statistical techniques.
- Biological applications.
- Non-biological applications of indexing, for other reference data sets.

We now elaborate on these possible directions.

Scaling up

Comparison of entire genomes can be accomplished in two different ways, depending on the data processing scenario set by the biologist. One way to compare two sequenced genomes is to build a suffix tree, and to find common sequences by traversing it. The other scenario is to build a suffix tree for the stable genomic data and let biologists query it with sequences which they are producing in the lab. We believe a large suffix tree index would be useful in both contexts. To this end we propose that large scale tests be carried out with DNA, for instance with the entire human genome dataset. Our preliminary tests with 286 Mbp of DNA are described in Appendix B, an invited paper which we submitted to VLDB Journal in November 2001. This technology needs to be scaled up, so that indexes over mammalian genomes are possible, and further testing work could throw more light on the usefulness of such indexes.

We believe that for large scale testing an adaptive testing strategy would be required. One of the main difficulties in the large test for proteins was our inability to predict the running time of the test for some combinations of query length and threshold. An adaptive testing strategy would provide a public interface to some of the measurements gathered during the search itself. This would allow the program to automatically abort a computation which is not promising. This strategy could be extended to deal with the current problems of garbage collection within PJama (see Section 7.5.2).

Data structure optimisation

The prototype index we presented needs further refinement. Most indexing technologies are developed in an evolutionary manner, and refinements are examined in a succession of steps which gradually improve on the initial prototype. Similarly, alternative data structures also need to be examined in detail.

Within this context of refinement and comparison, we believe that known alternatives require re-assessment. Those will include the suffix array, the balanced suffix binary search tree, and a new data structure called the suffix vector [160]. Similarly, alternative implementations of those structures should also be considered. A new specialised layout for the suffix tree might also be developed, for instance based on subtrees being stored as integer arrays (in a way similar to the level-compressed trie [11]). In the same vein, compression of the indexed string itself could be attempted. In particular the work of Clark and Munro [55, 164] might open up some new ways of dealing with data structure compression and optimisation. This work suggests decomposing a string over any alphabet into its binary representation and building of a binary tree for that structure. A hybrid data structure combining the traditional tree with links with a naive tree could also be considered. This would

use suffix links just at the top few levels, down to the depth explored by the approximate matching algorithm.

Malcolm Atkinson's students [128, 186] have already been following this line of investigation. Further work needs to compare the alternative data structures from the point of view of their efficient use in approximate pattern matching. We now have a significant body of measurement data, including a record of the number of nodes traversed in the tree for each query, and the suitable combinations of query length and threshold. Given those data, and additional measurements that would be carried out for larger trees, the data structure could retain the tree top, and the tree itself below a certain level, dictated by the maximum query length used and threshold, could consist of leaf summaries. The actual cut-off point and possible gains from this approach could be easily arrived at, given a full tree traversal by string depth, and the count of nodes at each level.

The process of reengineering the implementation of the suffix tree alternatives carries a significant engineering cost but it may result in large performance gains. Any reduction in the overall size of the data structure is expected to reduce the time needed to fetch the data from disk which is one of the main limiting factors in any disk-resident index.

Persistence for large indexes

We developed our prototype using general-purpose indexing technology. This enabled us to compare five different data structures. However, further development is needed to ensure that this technology is efficient. Several strands of research are feasible.

A decision can be made as to whether using a general-purpose persistence technology is appropriate. We used a database-like implementation of persistence which incurs overheads in terms of space and processing. For specialised indexes which are not updateable a new persistence mechanism which factors out database housekeeping overheads would be more suitable. Current work [128] minimises the object overhead which currently dominates the size of the suffix tree node, and reduces the interference between recovery logging and the tree building process. Since the index itself is used only in read mode, recovery is limited to the computation failure at creation time. As the tree structure can be built in stages, recovery does not necessitate a traversal of the entire object graph, but is limited to the current partition only. As a result, a tailored storage layer may turn out to be a superior solution in this context.

Alternatives, like Gemstone/J[11] and Oracle Extensible indexing framework [207, 13, 132] should also be considered. They would have the advantage of offering an easier way of integrating the index with other data types which we would like to query in a biological database system.

Further in this research, additional optimisations will also be possible. Those will investigate the disk placement of objects (clustering), prefetching and caching strategies. Those optimisations will have to take into account observation of user and disk activity, and accommodate the observed tree traversal pattern.

Optimisation of the approximate matching algorithm

This type of research would use the existing body of algorithmic knowledge to produce a faster implementation of the DP matrix evaluation calculated with the help of the index.

[11] http://www.gemstone.com/

Frequently in optimisation work a combination of theory and measurement is needed, and experimentation will throw a new light on the current understanding of theory and best practice.

We believe that the code used in BLAST could be adapted to this context, and measurement of the indexing gain delivered by the index will have to be compared with the filtering offered by BLAST. For instance in protein BLAST, mutated protein substrings are generated, and a tree index of those strings is built. This technique of indexing all possible mutations of query substrings could be tested in conjunction with our index, based on exact matching. Further, optimisations of the DP calculation method are also possible. Lastly, the derivation of thresholds at which the computation stops also needs to be reexamined.

A query will have to partitioned before being evaluated. The actual partitioning strategy, and therefore the depth of tree traversal will rely partly on the measurements we have already carried out for proteins and DNA, as outlined in Appendix B. We believe that scanning a batch of queries to find repeating substrings offers another possibility for optimisation.

Statistics

Statistical measures of the goodness of a sequence alignment are a subject of debate [158, 69]. BLAST follows one particular model, but other views are also possible. Integration of statistical measures with indexing technology may offer new solutions in the future. It is conceivable that known distributions of sequence repeats could be integrated into the index structure itself, or could be used to pre-screen the query. The query itself could also be examined using statistical tools [68] and an adaptive query strategy is also imaginable, based on statistical properties of the index. Statistical approaches to database queries, using summaries [42, 124], are becoming an integral part of database systems in a situation where a full database scan is not feasible. This mirrors the situation in BLAST where we are not guaranteed to find all the relevant hits. However, by combining indexing and statistics about the indexed sequence, we could conceivably deliver all relevant matches faster.

We see it as one of the limitations of our research that we could not develop sufficient statistical expertise to include statistical considerations in our work. We believe that future work on sequence indexing might remedy this insufficiency.

Biological applications

This is a wide area for future research, and we concentrate only on a few possible directions. Gusfield [99] quotes at least 50 possible uses of a suffix tree but we concentrate on contexts where large trees are needed.

It would be interesting to use large suffix trees with approximate motif discovery techniques to identify sequence motifs [37, 67, 225, 149, 187]. Current pattern discovery approaches have been limited by the maximum tree size. Combination with statistical filtering of possible patterns might offer a useful way of finding the relevant repeated substrings.

As the partial matches may be assessed differently depending on the biological context, visualisation mechanisms appropriate to this use of suffix trees could be developed. Currently, BLAST at NCBI[12] uses colour coding to distinguish between strong and weak sequence similarities. Beyond that, matches to different chromosomes and species could be displayed in positional context, and the density of partial matches in a particular area could

[12] http://www.ncbi.nlm.nih.gov/BLAST/

indicate that there is high overall sequence similarity. It would be interesting to display all the hits returned by the index and let the visualisation tool organise them into meaningful patterns.

Since versions of suffix tree which are suitable for both sequence and structure discovery are known, it is not inconceivable that our techniques could be extended to this context [202].

7.5.3 Priorities

We believe that the best approach to develop an initial solution useful in biological sequence searching would have to combine an optimised data structure with scaling up to index two mammalian genomes. Our preferred optimisation would combine integer arrays storing the top of the tree with leaf summaries for nodes at depths exceeding the tree traversal depth. Further work would combine the index with a set of data visualisation and manipulation tools.

Stage two of index research could include a comparison with the suffix array, and the use of protein similarity matrices, as well as the adoption of known statistical measures of sequence similarity.

Finally, or perhaps in parallel, database optimisation techniques could be used to improve the clustering and prefetching strategies for the index under investigation.

7.6 Limitations of our work

We now look at the limitations of our work. Those result from the choice of data sets for testing, from our persistence mechanism, from lack of proper biological evaluation of our work, and from our lack of statistical expertise.

7.6.1 Data related limitations

We chose datasets based on their availability, and they are representative in the case of proteins (where we used the largest available protein database) but they are not representative of all the DNA datasets in use. In particular, we built our index using genomic data, and not short sequence fragments which result from individual sequencing runs.

We did not preprocess the data used for querying to generate an even spread of possible queries, in terms of sequence composition or frequency of repeats. Additional preprocessing of our DNA data sets might have helped us to identify a statistically significant sample of queries which reflect the typical features of sequence which we indexed. We wanted, however, to approximate queries as they are submitted to a BLAST server. We made an attempt to gather such query data but it turned out that web server logs do not allow for collecting such data easily. Biologists (Julian Dow, private communication) demonstrated that typical searches use known protein and DNA sequences and assured us that using any sets of sequences constitutes a possible query set.

We feel that the tests carried out with protein data represent another limitation. We found that protein queries take a long time to evaluate. This is due to a larger alphabet and a higher fan-out of a protein index. We executed protein queries for one week, and had to abort several runs. An adaptive query strategy, where queries would automatically abort if

they take too long to evaluate or produce an exceedingly large result set would have helped in this case.

7.6.2 Persistence limitations

We did not test other persistence implementations, because of the time constraint imposed on a single PhD project. We believe that other persistence alternatives need to be explored.

We did not explore the management of the garbage collection problem within PJama and this means that the times reported for query evaluation are excessively long. Since PJama does not manage garbage collection well, our initial attempts to execute a scan of the top of the tree were unsuccessful. We had to introduce explicit calls to the garbage collector. We decided to take a shortcut and call the GC after every match found which scanned the subtree in search of leaves. In practice, we believe that GC must have been called too often and led to inefficient query evaluation. A better solution would have included calls to the memory subsystem to establish how much memory was being used, and to carry out GC only if needed. However, such a solution would require a heuristic which would help us decide when to force GC, and if that heuristic failed, we would have been faced with system crashes.

7.6.3 Lack of biological evaluation

We did not manage to carry out a biological evaluation and a comparison with BLAST. It turns out that many issues will have to be explored in detail before such an evaluation and comparison with BLAST are possible. This situation is disappointing but reflects the complexity of the problem we are trying to solve.

7.6.4 Statistical refinement

We would have liked to have included a higher level of statistical refinement to this work. This might have helped in the choice of test data, in the design of the index, and in the final sequence similarity measures which are statistically based. We believe that further work will be able to use those techniques.

7.7 Our contribution to methodology of computing science research

We see our work as an attempt at introducing persistence technology into large scale sequence similarity searching scenario. We believe the first step has been made, and we managed to gather a large body of experimental evidence to convince others that indexing of sequence data may be possible. We hope that we also may have encouraged others to experiment with persistence, and look at other data types which currently do not use indexing.

We showed that combining the fruits of algorithmic research with databases is a very interesting area of research and can enrich both fields. Our discussions and exchanges with both algorithmic specialists and database specialists testify to this thesis.

We hope that by attempting to index sequence data we might bring forward integrated solutions to bioinformatics data storage and processing, and have significant impact on the

development of new technologies for other data types which are used in biomedical research and drug development.

7.8 Closing

This thesis presented the motivation for improving the technologies used in searching biological data, and focused on sequence data searching. Our contribution is the development of a practical method of building large suffix trees, in excess of RAM, and first measurements of the possible speed up in sequence searching which is produced by this indexing structure. The recognition of the requirement for new data processing methods in biology is also one of our findings. We believe that our work contributes to the possible future development of new databases which will support large scale data operations on biological data types, currently outside the scope of database technologies.

Appendix A

Physical map integration using a relational database: the example of the Human Chromosome 21 DB

Ela Pustułka-Hunt, Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, UK, Hans Lehrach, and Marie-Laure Yaspo, Max-Planck-Institut für Molekulare Genetik, Ihnestrasse 73, D-14195 Berlin, Germany

This draft was first produced in November 1999 and updated in April 2000.

Abstract

Motivation: Within the human genome project initiative, building chromosome specific databases integrating various data sources and physical maps provides the best support for data analysis and dissemination. Physical map construction and integration are either based on semi-manual methods or require the creation and maintenance of complex scripts. The resulting maps are considered to be idiosyncratic, as they do not expose the logic behind the positioning of the mapped features. The mapping itself is extremely complex and mapping decisions not directly traceable.
Results: We developed a new computing approach which simplifies the construction of integrated maps, and we used it to create an integrated physical map of Human Chromosome 21, and to position genomic clones and transcriptional units along this chromosome. Relational database techniques allowed for simultaneous mapping of sets of items, based on both published datasets and on our laboratory results recorded in a database, and allow us to make improvements to the existing maps as new data become available. Our mapping decisions which interpret the available experimental data are provided as item annotations. We describe our data model and the techniques used in map integration. We discuss how this method is used in conjunction with other software including ACeDB and a Java map browser. This approach should be applicable to any physical mapping project, particularly in the context of integrating chromosome maps prior to sequencing.
Availability: Chromosome 21 Database and maps are available at http://chr21.molgen.mpg.de/.
Contact: *ela@dcs.gla.ac.uk, yaspo@molgen.mpg.de*

Introduction

The work we describe started in 1996. Our chromosome 21 integrated map was built gradually, and simultaneously with the effort of data integration. Map refinement is an ongoing process relying on the very powerful abstraction of mapping data which we developed, and is backed by a database approach to map generation, which uses a relational database to perform exhaustive searches on the entire data space. The relational query engine optimises query performance, and the use of indexes and database tuning techniques enables operations on a large data set encompassing some 240,000 results referring to 40,000 potential map objects, with currently 4,500 mapped objects, where one map object may represent an object cluster.

Our decision to use a relational database for mapping was based on several premises. We were planning to make our data available on the web. We knew that without code optimisation and data compression we would not be able to keep our projected data sets in computer memory, so that delegating optimisation issues to a database engine was the right choice, see [49] for an overview of database query optimisation techniques. We realised that the task of map integration was separate from map building, and in particular we had access to only some of the data used to produce maps which we were integrating, so that is several chromosome regions we could not recalculate the maps, but had to scale and align the existing maps instead. A significant portion of our data came from hybridisation experiments involving small clones and had different characteristics from YAC STS content data where the ratio of STS to YAC hits is high (6.2 in [102]). For instance, there were only 16,000 strong positive hybridisations between 11,000 cosmids and 8,000 cDNAs, and the cosmids did not form contigs.

Our approach was to concentrate on map refinement, and gene positioning in particular. Existing software did not directly cater to the requirement of building a physical map based on NotI fragments, and did not interface readily to a relational database. An interested reader may consult [34, 102, 167] for a discussion of available mapping tools and issues in map integration. We judged that managing the data flows between different data formats, databases and mapping tools would be beyond the means of our project, and tried to reduce our software environment to minimum.

We use relational calculus to build an integrated map, and because of exhaustive searching performed by the database engine we are guaranteed the best possible answer, based on our judgement of the quality of data we are using. We can easily exclude unclear results from mapping, and provide feedback on lab results, leading to re-evaluation or repetition of an experiment.

Our approach is viable for large data sets, and allows for incremental map building and refinement. As all data reside in one database, re-running a list of SQL commands can be done each time new data are loaded, and the queries to be run depend on the type of data being added. Data loading and integration are handled by Perl scripts and are not discussed further.

The structure of data during the process of physical mapping is very dynamic and the biggest challenge was to integrate data gathered from heterogeneous sources and also showing different levels of accuracy. This implies in general that data management supporting

mapping projects should accommodate frequent updates and modifications, and should be based on a robust database.

We believe that a fully automated solution to physical map integration is not viable as data come from many sources and differ in resolution and accuracy. Moreover, since most of the software packages used in mapping include a visual assessment of data complexity and quality, it is not likely that the logic of physical map integration can be fully encapsulated in software. Instead, a semi-manual method is preferable, where a map is constructed incrementally, with visual and programmatic checks at different stages of map construction, and with annotations being added to map items for which inherently contradictory results exist.

Our approach to map integration improves significantly on current practice, by minimising the use of scripts and therefore the labour involved in mapping. By relying on standard and robust computing tools (a relational database), we achieve map integration more easily, in a traceable manner. We also achieve consistent annotations and referential integrity of data with less programming effort. The graphical display is an ACeDB-inspired Java map browser communicating with a database (Grigoriev et al., 1998; D. Leader, http://www.-biochem.gla.ac.uk/BMB/Res/MBGE/DPL.html), which is used to visualise the integrated map and to highlight possible problems. In this paper we give an example relational schema, and illustrate the map building process with relevant SQL statements. We also discuss the issues of data import and export, and the database tuning techniques used in this context.

System and Methods

Data sources

The integrated physical map of Chromosome 21 used a backbone initially based on YAC pocket map and YAC minimum tiling path [172, 88], providing anchors for data integration. Physical distance intervals along Chromosome 21 were defined by a NotI restriction map [123]. Published high resolution bacterial clone contigs were imported and anchored to the map [73, 210, 98, 105]. Transcriptional mapping was mostly based on a previously described strategy [235]. Mapping of genes and transcription units isolated from cDNA selection and exon-trapping experiments refers to the following data sources imported into the database: (Cheng et al., 1994; Yaspo et al., 1995; Yaspo et al., unpublished; Tassone et al., 1995). ESTs were collected from the UniGene Initiative Set [197]. Known Chromosome 21 genes were generally localized to YACs from data culled from literature. The gene list was actualized after the index of protein entries on Chromosome 21 maintained on the ExPASy server (Bairoch, http://www.expasy.ch). STS markers were imported from the Genome Database (GDB, http://www.gdb.org) or / and from previously established maps (see above) and [209]. A large part of the data import consisted of large-scale hybridisation data from our laboratory produced in the course of constructing sequence-ready maps for Chromosome 21. Finally, somatic cell hybrid data were also loaded in the data set [96].

Computing methods

We used an Oracle database version 7 on a DEC alpha workstation with 128MB RAM, Perl scripts [228], an ACeDB database (Durbin and Mieg, http://alpha.crbm.cnrs-mop.fr) and a spreadsheet for map data entry from printed maps. Perl scripts were used to move

data between ACeDB and Oracle, to send multiple BLAST requests [7] and retrieve data sets from GDB and UniGene. Data parsing was performed using Perl regular expressions. ACeDB vertical map display was used to check map consistency during the initial map construction while a Java map browser provided internet access to maps. The present Oracle database was built and is being accessed via OPM [50]. Mapping is carried out using SQL, which simplifies the syntax and improves performance.

Algorithm and Implementation

Data integration strategy

An initial map consisting of YACs and STS markers corresponding to a minimum tiling path (MTP) map [88] was built by a Perl program which started from the telomere and walked along the chromosome to position the YACs and STSs approximately, with STSs placed equidistantly within the YACs, while preserving STS-YAC matches and YAC lengths. STS and YAC positions were then shifted to reflect true NotI distances, as defined by the NotI map [123], by walking the chromosome with SQL update statements, and moving map sections up or down the scale for each known NotI distance. Additional STS markers were then added, based on published data, or had their position adjusted, where exact STS distances were available. A few gaps in the MTP were filled with additional YACs from high resolution clone contigs from other maps [53] where overlaps with already placed STS markers were known. In the next step, available YAC-to-cosmid hybridisation results [172] and raw laboratory data were used to position the cosmids relative to known YACs, with approximate cosmid positions computed based on positive overlapping YACs. Additional cosmids, PACs and BACs from the published maps described in Section **Data Sources**, were also added. The resulting map containing a large number of clones could then be used to position trapped exons and cDNAs, based on cDNA to cosmid and cDNA to YAC matches, as defined by hybridisation [235]. This strategy was then complemented with positioning new STS markers in bins defined by somatic cell hybrid breakpoints, and with BLAST results which matched cDNAs against known exons or STSs. BLAST matches were used in mapping in a way analogous to hybridisation or PCR data, and documented in the database. Additionally, genes were positioned on the map manually. The resulting map contains positions for the following items: genomic clones (YACs, BACs, PACs, cosmids), STS markers, ESTs, genes, and, finally, cDNAs and exons grouped into transcriptional units mapped to genomic clones. Mapped items which are recorded in public databases have a hyperlink to the original data source, and the rationale behind the map position is shown as annotations. Data were tagged for ownership, in particular, hybridisation results derived at Max-Planck Institute are shown with the date on which they were entered in the database.

Database design

Our database design is a conceptual simplification of mapping data encountered both in experimental and *in silico* approaches to mapping (using BLAST). We group all mapping data into three entities. This allows us to use a small number of queries with changing parameters to perform all mapping operations. We use the following tables:
Map: mapped features and their positions, with annotations.
Results: experimentally and computationally derived relationships between objects (posi-

tive or negative hybridisations, PCR, sequence homology from BLAST, etc.).
Attributes: clone or gene attributes, e.g. measured length(s) of clones or fragments.
The following SQL syntax describes the tables at the conceptual level. In the actual implementation, unique row identifiers (integers) in each table were added and used as keys to make the joins between tables more efficient.

```
CREATE TABLE MAP ( TYPE VARCHAR (50), NAME VARCHAR (50) CONSTRAINT
MNAME UNIQUE, POS1 NUMBER, POS2 NUMBER , ANNOTATION VARCHAR (2000),
CHECK (POS1 < POS2) );

CREATE TABLE RESULTS ( NAME1 VARCHAR (50), NAME2 VARCHAR (50), RESULT
VARCHAR (50), EXPERIMENTTYPE VARCHAR (50), EXPERIMENTSOURCE VARCHAR
(50), CHECK (NAME1 < NAME2) );

CREATE TABLE ATTRIBUTES ( NAME VARCHAR (50) CONSTRAINT ANAME UNIQUE,
TYPE VARCHAR (50), LENGTH NUMBER );
```

We use unique object names, following GDB and ExPASy nomenclature for genes, STSs and YACs, and GenBank and UniGene identifiers for other items. Constraints are used to ensure the uniqueness of Map and Results entries and the order of coordinates in a map entry. Indexing all columns allows the database engine to optimise queries. Synonyms are handled separately from mapping data, and are used in building object clusters, and in data presentation on the web. Object clusters are represented by one map object, while relationships between objects in a cluster are kept in the Results table.

Map integration

Our approach to map building is iterative. Reading new data into the database is followed by map recalculation. Initially, inconsistent results are isolated and in case of lab results returned to the lab, and for published results, annotated and recorded for future reference. For instance, new hybridisation results involving cosmids and cDNAs are delivered as a flat file, and a script generates database insert statements. At that point a check would be made to find out which cDNAs mapped inconsistently, and lists of inconsistent results would be produced to go back to the lab, and saved in the database. Section **Data discrepancies** gives example SQL queries used.

We explain our method with an example of adding a contig to an existing chromosome map as shown in Figure A.1. We use arbitrary small numbers as YAC lengths to simplify the presentation. In our YAC-STS mapping we excluded some YACs and matches because of large discrepancies between different data sources, or YAC chimericity. In recording YAC lengths, we entered all the available lengths initially, but used the shortest one given. Our example sequence ready **Contig** consists of BAC and PAC clones ordered according to STS markers:

TYPE	NAME	POS 1	POS 2
STS	STS_1	0	
STS	STS_2	10	
STS	STS_3	25	
STS	STS_4	30	
BAC	BAC_A	1	12
BAC	BAC_B	8	25
PAC	PAC_C	23	28
PAC	PAC_D	26	30

As the existing map stretches over a longer area, we look only at the section corresponding to the contig. Relevant **Map** entries are:

TYPE	NAME	POS 1	POS 2
STS	STS_1	200	
STS	STS_3	220	
STS	STS_4	228	
YAC	YAC_1	210	225
YAC	YAC_2	215	232
GENE	GENE_G	220	222

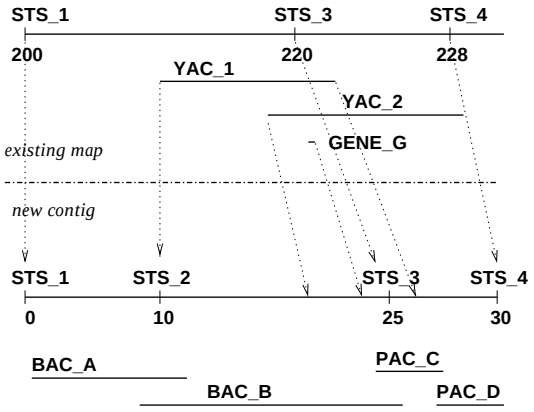


Figure A.1: A new contig and the original map. Dotted arrows indicate object correspondence.

Integrating the contig with an existing chromosome map can be decomposed into three logical steps:

- adjustment of markers in the existing map (as the bacterial clone contig is more accurate than the existing map)
- addition of new clones from the contig to the map
- adjustment of existing mapped clone or gene positions reflecting their relationships to other map objects.

We assume that **Map** and **Contig** are drawn to the same scale and orientation. If that is not the case, simple arithmetic operations can align them. We note that small map objects have an empty **pos2**, and ignore the existence of objects having more than 2 coordinates (i.e. somatic cell hybrids). We know that GENE_G was mapped based on positive results against both YAC_1 and YAC_2. We have the following experimental data in the **Results** table, and note a positive result for (YAC_1, STS_2) but STS_2 not being on the map yet.

NAME 1	NAME 2	RESULT
YAC_1	GENE_G	positive
YAC_2	GENE_G	positive
YAC_1	STS_1	*negative*
YAC_1	STS_2	positive
YAC_1	STS_3	positive
YAC_2	STS_3	positive

We proceed in three steps. We adjust STS positions in the existing map first, to remove the discrepancy in the distance between STS_1 and STS_4. We can shift all existing map objects to the right of STS_4, or to the left of STS_1. Choosing the first option, we move all objects to the right of or overlapping with STS_4 to the right. Following SQL can perform this operation:

```
CREATE TABLE MAPDIST AS SELECT (M1.POS1 - M2.POS1) OFFSET FROM MAP M1,
MAP M2 WHERE M1.NAME ='STS_4' AND M2.NAME = 'STS_1';

CREATE TABLE CONTIGDIST AS SELECT (M1.POS1 - M2.POS1) OFFSET FROM CON-
TIG M1, CONTIG M2 WHERE M1.NAME ='STS_4' AND M2.NAME = 'STS_1';

CREATE TABLE MOVEDIST AS SELECT ABS (C.OFFSET - M.OFFSET) DIST FROM
CONTIGDIST C, MAPDIST M;

CREATE TABLE STS4 AS SELECT POS1 POS FROM MAP WHERE NAME = 'STS_4';

UPDATE MAP M SET M.POS1 = (SELECT (N.POS1 + DIST) FROM MAP N, MOVEDIST
WHERE M.NAME = N.NAME) WHERE M.POS1 >= (SELECT POS FROM STS4) AND
M.POS2 IS NULL;

UPDATE MAP M SET POS1 = (SELECT (N.POS1 + DIST) FROM MAP N, MOVEDIS
WHERE M.NAME = N.NAME), POS2 = (SELECT (N.POS2 + DIST) FROM MAP N,
MOVEDIST WHERE M.NAME = N.NAME) WHERE POS2 >= (SELECT POS FROM STS4);
```

The second step adds all contig objects to the map. The contig is first aligned with the map, and then the map updated:

```
UPDATE CONTIG SET POS1 = POS1 + 200, POS2 = POS2 + 200;

UPDATE MAP M SET POS1 = (SELECT C.POS1 FROM CONTIG C WHERE C.NAME =
M.NAME), POS2 = (SELECT C.POS2 FROM CONTIG C WHERE C.NAME = M.NAME)
WHERE NAME IN (SELECT NAME FROM CONTIG);

INSERT INTO MAP (SELECT * FROM CONTIG WHERE NAME NOT IN (SELECT NAME
FROM MAP));
```

Lastly, we consider objects which were mapped by inference, or which may be related (via **Results**) to newly added objects. This must refer back to the way the existing map was constructed and may have to be carried out in a few phases. Here, YAC_1, YAC_2 and GENE_G were mapped approximately. YACs were aligned with STSs first, and the gene was

subsequently positioned in the YAC overlap. We now adjust the positions of those objects to reflect the corrected STS positions in the contig area. Objects in the map are only moved if the additional information provided by the contig contradicts the existing map. We check for possible conflicts between YAC and STS positions using the following query which will return *all* YACs and their *non-matching* STSs where YACs have been placed incorrectly, i.e. overlapping a negative STS:

```
SELECT R.RESULT, M1.NAME, M1.POS1, M1.POS2, M2.NAME, M2.POS1 FROM MAP
M1, MAP M2, RESULTS R WHERE M1.TYPE = 'YAC' AND R.RESULT='NEGATIVE'
AND M2.TYPE = 'STS' AND ( (M1.NAME = R.NAME1 AND R.NAME2 = M2.NAME)
AND (M1.POS1 < M2.POS1 AND M1.POS2 > M2.POS1)) OR ((M1.NAME = R.NAME2
AND R.NAME1 = M2.NAME) AND (M1.POS1 < M2.POS1 AND M1.POS2 > M2.POS1));
```

This query may return:

result	name	pos1	pos2	name	pos1
negative	X	30	100	Y	50

If such an inconsistency is discovered, YAC X has to be repositioned or the data discrepancy noted in **Map. annotation**.

In the map creation process we also take into account clone lengths stored in the **Attribute** table.

NAME	LENGTH
YAC_1	20
YAC_2	20
GENE_G	null

We compare clone positions to their lengths **Map.pos2 - Map.pos1 - Attribute.length**, and add map annotations or adjust the map accordingly. This problem was most apparent for YACs showing different lengths and STS content in different data sources. Our solution was to add YACs to the map only if there were no large discrepancies.

We now show how we positioned smaller objects on the map. We calculated the maximum extent of the interval in which a gene or cDNA could appear and then positioned the object in the middle using either a conventional gene length of 20Kb or by extending from the calculated central position in both directions by 1/2 of the known length. To simplify the task, we use temporary tables to store gene and cDNA positions:

```
CREATE TABLE TMP AS SELECT * FROM MAP WHERE TYPE = 'GENE' OR TYPE =
'CDNA';
```

The **tmp** table is then updated to reflect the maximum interval that a gene can take. This is calculated based on all positive objects as shown in Figure A.2.

```
UPDATE TMP SET POS1 = (SELECT MAX (M2.POS1) FROM MAP M1, MAP M2,
RESULTS R WHERE R. RESULT = 'POSITIVE' AND M1. TYPE = 'GENE' AND
((M1.NAME = R.NAME1 AND R.NAME2 = M2.NAME) OR (M1.NAME = R.NAME2
AND R.NAME1 = M2.NAME))), POS2 = (SELECT MIN (M2.POS2) FROM MAP M1,
MAP M2, RESULTS R WHERE R.RESULT = 'POSITIVE' AND M1. TYPE = 'GENE'
AND ((M1.NAME = R.NAME1 AND R.NAME2 = M2.NAME) OR (M1.NAME = R.NAME2
AND R.NAME1 = M2.NAME) ));
```

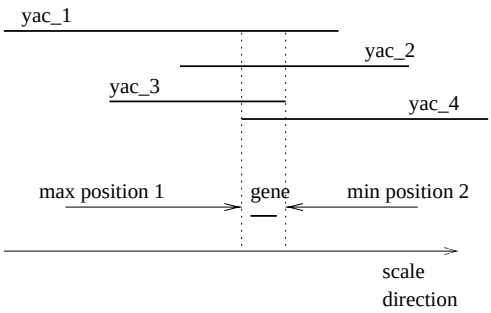


Figure A.2: Placing a gene in the overlap of positive YACs.

Finally, updates to map are performed based on object length.

```
UPDATE MAP SET POS1 = (SELECT ((T.POS1 + T.POS2)/2 - A.LENGTH/2) FROM
TMP T, ATTRIBUTES A WHERE T.NAME = A.NAME AND A.LENGTH IS NOT NULL),
POS2 = (SELECT ((T.POS1 + T.POS2)/2 + A.LENGTH/2) FROM TMP T, ATTRIBUTES
A WHERE T.NAME = A.NAME AND A.LENGTH IS NOT NULL) WHERE NAME IN (SE-
LECT NAME FROM TMP);
```

Data discrepancies

Prior to adding a new data set to the map, a check on data quality is performed. We use the table **Duplicated**, see section **Performance and Computability** to find a relevant subset of results first. For example, after adding new hybridisation results where cDNAs bear the prefix 'MPIpl', we select results relating to those cDNAs:

```
CREATE TABLE CDNACLONE AS SELECT NAME1, NAME2 FROM DUPLICATED WHERE
RESULT = 'POSITIVE' AND NAME1 LIKE 'MPIPL%';
```

We then supplement this data with map positions of features which hybridise to the cDNAs:

```
CREATE TABLE CDNAMAP AS SELECT * FROM CDNACLONE C, MAP M1 WHERE
M1.NAME = C.NAME1;
```

And, finally, we produce a table of contradictory results which will give feedback on the quality of lab data:

```
SELECT NAME1 NAME, MAX(POS1), MIN(POS2) FROM CDNAMAP GROUP BY NAME1
HAVING MAX(POS1)>MIN(POS2) UNION SELECT NAME2 NAME, MAX(POS1), MIN(POS2)
FROM CDNAMAP GROUP BY NAME2 HAVING MAX(POS1)>MIN(POS2);
```

Subsequent mapping queries can exclude this data, or reposition the positive features if appropriate, and add annotations to the map.

ACeDB and data integration

We progressively extended the ACeDB data model to accept our data and parsed and imported all data into ACeDB. We used electronically available data where possible, augmented with some map data scanned or typed in from paper publications. Data were tagged for ownership as appropriate. Data integration, in particular translation of web pages, was labour-intensive, as the complexity of data formats we encountered required extensive parser programming. Recently available translation tools [193] might simplify data translation.

After our MTP map was built, we migrated the mapping data into a relational database. A stored ACeDB query was run first. A script accessed ACeDB via the tace interface, submitted the query, stored retrieved data in a file, parsed it, wrote database insert statements to file, submitted them ORACLE and produced an error log. Those steps can be done manually as well. Exporting data to ACeDB follows a similar route. The map can be checked visually by clicking one by one on all map objects. ACeDB map display can be set to show wrongly mapped objects in red. Such visual checking can be satisfying, but is not reliable for a large number of items, as discrepancies can be easily overlooked. Using SQL to check the map provides a conclusive proof of correct mapping.

Data export to a web based map browser is done via a script which submits the SQL statement:

SELECT * FROM **Map**,

formats the data, and passes the results to a Java applet.

Discussion

Interpretation of published materials

We found that the interpretation of published maps presented considerable problems due to low resolution of printed maps and also occurrence of possibly conflicting data. Where electronic data were available, we used SQL to find the discrepancies in experimental results, and to make a decision regarding our interpretation. It was then possible to document such decisions as map annotations pointing to the original results. On the other hand, the interpretation of printed maps was often ambiguous. An apparent contig, including small gaps, might cover only 60 per cent of a particular interval. Additional difficulties arose from the fact that map digitisation or manual reading with a ruler are inexact, and an error margin of up to 20 per cent is to be expected. In retrospect, we wish to have had access to all original experimental results in electronic form, and to re-compute the map whenever new data became available. This is the approach to map refinement taken in the Chromosome 21 database, with all new results being made available in the database and used in map refinement, leading towards the integration of recently built sequence-ready contigs.

Map visualisation

The first attempt to build an ACeDB-based Chromosome 21 database was presented by O. Ritter at the 4th International Workshop on Chromosome 21 [65]. Our subsequent data integration work used the ACeDB database and map display to assess the quality of YAC-based tiling path map. We soon found out that ACeDB itself was not well suited to mapping

or to publishing the map on the web, and mapping was moved to ORACLE. When a Java map applet became available [97], other data were migrated to a relational database to be retrieved by the applet on demand. The first applet version did not have full functionality, but later improvements (D. Leader, http://www.biochem.gla.ac.uk/BM B/Res/MBGE/DPL.-html) resulted in a map browser which generates postscript files for printing, has advanced zooming and data selection capabilities, and offers a novel approach to the visualisation of overlapping objects, relying on superimposing objects at low map resolution. The new applet is fully interactive and has a drill-down facility giving direct access to supporting database data.

Schema modelling

Our minimum schema suffices for mapping. However, if an integrated map is going to be used as part of a database system, it may be desirable to perform database normalisation to ensure data quality and minimise time consuming data cleaning operations which currently dominate the curation of ACeDB databases.

Performance and computability

Using large data sets may require a combination of query partitioning and database tuning or data duplication. We discuss logical horizontal data partitioning first, and data duplication second. In mapping, a multiple join of the map table with the results table containing some 240,000 entries is made, potentially resulting in a very large Cartesian product. We proceed by concentrating on data subsets. When mapping cDNAs onto cosmids, we exclude other objects and consider 11,000 cosmids, 8,000 cDNAs and 16,000 cosmid-cDNA matches, as shown in section **Data discrepancies**. If a particular data table join does not fit in memory, data can be partitioned further, and map sections calculated one at a time.

Data duplication was used to present summary information for each database object, including the available results. We duplicate the **Results** data, so that for each positive match between A and B (A,B,positive) we add an entry (B,A,positive). This can be achieved by the following SQL which is run every time **Results** are updated:

```
CREATE TABLE DUPLICATED AS ((SELECT NAME1 , NAME2, RESULT, EXPERI-
MENTTYPE, EXPERIMENTSOURCE FROM RESULTS) UNION (SELECT NAME2 , NAME1,
RESULT, EXPERIMENTTYPE , EXPERIMENTSOURCE FROM RESULTS));
```

The table **Duplicated** is subsequently used in mapping. SQL queries become shorter as joins are performed on **Duplicated.obj1** instead of both **Results.obj1** and **Results.obj2**.

Limitations

Relational calculus does not allow for recursive queries of unknown depth. Therefore, data clustering which we performed for ESTs, STSs, exons and cDNAs, based on BLAST sequence matches, can use either PLSQL programs (a procedural extension of SQL) or export data, create clusters using a script, and insert them into the database. As UniGene data are already clustered, this was a small operation.

OPM constrained and changed our database management techniques. As an object oriented layer on top of a relational database, OPM reduced the amount of design work by

using inheritance. However, it made data management harder, as it did not support stored procedures for ORACLE. It guaranteed referential integrity using constraints, but it did not let us write stored procedures necessary for some data updates.

Conclusion

We conclude that our solution has been very successful in building an integrated chromosome map and in minimising the effort required for map refinement. We annotated our mapping decisions in the database and we can trace the provenance of map object positions to experiments performed in the lab or to the original data sources used in map construction.

We found a combination of a relational database and ACeDB to be useful. The first version of the Java map applet could not print maps or display STSs or ESTs satisfactorily. Mapping data were held in both systems, and mapping was performed in a relational database, while map display was done using ACeDB. Being able to update data in place in ORACLE outweighed the cost of producing two short scripts. Our initial approach to use just ACeDB and calculate maps using Perl was not satisfactory as it cost too much programming effort.

By simplifying database design and using SQL we managed to integrate several maps into one, and we are prepared to perform further map integration as and when required. We are also sure of the high quality of our map as we are positioned to find any data discrepancies quickly. Our method of integrating map data is reliable and can be applied in any map integration project.

We found that using standard database technology lets us improve our maps as new results become available. We add new contigs to our map efficiently, in particular all the sequence-ready contigs which have entered the sequencing phase are in the map with links to sequencing centers. The forthcoming Chromosome 21 genomic sequence produced by the Chromosome 21 Mapping and Sequencing Consortium (Riken, Tokyo University; IMB-Jena; Keio University; GBF Braunschweig; MPI-Berlin) will be fully available with corresponding maps in this database. After the sequencing of Chromosome 21 is accomplished, we will be able to refine our map, integrate genetic and RH maps [209], add new genes, gene features and SNPs, as a tool to support the shifting focus of genetic research towards gene functional analysis, and to provide the basic data for further linkage and disease studies.

Acknowledgments

This work was supported by grants BMBF01KW9608 (German Human Genome Project) and EU grant BMH4-CT96-0554. We wish to thank the bioinformatics team of the Resource Centre of the German Human Genome Project (RZPD) for their support and advice.

We thank David Leader for giving us access to the recently improved map browser and Guenter Teltow for his management work for the Chromosome 21 database. We thank Malcolm Atkinson, Karen Renaud, and the anonymous referees for their comments.

Appendix B

Computing resources and software used in the Human Genome Project

	Public consortium	**Celera**
raw sequence produced	23 Gbp	14.8 Gbp + 3-fold coverage of mouse genome (9 Gbp)
raw sequence used for assembly	23 Gbp	14.8 Gbp + 9 Gbp of public sequence

Table B.1: Volume of sequence data.

	Public consortium	**Celera**
computing power	27 node cluster of Alpha Server ES40s with 108 CPU's. Final assembly of 400,000 sequence fragments from 30,000 clones resulting in 2.7 Gbp using GigAssembler takes less than 4 days on a cluster of 100 800MHz Pentium III CPUs running Linux. Total Sanger Centre computing resources, some of which were used in the project, consist of 700 Alpha processors, 250 PCs, 150-plus network computers and 250 NT/Mac collection devices for data produced by ABI sequencers.	LIMS used a Virtual Compute Farm (VCF) of 440 Alpha CPUs, memory 2-8 GB per system. VCF was used for trace file processing and annotation. Genome assembly used 16 computers with total memory of 96 GB, largest single memory was 64 GB.
disk storage	Total file storage and processing used for sequencing in all centres not accounted for. A file server with 1 TB of disk used in the annotation and analysis at the Sanger Centre.	Total of 100 TB
lab automation	some labs fully automated, some partially	Laboratory Information Management System (LIMS) used to track samples

Table B.2: Hardware, storage and automation of human genome sequencing.

	Public consortium	**Celera**
map preparation	FPC (fingerprinted contigs)	none
sequencing	PHRED for base-calling	Paracel's Trace Tuner and rules incorporated into LIMS
sequence assembly for individual clones	PHRAP	see above
filtering of contaminants	not specified	BLAST
contig assembly	GigAssembler	2 strategies attempted: whole genome assembly (WGA) and compartmentalised shotgun assembly (CSA), the second method produced better results. Visual inspection and curation of of scaffolds was necessary.
alignment of contigs with chromosomes	ePCR and BLAST	ePCR
internal genome duplication analysis	BLASTN, all-against-all similarity search in DNA sequence	MUMmer based on suffix trees, using not DNA but predicted genes
gene prediction	Spidey, Acembly, Ensembl using Genscan, GeneWise, Genie	OTTO, rule-based expert system based on BLAST and SIM4 comparison to sequence databases, including Celera's mouse sequence. Genscan for gene prediction.
proteome analysis	InterPro front-end, BLASTP, PSI-BLAST, all-against-all Smith-Waterman, SMART	BLAST, Lek for grouping proteins into families using Smith-Waterman for all-against-all alignment, SMART and Pfam, Celera Panther Classification
human-mouse mRNA comparison	megaBLAST	not specified, probably from Otto (based on matches produced by SIM and BLAST)
Single Nucleotide Polymorphism finding	Polybayes, neighbourhood quality standard (NQS)	PowerBLAST

Table B.3: Software used in the human genome sequencing project.

We present supporting evidence for our analysis of the computing aspects of the Human Genome Project, presented n Chapter 3. This summary is based on the material from [226, 57] and various internet resources maintained by the members of the International Human Genome Sequencing Consortium. In Table B.1 we compare the data volumes used, produced or managed by the two groups which contributed to the draft sequence of the human genome. In Table B.2 we summarise the computing resources and in Table B.3 the software used by both projects. We then present our summaries and extracts from web sites prepared by the members of the Consortium. Where available, the descriptions of software are directly quoted from the source websites.

- FPC [204], http://www.sanger.ac.uk/Software/.

  FPC V4 (fingerprinted contigs) is an interactive program for building contigs from fingerprinted clones, where the fingerprint for a clone is a set

of restriction fragments. FPC has an algorithm to automatically cluster clones into contigs based on their probability of coincidence score. For each contig, it builds a consensus band (CB) map which is similar to a restriction map but it does not try to resolve all the errors. The CB map is used to assign coordinates to the clones based on their alignment to the map and to provide a detailed visualization of the clone overlap. FPC has editing facilities for the user to refine the coordinates and to remove poorly fingerprinted clones.

FPC was used for instance in the preparation of contigs for chromosome 21 [105], and relied on input via its graphical user interface.

- PHRED [77, 78] is a base-calling program available from http://www.sanger.ac.uk/-Software/. PHRED and PHRAP are also available from the authors' web site http://bozeman.mbt.washington.edu/. PHRED

  is an alternative to the standard ABI base-calling, and in tests makes fewer errors on average and calls accurately further into the read. Phred also generates a base-quality index for each base it calls, indicating the likelihood the call is correct. The assembly program Phrap can make use of these quality measures, both in assembly and when assessing the quality of the contigs' consensus sequence... It has been shown to make a fast (around 10-20 mins) and efficient assembly of a normal sized shotgun (600-800) reads into a reasonably small number of contigs (5-10 over 1 Kbp).

- BLAST [7, 8, 238] family of programs can be downloaded from http://www.ncbi.-nlm.nih.gov/BLAST/. BLAST stands for basic local alignment search tool and is the most popular sequence searching tool in biology. BLAST is discussed in the context of approximate matching, in Chapter 5.

- GigAssembler [133] is available from http://genome.ucsc.edu/goldenPath/algo.html. It merges human genome sequence fragments into the the human genome draft, based on mRNA, paired plasmid ends, EST, BAC end pairs, and other information.

- Whole genome assembler (WGA), an earlier version of which was described by Myers [165], consists of a Screener (repeat finding), Overlapper (all-against all matching), Unitigger (finding uniquely assembled contigs), Scaffolder (building scaffolds of contigs), and Repeat Resolver. Overlapper was the most compute-intensive part, consuming 10,000 CPU hours using 4 GB RAM, elapsed time 4 to 5 days with 40 4-processor alpha machines. Overall, WGA was allowed to use up to 100 GB RAM and took 20,000 hours.

  Compartmentalised shotgun assembly (CSA), applied to clusters of the whole sequence stretching over published BAC contigs (multiple megabase regions of the genome), was a variation on the WGA. Clusters were identified by taking BAC contigs from the public database, aligning those with Celera data and separating them from other sequence data. This analysis led Celera to conclude that some 240 Mbp of unique sequence from regions not covered by the public consortium was in their possession. Assemblies for each locale were performed consecutively, and then used

to build larger scaffolds and contigs which were finally placed on the genome maps using STS information (via electronic PCR) and human curation.

- Spidey, written by S. Wheelan (quoted as private communication in [57]) was used in gene analysis. No further information is available.

- Acembly is a

  > *graphic interactive program to support shotgun and directed sequencing projects,*

  written by Danielle and Jean Thierry-Mieg and Ulrich Sauvage, available at http://www.infobiogen.fr/doc/ACEDBdoc/Acembly.doc.html. It can also be run in an automatic mode and it *can use, to generate the assembly, both the sequences of the reads and various kinds of additional information, like multiple reads from the same subclone, mapping information or restriction fragment lengths.*

- Ensemble is at http://www.ensembl.org/. It is a pipeline of sequence analysis programs encompassing all types of gene prediction and feature detection. It presents an interface to a repository of sequence and annotation data, with navigational and query facilities based on HTML forms and a clickable GIF interface to genome maps. Data available at the web site are generated for each new sequence release and the full processing run using a complex combination of software modules takes several days. Data and software files are also available via FTP. The database used is Mysql [166]. A new genome browser to be used by Ensembl is being developed by the project Apollo.

- Smith-Waterman [203] is an exhaustive sequence similarity comparison program which examines the entire dynamic programming matrix in search of the highest similarity score which is then output along with the alignment. An implementation of this algorithm is available from http://www-hto.usc.edu/software/seqaln/. We refer to this algorithm in Chapter 5.

- SMART [198]

  > *allows the identification and annotation of genetically mobile domains and the analysis of domain architectures (http://SMART.embl-heidelberg.de). More than 400 domain families found in signalling, extra-cellular and chromatin-associated proteins are detectable. These domains are extensively annotated with respect to phyletic distributions, functional class, tertiary structures and functionally important residues. Each domain found in a non-redundant protein database as well as search parameters and taxonomic information are stored in a relational database system. User interfaces to this database allow searches for proteins containing specific combinations of domains in defined taxa.*

  SMART was used in the annotation of genes.

- ePCR [195, 196] can be used at http://www.ncbi.nlm.nih.gov/STS/. This software, based on hashing and an underlying database UniSTS, combining all known STS

resources, compares a query DNA sequence against the database of known STS amplimers (end sequences) and decides on the basis of amplimer distance and orientation if the genomic sequence contains any known STS sequences. This tool can be used to check if a piece of sequence in our possession can be placed on the map of known STSs. In the sequencing context, this is both a mapping and a quality control tool.

- MUMmer [66] is a global alignment tool for entire genomes or longer fragments of genomic sequence. It uses a suffix tree to find repetitive sequences in two genomes, and then attempts to create a global alignment where parts of the original sequences may be reshuffled.

- PolyBayes [150]

  > *is a computer program for the automated analysis of single-nucleotide polymorphism (SNP) discovery in redundant DNA sequences. The primary motivation for its development is to provide a general and reliable tool for the discovery of genetic variations in what is an exponentially increasing volume of sequence data in public and private databases. The software integrates algorithmic solutions to three of the main challenges in sequence-based SNP discovery: multiple sequence alignment, paralog identification and SNP detection.*

  Software is available at: http://genome.wustl.edu/gsc/Informatics/polybayes/.

- Neighbourhood quality standard (NQS) [191] is an SNP identification program. As both PolyBayes and NQS combine sequence analysis and statistics, they remain outside our focus.

- SIM is an alignment program which finds k best non-intersecting alignments between two sequences or within a sequence using dynamic programming techniques [109]. This software belongs in the area of multiple sequence alignment which we do not discuss in our thesis.

Appendix C

VLBD Journal invited paper — Database Indexing for Large DNA and Protein Sequence Collections

This paper is an invited submission to VLDB Journal in November 2001 by Ela Hunt, Malcolm P. Atkinson and Robert W. Irving.

Abstract

Our aim is to develop new database technologies for the approximate matching of unstructured string data using indexes. We explore the potential of the suffix tree data structure in this context. We present a new method of building suffix trees, allowing us to build trees in excess of RAM size, which has hitherto not been possible. We show that this method performs in practice as well as the $O(n)$ method of Ukkonen [224]. Using this method we build indexes for 200 Mb of protein and 300 Mbp of DNA, whose disk-image exceeds the available RAM. We show experimentally that suffix trees can be effectively used in approximate string matching with biological data. For a range of query lengths and error bounds the suffix tree reduces the size of the unoptimised $O(mn)$ dynamic programming calculation required in the evaluation of string similarity, and the gain from indexing increases with index size. In the indexes we built this reduction is significant, and less than 0.3% of the expected matrix is evaluated. We detail the requirements for further database and algorithmic research to support efficient use of large suffix indexes in biological applications.

Introduction

The potential for indexing

Indexing technologies speed up data searching and have been very successfully applied in many areas of data processing. Different indexing mechanisms have been developed, each particularly suited to the type of data it is indexing and the type of search that is required. Indexes like B-trees [59] are now standard in database systems, and some newer indexing structures[1] are making their way into commercial systems. Text indexing for English text

[1] http://solutions.altavista.com/

is very well advanced [234] but some data types, in particular biological sequence data or images still elude indexing, and no commercially available database system known to us can index DNA or protein strings. The challenge in this area is the fact that biological sequences are searched not exactly, but using approximate matching techniques. Navarro [168] in his recent survey says that approximate string matching using indexes is an important but underdeveloped area of research. Our interest lies in this area, and in particular in the application of string indexing to biological sequences. We investigate the suffix tree structure, show how to build suffix trees for any size of data, and demonstrate that this data structure can speed up biological sequence searching.

Biological sequences

DNA sequences, which hold the code of life for every living organism, can be abstractly viewed as very long strings over a four-letter alphabet of *A, C, G* and *T*. Proteins, which use an alphabet of 20 symbols, are translations from selected stretches of DNA, using a predefined translation table where each 3 letters of DNA translate to one amino-acid (AA).

Many projects to sequence the genome of some species are well advanced or concluded. The very large number of species (and their genetic variations) that are of interest to man, suggest that many new sequences will be revealed as the improved sequencing techniques are deployed. At the same time, proteins from those species are also being investigated and conceptual translations of entire genomes or their parts from DNA to protein are also made. Consequently, we are at a technical threshold. Techniques that were capable of exploiting the smaller collections of genetic data, for example via serial search, may require radical revision, or at least complementary techniques. As the geneticists and medical researchers with whom we work seek to search multiple genomes to find model organisms for the gene functions they are studying, we have been investigating the utility of indexes. The fundamental lack of structure in genetic sequences makes it difficult to construct efficient and effective indexes.

The length of a DNA sequence is measured in terms of the number of base pairs (bp), and only one base in each pair is represented, as the other base is its complement (A complements T and C complements G). Because of large genome sizes, gigabase pairs (Gbp) or megabase pairs (Mbp) are more convenient units. For example, mammalian genomes are typically 3 Gbp in length. The largest public database of DNA[2], which contains over 17 Gbp (October 2001), is an archive which holds indexes to fields associated with each DNA entry but does not index the DNA itself. In the industrial domain, Celera Genomics[3] have sequenced several small organisms, the human genome, and four different mouse strains. The volume of protein data is smaller, and the latest release of SWISSPROT and TREMBL databases[4] counts around 200 million AAs, i.e. 200 megabases (Mb), as AAs are stringed into single strands and not paired helices. However, protein searching dominates because functional similarities between distant species are best seen at the level of protein, where similar stretches for amino-acids code for similar spatial structures and chemically active sites.

[2] http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=Nucle otide

[3] http://www.celera.com

[4] http://www.expasy.org

Sequential scanning

Both DNA and protein sequences are accessed as flat files. Searching for similar sequences is usually carried out by sequentially scanning the data using a filtering approach [176, 7, 8], and discarding areas of low string similarity. Typically, this approach uses a large infrastructure of parallel computers. At the Sanger Centre, http://www.sanger.ac.uk, a farm of over 400 computers is available, and a large proportion of them are used in sequence similarity searching. The viability of this approach to searching depends on biologists being able to localise the searches to relatively small sequences, on skill in providing appropriate search parameters, and on batching techniques. Even under these circumstances it cannot always deliver fast and appropriate answers. Using BLAST on the hardware configuration described in Section C (and all 4 processors), we compared 99 queries[5] (predicted human genes of length between 429 and 5999 bp) to a BLAST "database"[6] for 3 human chromosomes (294 Mbp, 10% of the human genome). The search took 62 hours (average 37 minutes per query), with default BLAST parameters, and delivered 6559 hits with an average of 66.25 hits per query and a median of 34. Some hits spanned only 18 characters but those had very high similarity. 17 out of 99 queries came from the chromosomes stored in the BLAST "database" and they produced several exact hits each (corresponding to the non-contiguous nature of DNA strings contributing to human genes).

As there is a rapid rise in both the volume of data and the demand for searches by researchers investigating the mechanisms of cancer and inherited diseases like hypertension and diabetes, it is worth investigating the possibility of accelerating these searches using indexes.

The indexing potential

The appropriate indexes over large sequences can take many hours to construct, hence it is infeasible to construct them for each search[7]. On the other hand, the sequences are relatively stable, so that it may be possible to amortise this construction cost over many thousand searches. That depends on developing techniques for storing the indexes persistently, i.e. on disk. As we will explain, that has not proved straightforward, but we believe that we now have the prototype of a viable technology. We focus our attention on persistent suffix trees for reasons given below.

To our knowledge, no existing database technology can support indexed searches over large DNA strings and the feasibility of indexed searches over large strings is an open research question [26, 168]. Inverted files [234] are not suitable, because DNA cannot be broken into words. For the same reason the String B-tree [81] and a prefix index [127] may not be appropriate choices. Approaches based on q-grams [222, 169, 44, 157, 171] are fast and proven, but cannot deliver matches that have low similarity[8] to the query [168]. The suffix array [148] is the closest competing structure, as it needs less space than a suffix tree. This structure is under investigation [25] and might deliver fast searching for large sequence repositories. However, it is not obvious how best to scale it up. Other competing structures

[5] ftp://ftp.ensembl.org/current/data/fasta/cdna/ensembl. cdna.gz

[6] BLAST package includes a command *formatbd* which compresses the sequence and creates indexes of sequence names and occurrences of non-repetitive and repetitive DNA.

[7] For example, the most space efficient main-memory index would take 9 hours and 45 Gbytes of RAM to index the human genome [138].

[8] Low similarities are often biologically significant.

include the LC-trie [11] which is a compressed suffix tree and the suffix binary search tree (SBST) [125, 126]. The SBST can be viewed as a tree implementation of a suffix array and is more space efficient than a suffix tree. It performs very well in exact matching tests [120].

It appears that the suffix tree [232, 153, 224, 99] is a good candidate data structure for this type of indexing, but so far, suffix trees on disk could only be built for small sequences, due to the so-called "memory bottleneck" [79] or "thrashing" [36]. Baeza-Yates and Navarro [25] state that "suffix trees are not practical except when the text size to handle is so small that the suffix tree fits in main memory", and use a suffix array instead, which reduces the storage required for the index. In this paper we respond to this challenge, and show how to build large suffix trees. We also adapt the algorithm of [25] to the needs of biological sequence searching, i.e. to the calculation of sequence similarity and not edit distance. We focus on the indexing gain, i.e. the actual reduction in the size of the matrix comparison problem, which in the worst case is $O(mn)$, and show that for index sizes of 200-300 million letters, the actual matrix size is reduced from mn to $0.003mn$ or less, assuming suitable combinations of query length and error level.

Overview of the paper

The rest of this paper is structured as follows. Section C summarises previous work, and Section C introduces the suffix tree. Section C presents our new algorithm for the construction of very large suffix trees. The test data and experiments results with tree construction and exact matching are described in Section C. Section C discusses our algorithm for suffix-tree-based string similarity searching using the dynamic programming method (DP), and Section C presents the approximate matching results. The discussion of our work is in Section C. The paper closes with plans for further work, in Section C, and Conclusions.

Previous work

We first review persistent suffix tree construction and suffix tree storage optimisations. We then position the dynamic programming technique in the context of approximate matching. Finally, we focus briefly on biological applications which use approximate matching techniques.

Persistent trees

Persistent indexes to small sequences have been built previously. Bieganski [33], built persistent suffix trees up to 1 Mbp. Recently, Baeza-Yates and Navarro [170, 25] built persistent suffix trees for sequences of 1 Mbp using a machine with small memory (64 MB) and concluded that trees in excess of RAM size cannot be built. Farach's theoretical work to remove the I/O problem [79] reduces suffix tree creation complexity to that of sorting and extends the computational model to take into account disk access. The "memory bottleneck" is considered to lie in random access to the string being indexed. In our opinion, it is not only the source string itself but the tree data structure and the suffix links which contribute to the bottleneck. An empirical evaluation of that method has not been reported. The only recent accounts of large persistent suffix trees representing sequences of 20.5 Mbp are in our previous work [113, 120].

Optimisations

Optimisations of suffix tree structure were undertaken by McCreight [153], and more recently by Kurtz [138]. Kurtz reduced the RAM required to around 13 bytes per character indexed, for DNA (our measurements using Kurtz's code), but his storage schemes have not been tested on disk yet. We believe that some extra space overhead will be inevitable. Since Kurtz's tree uses suffix links, it may suffer from the same "memory bottleneck" if moved into the database world. It appears that further investigation in this direction is warranted.

Compact encodings of the suffix tree, based on a binary representation of the text, have been investigated by Munro and Clark [54, 55, 164] and Larsson [10, 140], but Munro [164] states that compact suffix trees will require too many disk accesses to make the structure viable for secondary memory use.

Approximate matching techniques

Recent overviews of approximate text searching methods [168, 26] are available and present a full classification of the available techniques. The techniques can be divided into dynamic programming (DP), automata, bit-parallelism and filtering.

	A	R	N	D	C	Q	→
A	4	-1	-2	-2	0	-1	
R	-1	5	0	-2	-3	1	
N	-2	0	6	1	-3	0	
D	-2	-2	1	6	-3	0	
C	0	-3	-3	-3	9	-3	
↓							

Table C.1: A fragment of BLOSUM62.

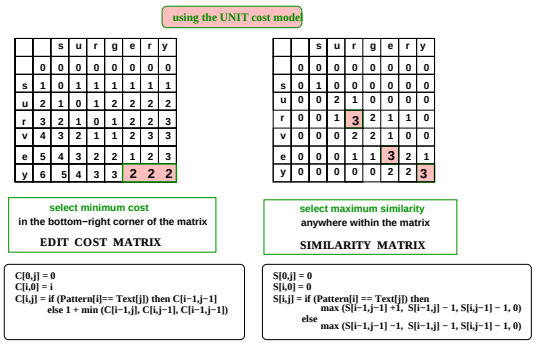


Figure C.1: Edit cost and similarity matrices for the comparison of the pattern *survey* with the text *surgery*.

DP is the technique of choice in the biological context. It involves the calculation of a matrix where one dimension is the text and the other the pattern. By using a cost function which rewards a match between any two characters, and punishes a mismatch or a character skip in the text or the pattern, an overall measure of sequence divergence can be calculated. In most theoretical work an edit cost is calculated which defines the number of transformations (inserts, deletes and replacements) which will mutate the text into the pattern, and unit edit

cost functions are used. In many biological contexts it is the similarity function which is of interest, and this is calculated in a similar manner but using a matrix which has a similarity value for each possible match or mismatch, like the BLOSUM62 matrix [104] shown in Table C.1. We illustrate the difference between the edit cost function and the similarity function in Figure C.1, where we show unit cost and similarity functions. The edit and similarity functions are further explained in [142, 203, 99, 177]. The complexity of matrix calculation is $O(mn)$ and optimisations of the DP calculation which reduce this complexity are known and widely used [168].

Automata can represent the pattern and be used to find portions of text which match it [154]. Representing the text as an automaton moves the problem into the area of text indexing in a database context which is our focus. A suffix tree [232] or a suffix array [148] can be viewed as such an automaton, and the factor oracle is a recent extension of the use of automata in text searching [6]. Our work combines an automaton with the DP calculation.

Bit-parallelism can be used in combination with a pattern automaton. This technique represents the text and the pattern as bit sequences, divides them into computer words, and performs fast comparisons based on register logic, for instance the SHIFT and OR functions [29]. It is a challenge to find appropriate logic functions to represent the automaton computation. In comparing biological sequences the cost function itself may be read from a matrix, so that bit parallelism combined with a pattern automaton is currently of limited use. Future work may however change this.

Filtering techniques [215] can be used to focus the search on parts of the text which potentially could harbour a match so that the DP calculation applies to less data and the overall complexity of text comparison is reduced below $O(mn)$. This approach can use different methods of text scanning or partitioning. Filtering used on its own is considered to deliver efficiently only the matches which are very close to the query [168].

These strictly algorithmic approaches to pattern matching are recently being complemented with approaches borrowed from the area of signal processing and data compression. Interesting new avenues have been opened by Kahveci and Singh [131] and Ferragina and Manzini [82]. Kahveci and Singh use the wavelet transform to map genomic strings to their local frequencies for different resolutions. They develop algorithms for both range and nearest neighbour queries and present experimental results for up to 30 Mbp of DNA sequence. Their technique is very promising. It needs to be investigated how to scale it up, and how to deal with gapped alignments. Ferragina and Manzini combine compression with a suffix array data structure and show that the performance of exact matching can be significantly improved. Approximate matching, however, is still a challenge in that context, and their DNA index is not large (4.6 million base pairs).

Biological applications

Biological applications use different combinations of these methods, and Gusfield [99] provides an in-depth treatment of most areas of biological string processing. We mention applications which are close to our focus of interest. BLAST [7, 8] combines a DP calculation with q-grams, filtering, automata and bit-based comparisons. It is a heuristic approach which cannot guarantee that all the significant matches are reported. BLAST is used very widely in many contexts and has been used extensively in human genome analysis [226, 57]. In pattern discovery and gene prediction suffix trees can be used [37, 66, 149, 225], along with other methods. Rocke [187], for instance, combines Gibbs sampling with a suffix tree

index in the area of gapped motif discovery. An application of persistent hash position trees which are close relatives of suffix trees was reported in the context of sequence assembly [156]. Persistent suffix arrays have been used in EST analysis [44, 157] and transient suffix trees in repeat finding [139]. Considerably less research has been done in the use of suffix trees for protein analysis, with the exception of recent work in the area of protein classification [67]. The constraint in the use of suffix trees so far has been the difficulty of building them on disk in excess of RAM size. We solve this problem and can therefore test the indexing gains on large indexes to both DNA and protein data.

Suffix trees

Suffix trees are compressed digital tries. Given a string, we index all suffixes, e.g. for a string of length 10, all substrings starting at index 0 through 9 and finishing at index 9 will be indexed. The root of the tree is the entry point, and the starting index for each suffix is stored in a tree leaf. Each suffix can be uniquely traced from the root to the corresponding leaf. Concatenating all characters along the path from the root to a leaf will produce the text of the suffix.

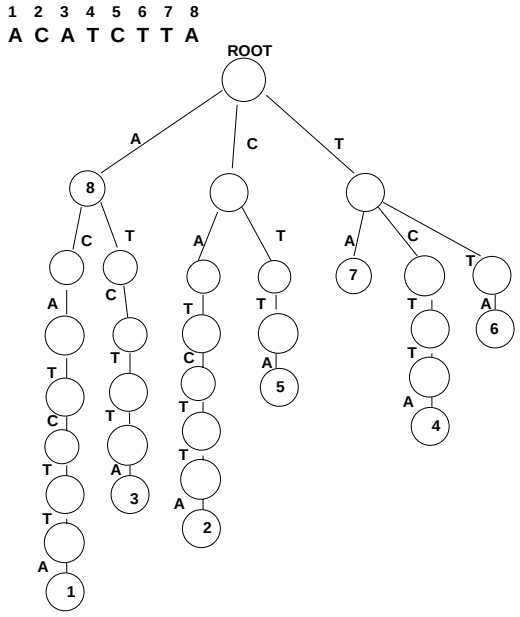


Figure C.2: An example trie on **ACATCTTA**.

An example digital trie representing **ACATCTTA** is shown in Figure C.2. The number of children per node varies but is limited by the alphabet size. This trie can be compressed to form a suffix tree, shown in Figure C.3.

To change a trie into a suffix tree, we conceptually merge each node which has only one child with that child, recursively, and annotate the nodes with the indices of the start and end positions of a substring indexed by that node. Commonly, a special terminator character is also added, to ensure a one-to-one relationship between suffixes and leaves (otherwise a suffix that is a proper prefix of another suffix would not be represented by a leaf — for

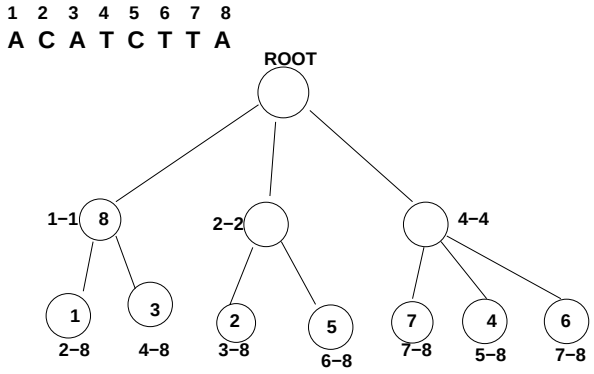


Figure C.3: An suffix tree on **ACATCTTA**.

instance node number 8 in Figure C.3). The change from a trie to a suffix tree reduces the storage requirement from $O(n^2)$ to $O(n)$ [232, 153, 224].

Most implementations of the suffix tree also use the notion of the suffix link [224]. A suffix link exists for each internal node, and it points from the tree node indexing *aw* to the node indexing *w*, where *aw* and *w* are traced from the root and *a* is of length 1. Suffix links were introduced so that suffix trees could be built in $O(n)$ time. However, in our understanding, they are also the cause of the so-called "memory bottleneck" [79]. Suffix links, shown in Figure C.4, traverse the tree horizontally, and together with the downward links of the tree graph, make for a graph with two distinct traversal patterns, both of which are used during construction. Ineluctably, at least one of those traversal patterns must be effectively random access of the memory. At each level of the memory hierarchy this induces cache misses. For example, it makes reliance on virtual memory impractical.

As would be expected from this analysis, we have observed very long tree construction times when using disk with the $O(n)$ suffix-link based algorithms. A first approach is to attempt to build the trees incrementally, checkpointing the tree after each portion has been attempted. Here, the suffix-link based algorithm exhibits another form of pathological behaviour. The construction proceeds by splitting existing nodes, adding siblings to nodes and filling in suffix-link pointers. As a result of the dual-traversal structure, no matter how the tree is divided into portions, a large number of these updates apply to the tree already checkpointed. This has the cost of installation reads and logged writes, if the checkpointed structure is not to be jeopardised. In addition, the checkpointed portions of the tree are repeatedly faulted into main memory by the construction traversals.

These effects combine to limit the size of tree that can be constructed and stored on disk using suffix-link based algorithms to approximately the size of the available main memory. For example, in Java, using 1.8 Gbytes of available main memory we could build transient trees for up to 26 Mbp of DNA sequence. Using the suffix-link based algorithm under PJama, checkpointing trees indexing more than 21 Mbp has not been possible [113, 120] (the reduction on using PJama is due to two effects: (i) it increases the object header size, and (ii) it competes for space, e.g. to accommodate the disk buffers and resident object table [21, 143]). We have therefore investigated incremental construction algorithms in which we forego the guarantee of $O(n)$ complexity.

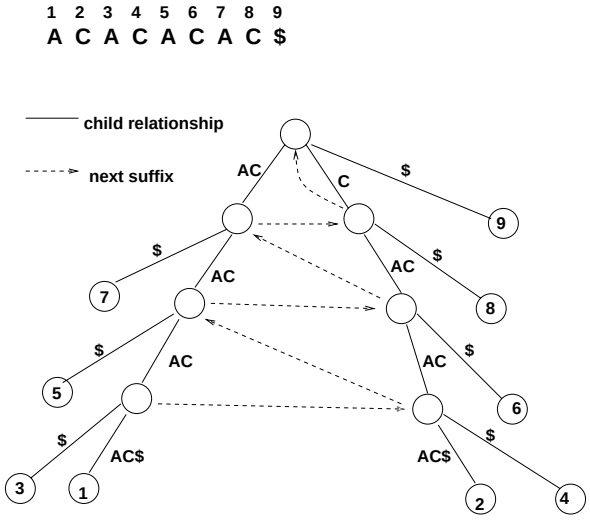


Figure C.4: Suffix tree and links on **ACACACAC$**.

Exact matching

Exact pattern matching in a suffix tree involves one partial traversal per query. From the root we trace the query until either a mismatch occurs, or the query is fully traced. In the second case, we then traverse all children and gather suffix numbers representing matches. The complexity of a suffix tree search is $O(k + x)$ where k is the query length and x the number of matches in the index. Looking for queries of length k may be expected to bring back a $\frac{1}{a^k}$ fraction of the whole tree, where a is here the size of the active alphabet. For example, a query of length 4 over a DNA index might retrieve $\frac{1}{256}$ of the tree, on average. Composite algorithms may be necessary, where short queries are served by a serial scan of the sequence, and longer queries use the index. The threshold at which indexing for exact matching begins to show an advantage depends on the precise data structure used, on the query pattern, and on the size of the sequence. We currently estimate this threshold to be in the region of minimum query length of 10 to 12 letters for single human chromosomes. For proteins, with a larger alphabet, much shorter queries can be evaluated efficiently.

The new tree construction algorithm

The new incremental construction algorithm trades ideal $O(n)$ performance for locality of access on the basis of two decisions:

1. to abandon the use of suffix links, and
2. to perform multiple passes over the sequence, constructing the suffix tree for a subrange of suffixes at each pass.

These are both necessary, and they result in a fan-like tree structure in which partitions can be built either consecutively or in parallel, see Figure C.5. Removing the suffix links means that the construction of a new partition corresponding to a different subrange does not need to modify previously checkpointed partitions of the tree. Using multiple passes,

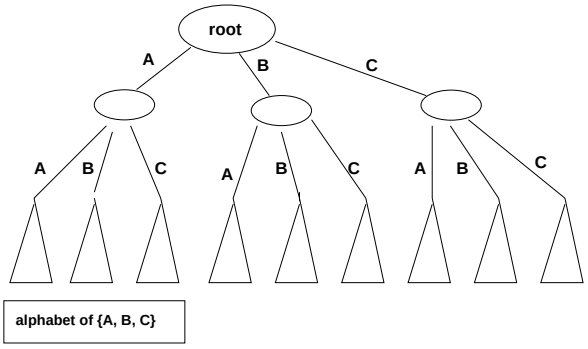


Figure C.5: A fan-like structure of the partitioned suffix tree, with 9 partitions, using prefix length 2.

each dealing with a disjoint subrange of the suffixes, means that it is not necessary to access or update the previously checkpointed partitions. Data structures for the complete partitions can be evicted from main memory and will not be faulted back in during the rest of the tree's construction. Thus the main memory is available for the next partition and its size is a determinant of the partition size and hence the number of passes needed. An additional benefit of this partitioned structure is that the probable clustering of contemporaneously checkpointed data will suit the lookup and search algorithms. Further details of our algorithm are now presented.

Phased tree construction

Several $O(n)$, suffix-link based, tree building algorithms are known [232, 153, 224, 79, 147], but they have not proved appropriate for large persistent tree construction undertaken by Navarro [170] or ourselves. In contrast, the algorithm we use is $O(n^2)$ in the worst case, but due to the pseudo-random nature of DNA, the average behaviour is $O(n \log n)$ *for this application* [212].

We base our partitions on the prefixes of each suffix, since the suffixes that have the prefix **AA** fall in a different subtree from those starting with **AC**, **AG** or **AT**. The number of partitions and hence the length of the prefix to be used is determined by the size of the expected tree and the available main memory. It may be the case that smaller partitions would be better because their impact on disk clustering would accelerate lookups, but this has yet to be investigated.

The number of partitions required can be computed by estimating the size of a main-memory instantiation S_{mm}, available for tree construction, and the number of partitions, p, is

$$\left\lceil \frac{S_{mm}}{A_{mm}} \right\rceil ,$$

where A_{mm} is the available main memory. The actual partitioning can be carried out using either of the two approaches we outline. One way is to scan the sequence once, for instance using a window of size 3 (sufficient for 286 Mbp of DNA and 2 GB RAM), count the number of occurrences of each 3-letter pattern, and then pack each partition with different

prefixes, using a bin-packing algorithm [59]. Alternatively, we can assume that, given the pseudo-random nature of DNA, the tree is uniformly populated. To uniformly partition, we calculate a prefix code, P_i, for each prefix of sufficient length, l, using the formula:

$$P_i = \sum_{j=0}^{l-1} c_{i+j} a^{l-j-1},$$

where c_k is the code for letter k of the sequence, and a is the number of characters in the alphabet[9]. The code of a letter is its position in the alphabet, i.e. **A** codes as 0, **C** codes as 1, etc. The minimum value for P_i is 0 and its maximum is $a^l - 1$. So the range of codes for each partition, r, is given by:

$$r = \left\lceil \frac{a^l - 1}{p} \right\rceil.$$

The suffixes that are indexed during the jth pass of the sequence have $jr \leq P_i < (j+1)r$. The structure of the complete algorithm is given as pseudocode below:

```
for j in partitions do
  for i in 0..totalLength do
   if suffix i is in
partition j
    new Node(i);
    insert node;
   endif
  endfor
  checkpoint;
endfor
```

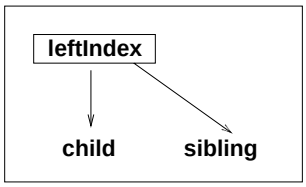


Figure C.6: Node of a thin naive tree.

A suffix tree node in our implementation consists of three fields: $child$ reference, $sibling$ reference and an integer, $leftIndex$, shown in Figure C.6. A new node represents a suffix stretching from position i to the end of the text string. It has $null$ child and sibling fields, and its $leftIndex$ set to i (its suffix number). Insertion starts from the root, and as the search for the insertion position proceeds down the tree, the left index is updated. This downward traversal matches the new suffix to suffixes which are already in the tree, and which share a prefix with the new suffix. When the place of insertion is determined, the node will either be added as a sibling to an existing node, or will cause a split of an existing node, see Figure C.7.

[9]Combinations of * can be used to denote unknowns, sequence concatenation and end of sequence. Hence a can be reduced to 5. In this case l set to 8 provides even division of partitions for all likely sequence length to available memory ratios.

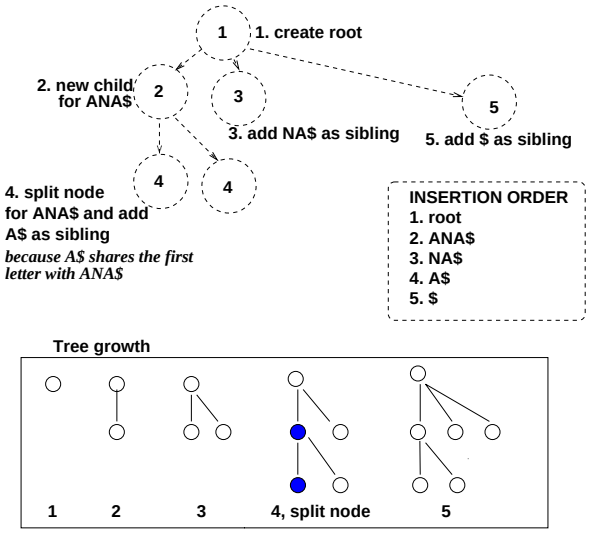


Figure C.7: Tree creation for **ANA$**.

Space requirements

Our new implementation disposes of suffix links. Further to that, we reduce storage by not storing the suffix number and the right index into the string for each node. The suffix number is calculated during tree traversal (during the search). The right pointer into the string is looked up in the child node, or, in the case of leaves, is equal to the size of the indexed string. Each tree node consists of two object references costing 4 B each (child, sibling), one integer taking up 4 B (leftIndex) and the object header (8 B for the header in a typical implementation of the Java Virtual Machine). The observed space is some 28 B per node in memory. The difference is due to PJama's housekeeping structures, such as the resident object table [143].

PJama's structure on disk adds another 8 B per object over Java, i.e. 36 B per node. The actual disk occupancy of our tree is around 65 B per letter indexed, close to that expected. The observed number of nodes for DNA and proteins remains between $1.6n$ and $1.8n$, where n is the length of the sequence, giving an expectation of between 58 and 65 bytes per node. Some of this space may well be free space in partitions, and some is used for housekeeping [178]. If we wanted to encode the tree without making each node an object, we would require 12 B per node, that is around 21 B per character indexed. But further compression could be obtained by using techniques similar to those proposed by Kurtz [138].

Tree building and exact matching in practice

In this section we summarise our experiments in tree building and exact matching on DNA strings. Results for protein data were analogous, and we do not report them here. We used DNA from 6 single chromosomes of the worm *C. elegans*, of 20.5 Mbp maximum[10] and

[10] ftp://ftp.sanger.ac.uk/pub/C.elegans_sequences/CHROMOSOMES/

some 286 Mbp merged DNA from human chromosomes 21, 22 and 1[11]. As queries we used short sequences, from the STS division of Entrez[12] for human data, and for the worm queries, short sequences called cDNAs. From each sequence initial characters were taken to be used as query strings.

Our alphabet in this experiment consists of **A, C, G, T**, a terminal symbol **$**, and ***** used as a delimiter for merged sequences.

Tests were carried out using production Java 1.3 for transient measurements, and PJama, see Section C, which is derived from Java 1.2 and uses JIT, for the persistence measurements. All timing measurements were obtained using SunOS 5.7 on an Enterprise 450 SUN computer with 2 GB RAM, and data residing on local disks. In this experiment our algorithm did not use multithreading and therefore only one of the four 300 MHz SPARC processors was used for the main algorithm. Parts of the Java Virtual Machine, and PJama's object store manager, will have made some use of another processor for housekeeping tasks.

The total number of lines of Java code for the 5 data structures examined was 3216, which includes over 10% lines of comments and print statements. The naive tree accounts for less than 550 lines of Java code.

The persistence platform

The first set of experimental trials of our algorithms was conducted using the PJama[13] platform [21, 17, 22, 23, 20, 23, 179, 100, 178]. We selected PJama to minimise the software engineering cost of our experiments. PJama enabled easy transitions between different underlying tree representations, and immediate transparent store creation from Java without any intermediate steps. Both transient and persistent trees can be produced using the same compiled code, but a different command-line parameter for PJama indicating whether a persistent store is being used.

Although tuned, purpose-built mechanisms will be appropriate for large-scale indexes, the cost of implementing them and maintaining them would be an impediment to rapid experimentation. In addition, a great many index technologies are proposed and tested, in this area of application, as well as many others. Hence, if we can make the general purpose persistence mechanism work for indexes, there could be considerable pay offs in reduced implementation times and more rapid deployment.

We are investigating other persistence mechanisms, including an object-oriented database, Gemstone/J[14], and tailored mapping to files. The latter may ultimately be necessary, given the data volumes and performance requirements. However, for the present, the general purpose object-caching mechanisms of PJama allow rapid experiments with a variety of index structures and matching algorithms.

Tree construction in memory

We compared two versions of the tree built using Ukkonen's algorithm [224], two versions of the naive tree of our construction, and the suffix binary search tree [125, 126]. The two trees constructed using Ukkonen's algorithm, time complexity of $O(n)$, differed only in

[11] ftp://ncbi.nlm.nih.gov/genomes/H_sapiens
[12] ftp://ncbi.nlm.nih.gov/repository/dbSTS/
[13] http://www.dcs.gla.ac.uk/pjama
[14] http://www.gemstone.com/products/j/

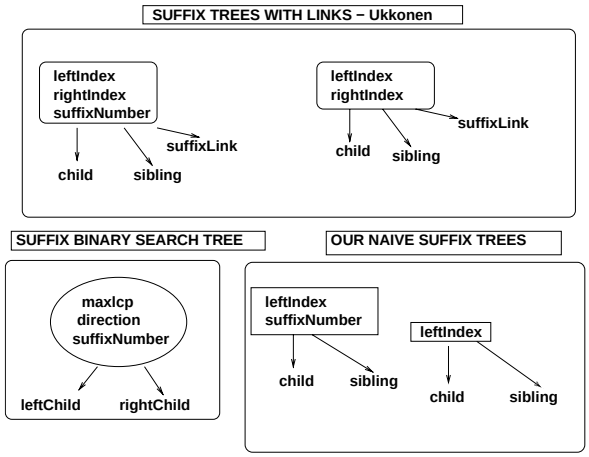


Figure C.8: Transient indexes built in memory.

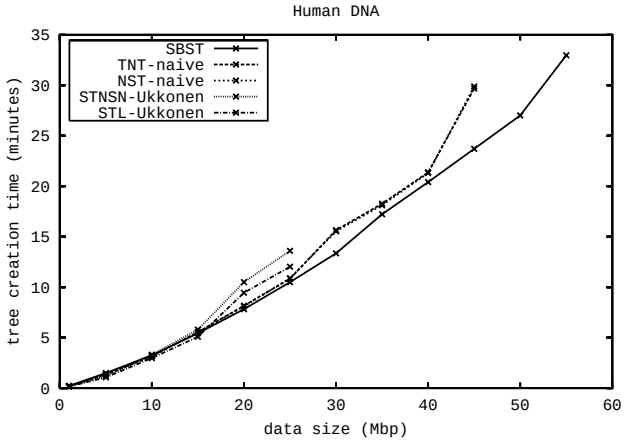


Figure C.9: Time required to build an index for human DNA, the graph for TNT is superimposed on the graph for NST.

one detail: one of them did not explicitly store the suffix number. The two naive trees, time complexity of $O(n^2)$, had an analogous structural relationship: no suffix number in one of the trees. The suffix binary search tree [125, 126] which builds in $O(n \log n)$ time had nodes consisting of a left child, right child, suffix number, maximum longest common prefix and a direction bit. It had the smallest overall space requirement, because per each text character only one node is needed, while the suffix tree needs up to two nodes per text character. We show the node layout of the structures we investigated in Figure C.8, and the comparison of index build times in Figure C.9. Surprisingly, tree creation times seem to be influenced predominantly by the space complexity of the data structure, and there appears to be no difference between linear and worse than linear construction algorithms. Our data for protein trees (not shown) exhibit the same behaviour, except that protein suffix trees are slightly more compact (have fewer nodes) and the suffix trees and the SBST take a little longer to construct.

A small persistent tree

We carried out tests with our implementation of the $O(n)$ tree building algorithm [224]. A tree for 20.5 Mbp of DNA was created in memory in 7 minutes on average. However, on disk, the creation time was around 34 hours, and checkpoints at 12 million and then every 0.5 million nodes were required. We used a 2 GB log file, and one store file of 2 GB. This was the largest tree of this type that we could build. It fitted mostly in memory (2 GB RAM, 2 GB store, some space needed for the JVM). Table C.2 shows the results obtained for a batch of 10,000 exact matching queries run on a cold store.

query length	avg time per query (ms)	total hits per batch
8	920	8,568,303
9	263	2,553,520
10	142	758,523
15	36	3,687
50	34	394
100	34	305
200	33	107

Table C.2: Cold store, a batch of 10,000 exact queries over 20.5 Mbp of worm DNA using an $O(n)$ index.

A large persistent tree

We then indexed 286 Mbp of DNA using the new suffix tree construction algorithm presented in this paper. The store required a 2 GB log and 19 GB in files of 2 GB maximum. Store creation time was 19 hours in our first run, and later 13.5 hours. Queries of the same length were sent in batches, without the use of multithreading[15].
We carried out exact string matching experiments on a cold store, see Figure C.10, and on a warm store, see Figure C.11. We observed that large batches produced faster response times, due to the benefit of objects that had been faulted in for previous queries still being cached on the heap.

Table C.3 demonstrates why the short queries take long to return results. The time of query evaluation can be divided into *matching* the query's text by descending the tree, and faulting in and traversing the subtree below the matched node to *report* the results. For short queries, many results are reported and the reporting time dominates because of slower access to secondary memory. For longer queries, fewer results are found, and the average query response improves. This observed slow performance for large result sets is partly due to using Java.lang.Vector class to store the matches. We have now removed this source of reporting inefficiency.

[15] In other experiments [120], we have demonstrated a significant speed up by using multiple threads to handle a batch of queries over a forest of suffix binary search trees.

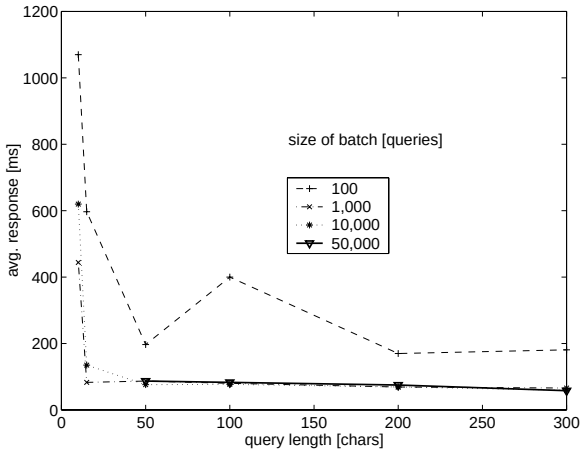


Figure C.10: Cold store query performance.

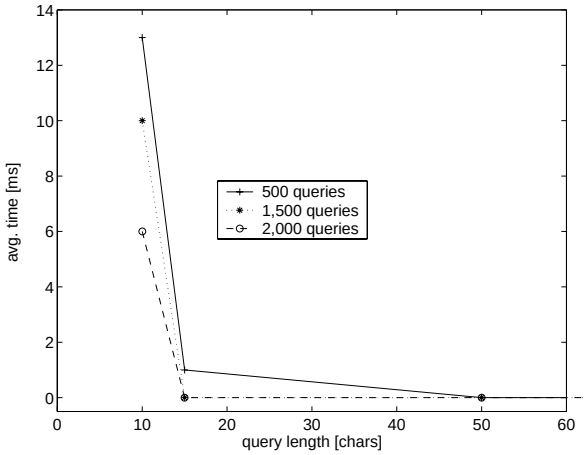


Figure C.11: Queries run over a warm store.

Approximate matching algorithm

Approximate searching in a suffix tree has been traditionally optimised based on the tree with suffix links, see the work by Ukkonen [223] and Cobbs [56]. As the DP calculation is carried out in a suffix tree with links, the last column of the alignment matrix is preserved along with a reference to the node it applies to. Following suffix links one can exclude from calculation all shorter suffixes. The space overhead of this optimisation is considerable as a record of nodes visited and the relevant matrix column have to be kept, and suffix links are needed. Baeza-Yates and Navarro [25] show that Cobbs' implementation of his method is slower than the depth-first tree traversal which they implement.

The approach we take follows [25], and is closely related to that of [187]. [25] assumes unit edit costs and uses a suffix array to simulate a suffix tree. A long query is broken into smaller substrings which are then used as individual queries, and when the index returns potential matches those will have to assembled into longer string alignments. The top of

batch size	query length	avg time per query (ms)	total hits per batch
100	10	1070	155,007
1000	10	444	1,289,800
10000	10	620	10,217,838
100	50	197	18
1000	50	87	221
10000	50	76	660
50000	50	87	25376

Table C.3: Cold store, batches of queries evaluated over a persistent index to 286 Mbp of DNA.

the suffix tree is scanned depth-first down to a maximum string length $k + m$ where k is the number of errors allowed in each small fragment, and m is the query length (a part of the original query). The DP calculation comparing the indexed text and the query is carried out. The minimum depth of traversal is $m - k$. After traversing $m - k$ characters if the edit cost is still 0, we have matched the query with k errors and can output a match. As consecutive columns are being calculated, if the edit cost becomes too high (there are not enough columns left to bring the edit cost down to the required maximum k), the calculation can stop early. The space overhead of this traversal is small, as the DP matrix has size $(m + 1) * (m + k + 1)$. The matrix is used as a stack with the index pointing to the current string depth of a node as measured from the root.

Our work builds on this algorithm but we make the following restrictions and changes. We use a suffix tree and not a suffix array and build a much larger index. We do not consider an edit function but a similarity function [203] so that our solution can be extended to deal with non-unit costs and gaps (i.e. an "an arbitrary score matrix" [187]) This precludes the use of fast bit arithmetic or an automaton for the pattern. We do not use filtration yet, as more work is required to refine this approach for use with protein similarity matrices. We implement the optimisation suggested by the authors which explores only the children of the root which start with the first $k+1$ letters of the query. Our main interest is in measuring the actual gain from using the suffix tree as opposed to carrying out the full matrix calculation.

We redefine the problem as follows: given the pattern of length m find all pattern occurrences which reach the threshold t, given a similarity function (our current function is +1 for match and -1 for mismatch or character skip). This implies using a matrix of size $(m + 1) * (2m - t + 1)$.

The suffix tree is traversed depth first and the DP calculation carried out using a rectangular matrix. The row and column zero are filled with nulls, as we are interested in finding local alignments. An index points at a matrix column and reflects current distance from the root in characters. The matrix is evaluated column-wise. Three conditions limit the depth of the tree traversal (and matrix calculation). These are:

1. break the DP calculation whenever the required similarity threshold is reached, find matches by traversing children,

2. stop calculating the matrix whenever we are certain that the current calculation will never reach the threshold,

3. break on reaching a separator or teminator character.

Condition 2 which is evaluated after each matrix column has been calculated is expressed as follows. $currentTextIndex$ is the array column number (equivalent to the distance in characters from the root). $2m - t$ is the last text position in the matrix, and we access the maximum score which was calculated in the current column, $maximumScoreInColumn$.

```
if currentTextIndex >= (2m - t) - t
then if (maximumScoreInColumn <=
        currentTextIndex - (2m - t) + t
        return
     endif
endif
```

This condition, will have to be refined for use with similarity functions used in biology, like the protein cost matrices PAM or BLOSUM [64, 104].

A full traversal of a suffix tree, down to the depth $2m-t$, could lead to performing more DP calculations than needed for the evaluation of the matrix spanning the entire text, i.e. $O(mn^2)$ (or $O(mn \log n)$ if we consider the pseudo-randomness of DNA data). This issue has been approached from the point of view of theory [25, 187] but our work concentrates on the engineering aspect and clearly distinguishes the gains from indexing from the effects of efficient matrix calculation or the possible loss of speed due to the use of a disk-resident index. We aim to discover what query and threshold lengths are appropriate so that we can guarantee that significantly fewer matrix columns are calculated than would be needed otherwise. To this aim we experiment with both DNA and protein indexes and with different query lengths and similarity thresholds.

Approximate matching results

A transient protein tree

Our first test concerns a transient naive tree indexing 36 Mbp of protein, i.e. the entire SWISSPROT database. We use human genes as queries, and break them into strings of 5 to 11 characters. We measure the number of DP matrix columns calculated during query evaluation. We take the *maximum* observed number of columns calculated in a given query and threshold combination and derive the *maximum* of the observed ratios relative to the DP matrix for the text of length 36 mln. Our results based on a sample of 1425 queries of varying lengths and thresholds are summarised in Table C.4. We observe that combinations of query length m and a threshold $m - 1$ or $m - 2$ deliver a speed up in comparison to full matrix calculation. A threshold value of $m - 1$ delivers a high efficiency gain and we observe that $less\ than\ 1\%$ of the full matrix is being evaluated. For the threshold of $m - 2$ the speed up is limited. We notice approximately two orders of magnitude ratio between the number of columns reported for $m - 2$ and $m - 1$. Thresholds of $m - 3$ and $m - 4$ increase the size of the matrix calculation.

A persistent protein index

A more significant indexing gain obtains for the persistent tree indexing 200 Mb of protein (all of SWISSPROT and TREMBL data) which we now present. This data is based on the evaluation of 10-15 queries for each combination of query length and threshold, and

threshold	query	ratio
4	5	0.0015
4	6	0.2489
5	6	0.0022
4	7	2.5200
5	7	0.3222
6	7	0.0033
4	8	6.9729
5	8	2.8402
6	8	0.3770
7	8	0.0039
6	9	2.9439
7	9	0.4234
8	9	0.0052
7	10	3.4541
8	10	0.5437
9	10	0.0062
9	11	0.5786
10	11	0.006

Table C.4: The fraction of 36 mln columns calculated for a range of thresholds and query lengths over a transient suffix tree index for 36 Mb of protein sequence.

the *maximum* observed number of columns evaluated divided by 200 mln, and shown in Table C.5. The total number of queries executed over this data set was 312. In a larger tree at the threshold equal to $m-1$ an even smaller portion, $less\ than\ 0.3\%$, of the full DP matrix is evaluated. For the threshold of $m-2$ the indexing gain is slightly more significant as well. This leads us to believe that for larger protein indexes the threshold of $m-2$ will also be beneficial in practice.

A persistent human DNA index

Table C.6 presents the indexing gain for a DNA suffix tree indexing 286 Mbp, i.e. 10% of the human genome, where both human and yeast DNA sequences[16] were used as queries. Short DNA queries report too many matches to be of use in sequence searching and we do not show them here. The results are based on 1334 queries of varying lengths and thresholds. We find that indexing DNA pays off significantly, and queries with the threshold of $m-2$ reduce the size of the DP calculation to 1% of the original matrix or less, while queries with just one mismatch, i.e. threshold equal to $m-1$ evaluate between 0.01% and 0.09% of the matrix. It appears that this behaviour holds irrespective of the origins of the DNA in the index and in the query. The threshold of $m-3$ also offers some reduction in matrix size and we expect the indexing gain to increase for larger indexes.
These results demonstrate clearly that the potential of suffix tree indexing might be considerable by delivering the benefits of the full DP calculation at a reduced cost. As the DP calculation generally dominates the time needed to perform a search, these data point to the fact that further work in this direction might lead to the development of a more efficient solution to approximate pattern matching.

[16]We used yeast chromosome 1 from ftp://genome-ftp.stanford.edu/pub/yeast/.

threshold	query	ratio
3	5	0.0474
4	5	0.0003
4	6	0.0697
5	6	0.0004
5	7	0.0807
6	7	0.0006
6	8	0.1088
7	8	0.0007
6	9	1.4185
7	9	0.1274
8	9	0.0010
7	10	1.6888
8	10	0.1606
9	10	0.0012
9	11	0.1830
10	11	0.0013
10	12	0.2035
11	12	0.0016
10	13	2.4061
11	13	0.2260
12	13	0.0022
12	14	0.2636
13	14	0.0024
13	15	0.3012
14	15	0.0026
14	16	0.3166
15	16	0.0029

Table C.5: Ratio of columns calculated to 200 mln, based on a suffix tree indexing 200 Mb of protein data.

Indexing performance

We now consider further issues relevant to the performance of index-based approximate matching.

- Is the prototype index implementation fast enough and how does it compare with carrying out the same DP calculation in memory without an index?
- Which of the query and threshold combinations deliver manageable numbers of matches, as all matches for a longer query broken into smaller parts will have to be merged?

We now address these questions.

Our performance comparisons are calculated as follows. We measure the query execution time and divide it by the size of the DP matrix which represents the product of the text and query. The formula we use is

$$\frac{time\,(sec)}{text\,(Mb) \times query}.$$

We use two benchmarks. One is BLAST which is optimised for use with multiprocessor machines. Running BLAST on the same data set and queries (using protein data) we measure

threshold	query	yeast ratio	human ratio
7	8	.0001	.0001
8	9	.0001	.0001
8	10	.0029	.0027
9	10	.0001	.0001
9	11	.004	.003
10	11	.0002	.0002
10	12	.0059	.0055
11	12	.0003	.0003
11	13	.0076	.0073
12	13	.0003	.0003
11	14	.1103	.121
12	14	.0111	.0104
13	14	.0004	.0004
13	15	.0129	.0116
14	15	.0005	.0006
15	16	.0007	.0008
16	17	.0008	na
17	18	.0009	na

Table C.6: Ratio of MAX columns calculated to 286 mln, based on a suffix tree for 286 Mbp indexing human genomic DNA and human and yeast DNA queries.

the size of the data processed (query size $\times$ database size in Mb, referring to uncompressed data sizes) and the time needed to process the query using 4 processors. Analysis of several runs on our computer, using protein data, yields the average ratio of time in seconds to matrix size to be:

$$2.39 \times 10^{-7}\ sec/Mb.$$

The other benchmark is a full DP matrix calculation, using the same data and matrix evaluation software (but without a suffix tree) and one processor (as in our suffix tree tests). To calibrate the DP calculation in memory we used a circular buffer twice the query length and Java version 1.3. This yielded the equation for the same relationship as being

$$time(sec) = 1.16 * matrixSize(Mb) + 66.0.$$

The purpose of those two benchmarks is first to measure the gain of our unoptimised prototype implementation against a similarly unoptimised matrix calculation — i.e. we are comparing like to like, with both implementations built under similar assumptions in Java. The second benchmark, BLAST is a comparison with a fully tuned software program which was developed over the years by a large team of specialists, and uses most of the known optimisations.

To calibrate the size of the result set returned by the query, we imagine a hypothetical query of 300 characters broken into short queries. We can then calculate the number of matches returned for a query of length 300. As our data have a symmetrical distribution with no outliers, we use the $average$ number of results, and $query$ length, and obtain the expected result set size as

$$matches = averageResultSize \times \frac{300}{query}.$$

We present the expected number of hits for a query of 300 characters and the ratio of time (seconds) to Mb (query size x database size) in Table C.7, for the protein data set of 200 Mb.

threshold	query	matches	seconds:Mb
3	5	15610540	1.8536
4	5	446953	.0187
4	6	679267	1.2842
5	6	18150	.0178
5	7	52780	.9954
6	7	1719	.0179
6	8	2087	.9773
7	8	258	.0103
7	9	405	.8622
8	9	136	.0137
8	10	273	.9787
9	10	147	.0475
9	11	164	.8038
10	11	67	.0223
10	12	244	.8529
11	12	128	.0352
11	13	185	.7919
12	13	87	.0401
12	14	133	.5443
13	14	62	.0504
13	15	136	.5683
14	15	60	.0468
14	16	135	.6819
15	16	65	.0604

Table C.7: Average expected number of partial matches for a query of length 300, and time to DP matrix size ratio in seconds per Mb for the index to 200 Mb of protein.

In Table C.8 we present the analogous results for the DNA index over 286 Mbp of the human genome and human and yeast queries.
A careful examination of both data tables reveals that the performance of the protein index is much slower than the performance of the DNA index. This mirrors the difference in the indexing gain observed for both data sets, and is related to the topology of both indexes which reflects two different alphabets. We believe that this issue requires further analysis. This could include gathering statistics on the number of nodes present at each string depth, as measured from the root, and a comparison with the actual number of nodes visited for all the combinations of query length and threshold.

In this experiment we knew that reaching the speed of BLAST would not be possible, firstly because we are using Java, and secondly because of the unoptimised matrix calculation. On the other hand, the comparison with the unoptimised DP calculation shows considerable efficiency gains. Current measurements indicate that the persistent platform always delivers a faster match for the threshold of $m - 1$ for both DNA and proteins. The threshold of $m - 2$ which offers a speed up in DNA matching is currently not attractive for proteins, but might become so for larger indexes.

We now look at the size of the returned result set. This is a significant consideration, as partial hits will have to be merged in search for longer string alignments. We assume

thres-hold	query	human matches	human sec:Mb	yeast matches	yeast sec:Mb
7	8	2212283	.0074	2291426	.0064
8	9	832903	.0029	702183	.0023
8	10	2018651	.0442	1617593	.035
9	10	51945	.0024	179303	.0011
9	11	881884	.0322	599274	.0382
10	11	179075	.0015	45276	.001
9	12	1172425	.9131	na	na
10	12	434412	.0448	110891	.0348
11	12	154098	.0021	9846	.001
10	13	448850	1.0419	na	na
11	13	130782	.0443	52516	.0386
12	13	85296	.003	74928	.0012
11	14	242862	.9658	221089	.7255
12	14	65970	.0459	7485	.055
13	14	58277	.003	5041	.0011
12	15	676255	.896	na	na
13	15	216599	.0549	1820	.05
14	15	53030	.0039	534	.0009
14	16	137673	.0624	na	na
15	16	49543	.004	169	.001
16	17	na	na	54	.0009
17	18	na	na	0	.0013

Table C.8: Average expected number of partial matches for a query of length 300, and time to DP matrix size ratio in seconds per Mb, for the index to 286 Mbp of human DNA.

here that dealing with results sets larger than 100,000 hits is not feasible, and base our considerations on a hypothetical query of 300 characters. It turns out that for the current size of SWISSPROT and TREMBL databases the minimum viable query length is around 7 characters, and this guarantees fewer than 100,000 partial hits. Probably with query length 8 and thresholds 6 and 7 the number of hits returned will be easier to process. However, the indexing gain for query length 8 and threshold 6 is currently modest, so threshold 7 seems to be the only viable choice. For the DNA index we tested, useful query and threshold values are query length 14 and thresholds 12 and 13. For both thresholds the indexing gains is satisfactory (1% and 0.04%, respectively). For longer queries the indexing gain is more significant but the sensitivity will descrease.

Discussion

We first discuss the new algorithm for the tree construction, and then turn our attention to the approximate matching problem. The new incremental algorithm for constructing disk-resident suffix trees without suffix links appears to have the potential to build arbitrarily large indexes efficiently. We are optimistic that this construction and the subsequent index use behaviour can be made sufficiently efficient that it will be a useful component of biological search systems. Some of the support for this claim is now presented.

Theoretical investigations of suffix tree building indicate that the use of suffix links to obtain an $O(n)$ algorithm is worthwhile. However, suffix links require space, and generate a difficult load on memory, with scattered updates and reads. In Figure C.12 we show *in-*

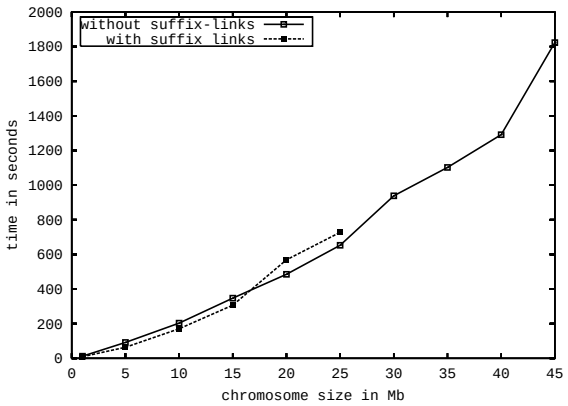


Figure C.12: Tree creation in memory.

memory performance comparison of suffix trees with and without suffix links. We show a close-up of tree creation times for two trees: a modified version of Ukkonen's algorithm [224] which does not perform a final tree scan to update the right text pointer in the leaves, and compare it to our tree without suffix links. We are limited here by 2 GB RAM, and carry out the tests using Java 1.3 with flags -server -Xmx1900m. The largest suffix-link tree we can build in this space is for 25 Mbp. Up to that value, no significant difference in tree construction speed can be observed (times are best times observed over several tree builds).

The incremental partitioned construction algorithm uses a partition size which we select. So far our experiments suggest that this should correspond to between 5 and 20 Mbp. This means that we are building the tree in a region where the $O(n)$ suffix-link algorithm offers no significant advantage.

The comparison of unoptimised *persistent* tree building times shows that our algorithm outperforms the suffix-link tree in terms of size, and we believe that building times in the region of 5 hours for the longest human chromosome will be possible. Our algorithm is scalable and can be adjusted to run on computers with different memory characteristics. More work is required to optimise the tree building, and to investigate the object placement on disk and its influence on query performance. Our algorithm opens up the perspective of building suffix trees in parallel, and the simplicity of our approach can make suffix trees more popular. In the parallel context, maintaining suffix links between different tree partitions may not be viable or necessary, as further characterisation of the space-time tradeoff between suffix trees with links and without is needed.

The approximate matching algorithm which we implemented and tested with large biological data sets shows clearly that indexing of sequence suffixes is beneficial, and can significantly reduce the size of the DP matrix calculation needed in query evaluation. We have no access to similar results for any of the known filtering approaches, including BLAST, and cannot make an appropriate comparison. Our approach combines filtering and the similarity calculation in one step, and this might also make a comparison with other approaches difficult, until we develop a fully optimised data structure, a well-tuned database layer, and a robust implementation of the matrix calculation.

The actual performance of the tree compares well with the analogous matrix calculation in Java, and future speedups can be delivered by using a combination of optimisation techniques which we are investigating. Those may include compression of the data struc-

ture itself (and departure from our simple object-oriented implementation), data clustering techniques, caching techniques and algorithmic optimisations.

The BLAST implementation we used as a benchmark shows how far our prototype is from becoming a product, and that further research is needed. We believe that implementing BLAST on top of a suffix index, for large datasets like the human genome, could deliver both faster searches, and, more significantly, would require fewer CPUs to carry them out. Our work clearly demonstrates that indexing may bring expected benefits in the area of biological searching.

Future work

Future work can be divided into the following interrelated parts.

- Improvements to the tree representation (data structure compression) and to the incremental construction algorithm.
- Investigation of the interaction between approximate matching algorithms and disk-based suffix trees.
- Investigation of alternative persistent storage solutions.
- Refinement of the cut-off used in the depth traversal of the tree, reflecting different similarity functions and gap costs.
- Post-processing of results to build longer alignments including gap costs and overall similarity measures.
- Integration of the algorithms with biological research tools, and usability studies.

Improving the tree representation is amenable to several strategies. We are investigating the replacement of the top of each tree with a sparse array indexed by P_i. We have also identified significant savings by specialising nodes (similar to some aspects of Kurtz's compression) and we are measuring the gains from storing summaries to accelerate reporting.

At the underlying object store level, we are looking at compressions that remove the object headers, at placement optimisations, and at improved cache management. We are experimenting with direct storage strategies.

As the deployed system will need to be trustworthy for biologists, we started field trials using Gemstone/J rather than PJama which is no longer maintained at the PEVM level [143, 23]. This will enable us to operate on other hardware and operating system platforms and to verify that the phenomena so far observed are not artifacts of PJama. Gemstone/J uses a similar implementation strategy to PJama, modifying the JVM to add read and write barriers. This provides comparable speed for large applications and nearly the same programming convenience. We plan to return to research into optimised persistent virtual machines once an optimised open source VM is available.

We are currently adopting biological measures of sequence similarity [7, 203] and concentrating on alternative persistence implementations. Our ultimate aim is to enable comparisons of different species based on DNA and protein sequence similarity. Future matching methods will be accompanied by statistical measures of sequence similarity, and will be

presented in the context of other biological knowledge. We see that future to lie in a uniform database approach to all types of biological data, including sequence, genome maps, protein structure, protein function, and gene expression data.

We plan to investigate several applications of suffix trees to biological problems. One of them is the identification of repeating sequence patterns on a genomic scale. Some of those patterns, positioned outside gene sequences, point to regulatory sequences controlling gene activity. We will also use our trees in gene comparison within and across species. Because of the RAM limit on suffix tree size, all-against-all BLAST is traditionally used in this context [57, 226], and it would necessitate up to

$$2\begin{pmatrix} 40000 \\ 2 \end{pmatrix}$$

gene alignments to perform full gene comparison within the human genome which has around 40,000 genes[17]. The use of large suffix trees in this context is likely to be beneficial. Finally, assembly of genomes can be speeded up using suffix trees [99].

Conclusions

An algorithm has been developed that promises to overcome a long standing problem in the use of suffix trees. It enables arbitrarily large sequences to be indexed and the suffix tree built incrementally on disk. Surprisingly, there seems to be no measurable disadvantage to abandoning the suffix links that have been introduced to achieve linear-time construction algorithms.

It has also been demonstrated that large suffix trees can reduce the required DP matrix calculation to a small fraction of the matrix. Both DNA and protein data sets benefit from indexing, and this benefit increases with the size of the index. This means that for genomic data sets indexing can potentially deliver faster query processing. Much further experimentation and analysis is required to develop confidence in these early, but intriguing results.

[17] Blast is run in both directions because it is an asymmetric matching algorithm.

Appendix D

Approximate Matching Test Log and Analysis

This appendix provides summary data and data analysis performed on the transient suffix tree index for 36 million bases of AA and a persistent index for 200 Mb of AA, as described in Chapter 6.

It consists of the following parts.

- The original output from our Java tests for the transient index.
- The output for the persistent index.
- SQL commands used to generate views containing summary results.
- SQL commands executed to retrieve data from the views and their output.

Output files for testing are as originally produced, but for legibility, they have been merged from individual runs and repetitive headers were removed, as well as some lines where a null query or query shorter than intended was submitted, due to intervening carriage returns in the gene file used as queries. SQL data files were produced using ORACLE8.

The transient results

The data format is as follows.

1. Column 1 is the query itself.
2. Column 2 is the threshold.
3. Columns 3-5 consist of 2 numbers and the times symbol. They represent the size of the DP matrix calculated for this query. For instance 11 x 19 signifies that the query length was 11 and the text dimension of the array was 19. The formula we used was $textDimension = 2 * queryDimension - threshold$.
4. Column 6 is the number of matches recorded.
5. Column 7 is the number of nodes visited in this traversal. The contents of this column may be used in database performance tuning in future work but are not analysed further in this thesis.

6. Column 8 is the number of matrix columns for which a DP calculation was carried out.

7. Column 9 is the time in ms needed for this traversal. This time measurement is the total elapsed time for the entire query evelautaion, and contains within it possibly considerable component of garbage collection time. We had to call the garbage collector using $System.gc()$ Java call after each part of the traversal which found matching leaves. Without this call we were not able to carry out any queries on the large PJama store.

```
Depth tree built in 3637102 ms
query thresh DPsize Matches nodes seen depth calculated timeMs
MASPSRRLQTK 3 11 x 19 478982 13823838 1207106956 1105260
MASPSRRLQTK 4 11 x 18 25877 12410927 924568194 862621
MASPSRRLQTK 5 11 x 17 1244 8782694 537401689 513515
MASPSRRLQTK 6 11 x 16 30 8523042 402341401 394654
MASPSRRLQTK 7 11 x 15 10 6638976 218135500 223989
MASPSRRLQTK 8 11 x 14 8 6089555 113637799 130352
MASPSRRLQTK 9 11 x 13 6 1539977 17755375 23819
MASPSRRLQTK 10 11 x 12 4 19073 211288 279
VITCFKSVLLI 3 11 x 19 572529 15628847 1358461049 1275196
VITCFKSVLLI 4 11 x 18 22908 12107165 903193148 855516
VITCFKSVLLI 5 11 x 17 952 11822442 720004043 692274
VITCFKSVLLI 6 11 x 16 33 9282192 436655208 431402
VITCFKSVLLI 7 11 x 15 9 7144013 235253700 243528
VITCFKSVLLI 8 11 x 14 7 5406208 100125146 115384
VITCFKSVLLI 9 11 x 13 5 1568189 18012852 24570
VITCFKSVLLI 10 11 x 12 2 18938 209561 279
TFIFWITGVIL 3 11 x 19 279711 10233707 883686199 825973
TFIFWITGVIL 4 11 x 18 13353 7990100 588209424 555693
TFIFWITGVIL 5 11 x 17 542 5568270 337938843 325368
TFIFWITGVIL 6 11 x 16 70 5391810 252268302 248479
TFIFWITGVIL 7 11 x 15 31 5164512 168544189 173910
TFIFWITGVIL 8 11 x 14 16 4388298 79405403 92343
TFIFWITGVIL 9 11 x 13 8 1307694 15113879 23451
TFIFWITGVIL 10 11 x 12 4 18707 207691 392
AVGIWGKVSLE 3 11 x 19 479922 13629301 1190063270 1138482
AVGIWGKVSLE 4 11 x 18 19391 11233290 837725152 792468
AVGIWGKVSLE 5 11 x 17 866 10961864 668500074 642507
AVGIWGKVSLE 6 11 x 16 53 8794178 413683754 407646
AVGIWGKVSLE 7 11 x 15 18 8416698 277428305 286420
AVGIWGKVSLE 8 11 x 14 14 7387961 137446474 158147
AVGIWGKVSLE 9 11 x 13 8 1824682 20830898 28407
AVGIWGKVSLE 10 11 x 12 4 18868 209011 281
YFSLLNEKATN 3 11 x 19 732887 16238329 1419041701 1333837
YFSLLNEKATN 4 11 x 18 36559 13683026 1027699277 970312
YFSLLNEKATN 5 11 x 17 2152 11500499 705663750 676313
YFSLLNEKATN 6 11 x 16 67 9179061 435392067 427676
```

YFSLLNEKATN 7 11 x 15 10 7444247 245262776 251986
YFSLLNEKATN 8 11 x 14 8 6832081 128108970 147042
YFSLLNEKATN 9 11 x 13 4 1498991 17326947 23254
YFSLLNEKATN 10 11 x 12 4 19373 214797 290
PFVLIATGTVI 3 11 x 19 714991 17401905 1506901275 1412062
PFVLIATGTVI 4 11 x 18 44715 17294015 1282432382 1211841
PFVLIATGTVI 5 11 x 17 2506 14652845 889693266 855454
PFVLIATGTVI 6 11 x 16 116 12350150 577024657 569079
PFVLIATGTVI 7 11 x 15 10 9572748 312397712 323956
PFVLIATGTVI 8 11 x 14 6 7084101 129973415 150473
PFVLIATGTVI 9 11 x 13 5 1477863 17054180 22927
PFVLIATGTVI 10 11 x 12 1 17448 193424 261
LLGTFGCFATC 3 11 x 19 356390 12072934 1035043933 971204
MASPS 4 5 x 6 1354 9188 45865 175
MASPS 5 5 x 5 26 101 405 2
MASPS 5 5 x 5 26 101 405 2
MASPS 5 5 x 5 26 101 405 2
RLQTK 4 5 x 6 1022 10275 51295 109
RLQTK 5 5 x 5 17 99 395 19
RLQTK 5 5 x 5 17 99 395 2
RLQTK 5 5 x 5 17 99 395 2
VITCF 4 5 x 6 256 9091 45415 141
VITCF 5 5 x 5 8 98 390 1
VITCF 5 5 x 5 8 98 390 2
VITCF 5 5 x 5 8 98 390 2
SVLLI 4 5 x 6 3148 9611 47960 101
SVLLI 5 5 x 5 64 101 405 2
SVLLI 5 5 x 5 64 101 405 2
SVLLI 5 5 x 5 64 101 405 1
TFIFW 4 5 x 6 296 8051 40280 84
TFIFW 5 5 x 5 4 101 405 2
TFIFW 5 5 x 5 4 101 405 2
TFIFW 5 5 x 5 4 101 405 1
TGVIL 4 5 x 6 2088 10725 53535 107
TGVIL 5 5 x 5 81 98 390 2
TGVIL 5 5 x 5 81 98 390 1
TGVIL 5 5 x 5 81 98 390 1
AVGIW 4 5 x 6 1228 9346 46660 94
AVGIW 5 5 x 5 24 100 400 1
AVGIW 5 5 x 5 24 100 400 1
AVGIW 5 5 x 5 24 100 400 2
KVSLE 4 5 x 6 293 8077 40290 76
KVSLE 5 5 x 5 0 49 145 1
KVSLE 5 5 x 5 0 49 145 1
KVSLE 5 5 x 5 0 49 145 1
YFSLL 4 5 x 6 1561 8930 44575 91
YFSLL 5 5 x 5 22 100 400 2

YFSLL 5 5 x 5 22 100 400 2
YFSLL 5 5 x 5 22 100 400 2
EKATN 4 5 x 6 1091 9485 47345 94
EKATN 5 5 x 5 102 97 385 2
EKATN 5 5 x 5 102 97 385 2
EKATN 5 5 x 5 102 97 385 2
PFVLI 4 5 x 6 535 8554 42700 83
PFVLI 5 5 x 5 24 92 360 2
PFVLI 5 5 x 5 24 92 360 1
PFVLI 5 5 x 5 24 92 360 4
TGTVI 4 5 x 6 1846 10452 52165 105
TGTVI 5 5 x 5 46 139 595 2
TGTVI 5 5 x 5 46 139 595 2
TGTVI 5 5 x 5 46 139 595 2
LLGTF 4 5 x 6 2069 6409 31945 68
LLGTF 5 5 x 5 86 156 680 2
LLGTF 5 5 x 5 86 156 680 2
LLGTF 5 5 x 5 86 156 680 2
CFATC 4 5 x 6 118 6742 33715 66
CFATC 5 5 x 5 0 59 195 1
CFATC 5 5 x 5 0 59 195 1
CFATC 5 5 x 5 0 59 195 1
ASAWM 4 5 x 6 231 7764 38790 75
ASAWM 5 5 x 5 5 137 585 2
ASAWM 5 5 x 5 5 137 585 1
ASAWM 5 5 x 5 5 137 585 2
KLYAM 4 5 x 6 361 9254 46185 91
KLYAM 5 5 x 5 0 63 215 1
KLYAM 5 5 x 5 0 63 215 2
KLYAM 5 5 x 5 0 63 215 1
LTLVF 4 5 x 6 1984 10612 52970 110
LTLVF 5 5 x 5 68 135 575 3
LTLVF 5 5 x 5 68 135 575 2
LTLVF 5 5 x 5 68 135 575 2
VELVA 4 5 x 6 2451 9658 48195 107
VELVA 5 5 x 5 92 119 495 6
VELVA 5 5 x 5 92 119 495 17
VELVA 5 5 x 5 92 119 495 2
IVGFV 4 5 x 6 1051 9578 47800 99
IVGFV 5 5 x 5 21 101 405 2
IVGFV 5 5 x 5 21 101 405 1
IVGFV 5 5 x 5 21 101 405 3
RHEIK 4 5 x 6 163 8073 40345 80
RHEIK 5 5 x 5 0 59 195 1
RHEIK 5 5 x 5 0 59 195 1
RHEIK 5 5 x 5 0 59 195 1
SFKNN 4 5 x 6 358 8100 40415 80

```
SFKNN 5 5 x 5 0 69 245 1
SFKNN 5 5 x 5 0 69 245 9
SFKNN 5 5 x 5 0 69 245 1
EKALK 4 5 x 6 3403 9500 47405 103
EKALK 5 5 x 5 151 98 390 1
EKALK 5 5 x 5 151 98 390 6
EKALK 5 5 x 5 151 98 390 1
YNSTG 4 5 x 6 786 9200 45920 94
YNSTG 5 5 x 5 39 101 405 1
YNSTG 5 5 x 5 39 101 405 2
YNSTG 5 5 x 5 39 101 405 1
YRSHA 4 5 x 6 383 8578 42870 86
YRSHA 5 5 x 5 3 97 385 1
YRSHA 5 5 x 5 3 97 385 2
YRSHA 5 5 x 5 3 97 385 1
DKIQN 4 5 x 6 692 9132 45575 92
DKIQN 5 5 x 5 15 101 405 2
DKIQN 5 5 x 5 15 101 405 1
DKIQN 5 5 x 5 15 101 405 2
LHCCG 4 5 x 6 84 6944 34755 69
LHCCG 5 5 x 5 4 95 375 1
LHCCG 5 5 x 5 4 95 375 1
LHCCG 5 5 x 5 4 95 375 1
TDYRD 4 5 x 6 362 8482 42360 84
TDYRD 5 5 x 5 8 101 405 2
TDYRD 5 5 x 5 8 101 405 1
TDYRD 5 5 x 5 8 101 405 2
TDTNY 4 5 x 6 414 9089 45375 89
TDTNY 5 5 x 5 3 109 445 2
TDTNY 5 5 x 5 3 109 445 1
TDTNY 5 5 x 5 3 109 445 2
SEKGF 4 5 x 6 1176 9648 48150 96
SEKGF 5 5 x 5 32 98 390 1
SEKGF 5 5 x 5 32 98 390 2
SEKGF 5 5 x 5 32 98 390 1
KSCCK 4 5 x 6 283 6183 30905 61
KSCCK 5 5 x 5 8 96 380 2
KSCCK 5 5 x 5 8 96 380 1
KSCCK 5 5 x 5 8 96 380 2
EDCTP 4 5 x 6 217 8438 42205 81
EDCTP 5 5 x 5 6 99 395 1
EDCTP 5 5 x 5 6 99 395 2
EDCTP 5 5 x 5 6 99 395 5
RDADK 4 5 x 6 621 8958 44700 88
RDADK 5 5 x 5 0 72 260 1
RDADK 5 5 x 5 0 72 260 1
RDADK 5 5 x 5 0 72 260 1
```

NNEGC 4 5 x 6 71 4289 21405 42
NNEGC 5 5 x 5 0 107 435 2
NNEGC 5 5 x 5 0 107 435 1
NNEGC 5 5 x 5 0 107 435 2
IKVMT 4 5 x 6 198 8336 41620 80
IKVMT 5 5 x 5 5 94 370 1
IKVMT 5 5 x 5 5 94 370 1
IKVMT 5 5 x 5 5 94 370 1
IESEM 4 5 x 6 677 8360 41725 84
IESEM 5 5 x 5 10 94 370 1
IESEM 5 5 x 5 10 94 370 1
IESEM 5 5 x 5 10 94 370 1
VVAGI 4 5 x 6 2402 6146 30630 67
VVAGI 5 5 x 5 108 159 695 2
VVAGI 5 5 x 5 108 159 695 3
VVAGI 5 5 x 5 108 159 695 2
FGVAC 4 5 x 6 836 9314 46480 92
FGVAC 5 5 x 5 5 100 400 2
FGVAC 5 5 x 5 5 100 400 5
FGVAC 5 5 x 5 5 100 400 1
QLIGI 4 5 x 6 1519 10008 49965 102
QLIGI 5 5 x 5 0 94 370 2
QLIGI 5 5 x 5 0 94 370 1
QLIGI 5 5 x 5 0 94 370 2
LAYCL 4 5 x 6 174 8212 41015 79
LAYCL 5 5 x 5 22 92 360 3
LAYCL 5 5 x 5 22 92 360 1
LAYCL 5 5 x 5 22 92 360 1
RAITN 4 5 x 6 561 9091 45370 87
RAITN 5 5 x 5 0 51 155 1
RAITN 5 5 x 5 0 51 155 1
RAITN 5 5 x 5 0 51 155 1
QYEIV 4 5 x 6 28 7909 39480 76
QYEIV 5 5 x 5 0 43 115 1
QYEIV 5 5 x 5 0 43 115 1
QYEIV 5 5 x 5 0 43 115 2
MAKNP 4 5 x 6 51 8464 42265 81
MAKNP 5 5 x 5 0 69 245 1
MAKNP 5 5 x 5 0 69 245 5
MAKNP 5 5 x 5 0 69 245 1
ENCED 4 5 x 6 113 6557 32755 64
ENCED 5 5 x 5 3 113 465 1
ENCED 5 5 x 5 3 113 465 6
ENCED 5 5 x 5 3 113 465 1
HILNA 4 5 x 6 360 8763 43755 86
HILNA 5 5 x 5 7 100 400 1
HILNA 5 5 x 5 7 100 400 2

HILNA 5 5 x 5 7 100 400 1
AFKSK 4 5 x 6 720 8257 41210 81
AFKSK 5 5 x 5 12 100 400 2
AFKSK 5 5 x 5 12 100 400 1
AFKSK 5 5 x 5 12 100 400 1
ICKSL 4 5 x 6 321 8156 40750 83
ICKSL 5 5 x 5 17 100 400 1
ICKSL 5 5 x 5 17 100 400 1
ICKSL 5 5 x 5 17 100 400 1
ICGLV 4 5 x 6 560 8420 42065 86
ICGLV 5 5 x 5 9 100 400 1
ICGLV 5 5 x 5 9 100 400 3
ICGLV 5 5 x 5 9 100 400 1
GILAL 4 5 x 6 2172 9115 45485 96
GILAL 5 5 x 5 0 61 205 2
GILAL 5 5 x 5 0 61 205 1
GILAL 5 5 x 5 0 61 205 2
LIVLF 4 5 x 6 1944 10145 50635 107
LIVLF 5 5 x 5 60 121 505 2
LIVLF 5 5 x 5 60 121 505 2
LIVLF 5 5 x 5 60 121 505 1
GSKHF 4 5 x 6 167 8552 42695 83
GSKHF 5 5 x 5 4 98 390 1
GSKHF 5 5 x 5 4 98 390 1
GSKHF 5 5 x 5 4 98 390 2
MASPSR 4 6 x 8 2888 990136 6155862 10860
MASPSR 5 6 x 7 74 11581 69630 122
MASPSR 6 6 x 6 2 112 552 1
MASPSR 6 6 x 6 2 112 552 1
LQTKPV 4 6 x 8 1794 1297462 8003748 14311
LQTKPV 5 6 x 7 38 12436 74892 131
LQTKPV 6 6 x 6 6 104 504 1
LQTKPV 6 6 x 6 6 104 504 2
TCFKSV 4 6 x 8 673 837631 5215386 9335
TCFKSV 5 6 x 7 2 8925 53838 98
TCFKSV 6 6 x 6 1 95 456 2
TCFKSV 6 6 x 6 1 95 456 1
LIYTFI 4 6 x 8 1138 1113732 6889446 12281
LIYTFI 5 6 x 7 21 11458 68928 121
LIYTFI 6 6 x 6 2 107 522 2
LIYTFI 6 6 x 6 2 107 522 1
WITGVI 4 6 x 8 2062 838505 5244606 9253
WITGVI 5 6 x 7 58 10421 62682 114
WITGVI 6 6 x 6 7 100 480 1
WITGVI 6 6 x 6 7 100 480 1
LAVGIW 4 6 x 8 4101 1457875 8960148 16116
LAVGIW 5 6 x 7 119 13228 79446 143

LAVGIW 6 6 x 6 3 117 582 1
LAVGIW 6 6 x 6 3 117 582 1
KVSLEN 4 6 x 8 1632 1339814 8251056 14830
KVSLEN 5 6 x 7 20 10020 60186 104
KVSLEN 6 6 x 6 0 49 174 0
KVSLEN 6 6 x 6 0 49 174 1
FSLLNE 4 6 x 8 4813 1270337 7851000 14085
FSLLNE 5 6 x 7 72 11628 69876 123
FSLLNE 6 6 x 6 3 117 582 1
FSLLNE 6 6 x 6 3 117 582 2
ATNVPF 4 6 x 8 1398 1176150 7265982 12927
ATNVPF 5 6 x 7 33 11606 69768 123
ATNVPF 6 6 x 6 3 110 540 1
ATNVPF 6 6 x 6 3 110 540 2
LIATGT 4 6 x 8 4508 1390192 8568768 15506
LIATGT 5 6 x 7 182 13063 78462 139
LIATGT 6 6 x 6 3 116 576 1
LIATGT 6 6 x 6 3 116 576 2
IILLGT 4 6 x 8 6508 866508 5352738 9780
IILLGT 5 6 x 7 154 7350 44142 78
IILLGT 6 6 x 6 6 178 948 2
IILLGT 6 6 x 6 6 178 948 2
GCFATC 4 6 x 8 519 753181 4678506 8408
GCFATC 5 6 x 7 9 7858 47322 85
GCFATC 6 6 x 6 4 103 498 1
GCFATC 6 6 x 6 4 103 498 1
ASAWML 4 6 x 8 479 711488 4383840 7739
ASAWML 5 6 x 7 5 9958 59880 103
ASAWML 6 6 x 6 1 158 828 1
ASAWML 6 6 x 6 1 158 828 1
LYAMFL 4 6 x 8 920 1052490 6509172 11664
LYAMFL 5 6 x 7 18 10469 63048 111
LYAMFL 6 6 x 6 4 101 486 1
LYAMFL 6 6 x 6 4 101 486 2
LVFLVE 4 6 x 8 3586 1078923 6659526 12079
LVFLVE 5 6 x 7 127 11171 67218 125
LVFLVE 6 6 x 6 3 136 696 2
LVFLVE 6 6 x 6 3 136 696 2
VAAIVG 4 6 x 8 5042 758472 4688496 8433
VAAIVG 5 6 x 7 239 12825 76998 136
VAAIVG 6 6 x 6 11 132 672 2
VAAIVG 6 6 x 6 11 132 672 1
VFRHEI 4 6 x 8 264 1022771 6322050 11277
VFRHEI 5 6 x 7 8 10117 60876 106
VFRHEI 6 6 x 6 4 93 438 1
VFRHEI 6 6 x 6 4 93 438 1
NSFKNN 4 6 x 8 1078 879423 5466402 9632

NSFKNN 5 6 x 7 18 9380 56400 101
NSFKNN 6 6 x 6 2 119 594 1
NSFKNN 6 6 x 6 2 119 594 2
EKALKQ 4 6 x 8 6647 1221862 7542816 13896
EKALKQ 5 6 x 7 238 12333 74064 130
EKALKQ 6 6 x 6 0 109 534 1
EKALKQ 6 6 x 6 0 109 534 1
NSTGDY 4 6 x 8 2028 1166815 7216890 12893
NSTGDY 5 6 x 7 42 11291 67884 119
NSTGDY 6 6 x 6 3 93 438 1
NSTGDY 6 6 x 6 3 93 438 1
SHAVDK 4 6 x 8 1436 1072502 6635214 11825
SHAVDK 5 6 x 7 14 10854 65310 115
SHAVDK 6 6 x 6 1 104 504 1
SHAVDK 6 6 x 6 1 104 504 1
QNTLHC 4 6 x 8 797 950859 5911284 10477
QNTLHC 5 6 x 7 9 10536 63336 111
QNTLHC 6 6 x 6 0 96 456 1
QNTLHC 6 6 x 6 0 96 456 1
GVTDYR 4 6 x 8 1686 1202550 7420620 13378
GVTDYR 5 6 x 7 36 12218 73470 127
GVTDYR 6 6 x 6 1 111 546 1
GVTDYR 6 6 x 6 1 111 546 1
WTDTNY 4 6 x 8 602 706865 4434042 7872
WTDTNY 5 6 x 7 4 9542 57468 100
WTDTNY 6 6 x 6 1 96 462 1
WTDTNY 6 6 x 6 1 96 462 1
SEKGFP 4 6 x 8 2122 1228909 7571610 13665
SEKGFP 5 6 x 7 64 11749 70710 127
SEKGFP 6 6 x 6 1 109 534 2
SEKGFP 6 6 x 6 1 109 534 1
SCCKLE 4 6 x 8 288 513555 3181482 5732
SCCKLE 5 6 x 7 4 8550 51426 91
SCCKLE 6 6 x 6 0 83 378 2
SCCKLE 6 6 x 6 0 83 378 1
CTPQRD 4 6 x 8 584 801119 4997658 9038
CTPQRD 5 6 x 7 5 8907 53604 97
CTPQRD 6 6 x 6 0 45 150 1
CTPQRD 6 6 x 6 0 45 150 1
DKVNNE 4 6 x 8 1395 1058249 6537534 11881
DKVNNE 5 6 x 7 10 10023 60276 110
DKVNNE 6 6 x 6 0 83 378 1
DKVNNE 6 6 x 6 0 83 378 1
CFIKVM 4 6 x 8 692 791195 4948116 8929
CFIKVM 5 6 x 7 20 8935 53910 98
CFIKVM 6 6 x 6 2 100 480 1
CFIKVM 6 6 x 6 2 100 480 2

IIESEM 4 6 x 8 1611 629852 3917766 7074
IIESEM 5 6 x 7 10 6294 37794 69
IIESEM 6 6 x 6 0 142 732 2
IIESEM 6 6 x 6 0 142 732 2
VVAGIS 4 6 x 8 5190 858552 5296962 9654
VVAGIS 5 6 x 7 162 8241 49482 88
VVAGIS 6 6 x 6 4 191 1026 2
VVAGIS 6 6 x 6 4 191 1026 2
GVACFQ 4 6 x 8 674 1168815 7194534 13046
GVACFQ 5 6 x 7 23 10771 64752 112
GVACFQ 6 6 x 6 3 105 510 1
GVACFQ 6 6 x 6 3 105 510 1
IGIFLA 4 6 x 8 2740 804813 4971402 8878
IGIFLA 5 6 x 7 92 12992 78054 136
IGIFLA 6 6 x 6 5 175 930 2
IGIFLA 6 6 x 6 5 175 930 2
CLSRAI 4 6 x 8 2148 1148671 7100070 12894
CLSRAI 5 6 x 7 103 11492 69096 123
CLSRAI 6 6 x 6 2 108 528 2
CLSRAI 6 6 x 6 2 108 528 1
NNQYEI 4 6 x 8 198 568453 3531852 6404
NNQYEI 5 6 x 7 0 5033 30180 54
NNQYEI 6 6 x 6 0 99 474 1
NNQYEI 6 6 x 6 0 99 474 1
MAKNPP 4 6 x 8 386 893334 5557062 9965
MAKNPP 5 6 x 7 1 9339 56064 102
MAKNPP 6 6 x 6 0 69 294 1
MAKNPP 6 6 x 6 0 69 294 1
NCEDCH 4 6 x 8 208 671497 4173996 7540
NCEDCH 5 6 x 7 0 6880 41472 73
NCEDCH 6 6 x 6 0 43 138 1
NCEDCH 6 6 x 6 0 43 138 0
LNAEAF 4 6 x 8 2836 1255403 7750926 14008
LNAEAF 5 6 x 7 41 12448 74874 130
LNAEAF 6 6 x 6 0 112 552 2
LNAEAF 6 6 x 6 0 112 552 1
SKKICK 4 6 x 8 1536 609735 3772500 6838
SKKICK 5 6 x 7 26 10301 61872 111
SKKICK 6 6 x 6 1 104 504 1
SKKICK 6 6 x 6 1 104 504 1
LKICGL 4 6 x 8 1222 1191374 7340778 13331
LKICGL 5 6 x 7 71 11187 67302 119
LKICGL 6 6 x 6 1 109 534 2
LKICGL 6 6 x 6 1 109 534 1
FGILAL 4 6 x 8 4271 1154385 7154238 12906
FGILAL 5 6 x 7 64 11616 69774 123
FGILAL 6 6 x 6 3 114 564 2

FGILAL 6 6 x 6 3 114 564 1
LIVLFW 4 6 x 8 3085 1191880 7366158 13356
LIVLFW 5 6 x 7 88 12334 74148 130
LIVLFW 6 6 x 6 0 156 816 2
LIVLFW 6 6 x 6 0 156 816 2
SKHFWP 4 6 x 8 167 858190 5309496 9501
SKHFWP 5 6 x 7 5 8497 51150 93
SKHFWP 6 6 x 6 0 73 318 1
SKHFWP 6 6 x 6 0 73 318 1
VPKKAY 4 6 x 8 2427 1009485 6248052 11213
VPKKAY 5 6 x 7 20 9640 57942 102
VPKKAY 6 6 x 6 0 85 390 1
VPKKAY 6 6 x 6 0 85 390 1
MEHTFY 4 6 x 8 594 719195 4512468 8031
MEHTFY 5 6 x 7 4 8502 51186 90
MEHTFY 6 6 x 6 0 93 438 1
MEHTFY 6 6 x 6 0 93 438 1
NGEKKK 4 6 x 8 3969 928773 5741436 10283
NGEKKK 5 6 x 7 118 9336 56124 102
NGEKKK 6 6 x 6 3 114 564 1
NGEKKK 6 6 x 6 3 114 564 1
YMEIDP 4 6 x 8 986 859956 5369850 9625
YMEIDP 5 6 x 7 15 10614 63978 118
YMEIDP 6 6 x 6 2 100 480 1
YMEIDP 6 6 x 6 2 100 480 1
TRTEIF 4 6 x 8 1662 686931 4259616 7741
TRTEIF 5 6 x 7 37 12496 75174 135
TRTEIF 6 6 x 6 1 171 906 2
TRTEIF 6 6 x 6 1 171 906 2
SGNGTD 4 6 x 8 2760 1096069 6777702 12188
SGNGTD 5 6 x 7 66 12210 73362 128
SGNGTD 6 6 x 6 0 114 564 1
SGNGTD 6 6 x 6 0 114 564 2
MASPSRR 4 7 x 10 5550 5920812 63416430 89384
MASPSRR 5 7 x 9 240 1011432 7379001 12574
MASPSRR 6 7 x 8 5 12577 88459 145
MASPSRR 7 7 x 7 2 112 651 2
QTKPVIT 4 7 x 10 3144 5614421 62437074 87051
QTKPVIT 5 7 x 9 93 1115693 8104341 13768
QTKPVIT 6 7 x 8 9 12962 91259 147
QTKPVIT 7 7 x 7 2 112 644 2
FKSVLLI 4 7 x 10 8462 6379387 71792805 100147
FKSVLLI 5 7 x 9 295 1285266 9307501 15713
FKSVLLI 6 7 x 8 10 13537 95130 158
FKSVLLI 7 7 x 7 1 112 644 2
TFIFWIT 4 7 x 10 881 4278128 44425458 62208
TFIFWIT 5 7 x 9 24 797730 5823797 9731

TFIFWIT 6 7 x 8 7 10502 73997 125
TFIFWIT 7 7 x 7 2 105 595 1
VILLAVG 4 7 x 10 12084 6158027 66628100 93007
VILLAVG 5 7 x 9 676 1466040 10601822 18074
VILLAVG 6 7 x 8 50 14396 101276 166
VILLAVG 7 7 x 7 5 143 861 2
WGKVSLE 4 7 x 10 3978 5509289 62717781 85412
WGKVSLE 5 7 x 9 114 1098839 7999614 13415
WGKVSLE 6 7 x 8 9 13128 92379 151
WGKVSLE 7 7 x 7 2 106 609 1
YFSLLNE 4 7 x 10 7590 6678237 73554789 101135
YFSLLNE 5 7 x 9 161 1118185 8151437 13519
YFSLLNE 6 7 x 8 6 13393 94332 157
YFSLLNE 7 7 x 7 2 115 665 1
ATNVPFV 4 7 x 10 2779 6585557 71652623 99206
ATNVPFV 5 7 x 9 81 1190297 8623447 14476
ATNVPFV 6 7 x 8 5 12556 88340 145
ATNVPFV 7 7 x 7 2 112 644 1
IATGTVI 4 7 x 10 7964 7036351 76090175 105109
IATGTVI 5 7 x 9 309 1245471 9040493 15177
IATGTVI 6 7 x 8 12 14133 99428 165
IATGTVI 7 7 x 7 2 121 707 2
LLGTFGC 4 7 x 10 5404 6095267 63392378 88816
LLGTFGC 5 7 x 9 176 878089 6354306 10735
LLGTFGC 6 7 x 8 7 8964 62993 108
LLGTFGC 7 7 x 7 2 186 1169 2
ATCRASA 4 7 x 10 2571 5346182 57519175 79539
ATCRASA 5 7 x 9 62 880407 6417369 10675
ATCRASA 6 7 x 8 3 11590 81676 133
ATCRASA 7 7 x 7 1 146 882 2
MLKLYAM 4 7 x 10 2255 5026772 53033904 74071
MLKLYAM 5 7 x 9 67 1056127 7684621 12853
MLKLYAM 6 7 x 8 5 12911 90811 150
MLKLYAM 7 7 x 7 0 73 371 1
LTLVFLV 4 7 x 10 7814 6022650 62295947 87923
LTLVFLV 5 7 x 9 339 834126 6042358 10173
LTLVFLV 6 7 x 8 15 15285 107625 180
LTLVFLV 7 7 x 7 5 192 1204 2
LVAAIVG 4 7 x 10 12600 6540110 69770582 97070
LVAAIVG 5 7 x 9 688 1474137 10645068 18143
LVAAIVG 6 7 x 8 29 15190 106750 175
LVAAIVG 7 7 x 7 2 123 721 1
VFRHEIK 4 7 x 10 678 4833458 55132378 74558
VFRHEIK 5 7 x 9 21 1149060 8313424 13962
VFRHEIK 6 7 x 8 4 11735 82439 136
VFRHEIK 7 7 x 7 4 93 518 1
SFKNNYE 4 7 x 10 1519 5835316 64137339 88127

SFKNNYE 5 7 x 9 50 1099341 7959294 13317
SFKNNYE 6 7 x 8 1 11276 79289 129
SFKNNYE 7 7 x 7 0 69 343 1
ALKQYNS 4 7 x 10 3284 7516527 82862626 113314
ALKQYNS 5 7 x 9 105 1543686 11097401 19783
ALKQYNS 6 7 x 8 2 14492 101934 168
ALKQYNS 7 7 x 7 1 113 658 2
GDYRSHA 4 7 x 10 1336 5541911 62510322 84804
GDYRSHA 5 7 x 9 13 1131964 8200773 13770
GDYRSHA 6 7 x 8 0 11689 82089 137
GDYRSHA 7 7 x 7 0 65 315 2
DKIQNTL 4 7 x 10 2826 5790124 66252676 89817
DKIQNTL 5 7 x 9 87 1321694 9546656 16073
DKIQNTL 6 7 x 8 5 12926 91007 148
DKIQNTL 7 7 x 7 2 111 644 2
CCGVTDY 4 7 x 10 1761 3855465 41560379 57380
CCGVTDY 5 7 x 9 32 535441 3911397 6653
CCGVTDY 6 7 x 8 4 5212 36757 60
CCGVTDY 7 7 x 7 2 162 1001 2
DWTDTNY 4 7 x 10 734 3317188 35304710 48697
DWTDTNY 5 7 x 9 7 708996 5203695 8499
DWTDTNY 6 7 x 8 1 9378 65975 109
DWTDTNY 7 7 x 7 0 104 588 2
SEKGFPK 4 7 x 10 3495 6975755 75387207 104331
SEKGFPK 5 7 x 9 130 1244867 8992305 15081
SEKGFPK 6 7 x 8 5 12666 89229 148
SEKGFPK 7 7 x 7 1 109 630 2
CCKLEDC 4 7 x 10 2368 4675467 49050288 68776
CCKLEDC 5 7 x 9 48 499674 3640049 5996
CCKLEDC 6 7 x 8 2 5023 35371 62
CCKLEDC 7 7 x 7 1 145 889 1
PQRDADK 4 7 x 10 2143 5267591 59032211 80713
PQRDADK 5 7 x 9 45 1048635 7620277 12588
PQRDADK 6 7 x 8 2 11765 82698 135
PQRDADK 7 7 x 7 2 103 588 1
NNEGCFI 4 7 x 10 671 4710920 50331288 70116
NNEGCFI 5 7 x 9 9 674300 4885251 8256
NNEGCFI 6 7 x 8 1 5782 40670 136
NNEGCFI 7 7 x 7 0 109 630 2
VMTIIES 4 7 x 10 2550 5673793 63132076 86479
VMTIIES 5 7 x 9 82 1068884 7769307 13041
VMTIIES 6 7 x 8 4 12096 85148 141
VMTIIES 7 7 x 7 1 105 595 2
MGVVAGI 4 7 x 10 6767 4229573 46158952 63557
MGVVAGI 5 7 x 9 448 1031686 7530439 12755
MGVVAGI 6 7 x 8 21 13311 93667 160
MGVVAGI 7 7 x 7 2 108 623 1

FGVACFQ 4 7 x 10 2262 6626654 70891996 98314
FGVACFQ 5 7 x 9 52 1086880 7874083 13320
FGVACFQ 6 7 x 8 6 11756 82670 139
FGVACFQ 7 7 x 7 2 125 735 2
IGIFLAY 4 7 x 10 4177 4564644 51160837 69958
IGIFLAY 5 7 x 9 163 875570 6339634 10660
IGIFLAY 6 7 x 8 16 14936 105084 174
IGIFLAY 7 7 x 7 2 189 1183 2
LSRAITN 4 7 x 10 5057 8159621 90720238 124548
LSRAITN 5 7 x 9 177 1611594 11597537 19644
LSRAITN 6 7 x 8 7 14238 100023 162
LSRAITN 7 7 x 7 2 120 700 2
MAKNPPE 4 7 x 10 1162 5559409 60937331 83659
MAKNPPE 5 7 x 9 16 1027974 7476707 12633
MAKNPPE 6 7 x 8 0 10883 76314 125
MAKNPPE 7 7 x 7 0 69 343 1
CEDCHIL 4 7 x 10 420 3504886 38389554 52805
CEDCHIL 5 7 x 9 14 840045 6096979 10382
CEDCHIL 6 7 x 8 0 9043 63700 109
CEDCHIL 7 7 x 7 0 87 469 2
AEAFKSK 4 7 x 10 4077 4832136 52124023 72584
AEAFKSK 5 7 x 9 154 803351 5820472 9841
AEAFKSK 6 7 x 8 3 13516 95158 159
AEAFKSK 7 7 x 7 1 183 1141 3
ICKSLKI 4 7 x 10 3276 5469366 59421362 82399
ICKSLKI 5 7 x 9 154 890130 6486704 10811
ICKSLKI 6 7 x 8 0 10270 72359 119
ICKSLKI 7 7 x 7 0 108 616 1
GLVFGIL 4 7 x 10 6687 7386380 79207177 110411
GLVFGIL 5 7 x 9 333 1364165 9863154 16724
GLVFGIL 6 7 x 8 14 14889 104713 180
GLVFGIL 7 7 x 7 0 135 805 2
LTLIVLF 4 7 x 10 8039 5965082 63868868 89717
LTLIVLF 5 7 x 9 342 960819 6961381 11769
LTLIVLF 6 7 x 8 18 16958 119357 197
LTLIVLF 7 7 x 7 1 214 1358 3
GSKHFWP 4 7 x 10 619 5823829 62968899 87216
GSKHFWP 5 7 x 9 13 1164257 8377460 14194
GSKHFWP 6 7 x 8 0 10582 74340 122
GSKHFWP 7 7 x 7 0 101 567 1
VPKKAYD 4 7 x 10 3337 4648157 50804208 69966
VPKKAYD 5 7 x 9 52 1138242 8236081 14044
VPKKAYD 6 7 x 8 0 11078 77910 127
VPKKAYD 7 7 x 7 0 85 455 1
EHTFYSN 4 7 x 10 1410 4994558 55853602 76480
EHTFYSN 5 7 x 9 39 1003263 7291193 12220
EHTFYSN 6 7 x 8 0 9635 67725 115

EHTFYSN 7 7 x 7 0 55 245 2
EKKKIYM 4 7 x 10 5329 3266742 34207180 48237
EKKKIYM 5 7 x 9 200 678588 4914623 8330
EKKKIYM 6 7 x 8 10 12913 90769 152
EKKKIYM 7 7 x 7 0 111 637 2
IDPVTRT 4 7 x 10 3319 6108132 67351466 93141
IDPVTRT 5 7 x 9 127 1149738 8345232 14031
IDPVTRT 6 7 x 8 12 13481 94983 158
IDPVTRT 7 7 x 7 0 113 651 1
IFRSGNG 4 7 x 10 3497 6028531 66716986 91408
IFRSGNG 5 7 x 9 95 1121997 8151934 13675
IFRSGNG 6 7 x 8 1 12422 87528 147
IFRSGNG 7 7 x 7 0 97 546 2
DETLEVH 4 7 x 10 5747 7513302 81634896 113375
DETLEVH 5 7 x 9 122 1228450 8908298 15085
DETLEVH 6 7 x 8 3 12915 90734 153
DETLEVH 7 7 x 7 0 54 238 1
FKNGYTG 4 7 x 10 2784 5711025 62459873 86445
FKNGYTG 5 7 x 9 57 1020804 7427847 12427
FKNGYTG 6 7 x 8 1 12041 84826 140
FKNGYTG 7 7 x 7 0 107 616 2
YFVGLQK 4 7 x 10 2622 5580170 63655221 86806
YFVGLQK 5 7 x 9 58 1157630 8414301 14165
YFVGLQK 6 7 x 8 0 11977 84112 139
YFVGLQK 7 7 x 7 0 100 567 1
FIKTQIK 4 7 x 10 2147 5883791 63326403 87905
FIKTQIK 5 7 x 9 64 982754 7130347 11966
FIKTQIK 6 7 x 8 3 10932 77070 127
FIKTQIK 7 7 x 7 0 106 609 1
IPEFSEP 4 7 x 10 3263 5755101 62604262 86757
IPEFSEP 5 7 x 9 64 1052091 7648368 12971
IPEFSEP 6 7 x 8 2 12033 84665 147
IPEFSEP 7 7 x 7 0 82 434 1
EEIDENE 4 7 x 10 5300 4720308 49843906 69736
EEIDENE 5 7 x 9 272 682251 4980521 8427
EEIDENE 6 7 x 8 9 7817 55104 93
EEIDENE 7 7 x 7 0 212 1351 2
ITTTFFE 4 7 x 10 2796 3192247 33492781 47112
ITTTFFE 5 7 x 9 51 644769 4675881 7862
ITTTFFE 6 7 x 8 0 12321 86730 143
ITTTFFE 7 7 x 7 0 99 553 2
MASPSRRL 4 8 x 12 8587 6609273 152011552 174378
MASPSRRL 5 8 x 11 481 6000150 76353792 102407
MASPSRRL 6 8 x 10 10 1189725 9924792 15809
MASPSRRL 7 8 x 9 4 14274 114872 177
TKPVITCF 4 8 x 12 4590 8717672 199326888 226678
TKPVITCF 5 8 x 11 132 6240751 78735208 103650

TKPVITCF 6 8 x 10 5 1189583 9886616 15462
TKPVITCF 7 8 x 9 4 14166 113944 178
SVLLIYTF 4 8 x 12 11839 8920507 204943496 230077
SVLLIYTF 5 8 x 11 258 6425626 80745672 106138
SVLLIYTF 6 8 x 10 12 1644404 13570984 21566
SVLLIYTF 7 8 x 9 2 15211 122336 189
FWITGVIL 4 8 x 12 7910 7180762 167553272 187877
FWITGVIL 5 8 x 11 334 4720223 62547576 80192
FWITGVIL 6 8 x 10 30 995782 8360512 13057
FWITGVIL 7 8 x 9 12 12227 98568 161
AVGIWGKV 4 8 x 12 5377 8360213 189326888 213189
AVGIWGKV 5 8 x 11 188 7176464 88594560 115720
AVGIWGKV 6 8 x 10 21 1266721 10488736 16516
AVGIWGKV 7 8 x 9 4 13244 106672 164
LENYFSLL 4 8 x 12 7732 8570960 197115288 221885
LENYFSLL 5 8 x 11 196 6628766 83552984 109322
LENYFSLL 6 8 x 10 11 1364936 11327888 17900
LENYFSLL 7 8 x 9 4 14447 116280 177
EKATNVPF 4 8 x 12 5680 9064778 212184328 238093
EKATNVPF 5 8 x 11 236 6993349 90705752 117991
EKATNVPF 6 8 x 10 8 1510670 12483160 19800
EKATNVPF 7 8 x 9 4 14815 119224 183
LIATGTVI 4 8 x 12 14727 10958402 251024912 283049
LIATGTVI 5 8 x 11 720 8096102 102210144 133499
LIATGTVI 6 8 x 10 23 1609278 13328312 20885
LIATGTVI 7 8 x 9 3 16738 134712 208
LLGTFGCF 4 8 x 12 7393 8096922 180040560 203020
LLGTFGCF 5 8 x 11 285 6099226 72599832 96646
LLGTFGCF 6 8 x 10 12 882068 7317400 11500
LLGTFGCF 7 8 x 9 4 9355 75224 117
TCRASAWM 4 8 x 12 3637 8055269 181869952 203465
TCRASAWM 5 8 x 11 116 5357295 66787608 87466
TCRASAWM 6 8 x 10 3 876404 7349144 11493
TCRASAWM 7 8 x 9 1 10293 82872 127
KLYAMFLT 4 8 x 12 2985 8872212 202749424 227891
KLYAMFLT 5 8 x 11 123 7273601 90578408 118838
KLYAMFLT 6 8 x 10 10 1314495 10896904 17138
KLYAMFLT 7 8 x 9 6 13650 109624 165
VFLVELVA 4 8 x 12 13319 8141370 184123800 207570
VFLVELVA 5 8 x 11 556 5621432 69380760 91412
VFLVELVA 6 8 x 10 53 1306561 10851856 17245
VFLVELVA 7 8 x 9 3 15061 121400 188
IVGFVFRH 4 8 x 12 4382 6959858 157440424 177029
IVGFVFRH 5 8 x 11 128 6238195 77624376 101828
IVGFVFRH 6 8 x 10 14 1191975 9913016 15640
IVGFVFRH 7 8 x 9 4 14770 118856 191
IKNSFKNN 4 8 x 12 5349 8213382 186902608 210008

IKNSFKNN 5 8 x 11 195 6322455 77887056 102381
IKNSFKNN 6 8 x 10 14 1061248 8840944 13897
IKNSFKNN 7 8 x 9 2 12009 96928 150
EKALKQYN 4 8 x 12 13456 9010528 205222680 230475
EKALKQYN 5 8 x 11 670 8146031 102245968 134168
EKALKQYN 6 8 x 10 9 1432314 11860240 18780
EKALKQYN 7 8 x 9 1 15765 126752 195
TGDYRSHA 4 8 x 12 2783 7911428 185410272 207235
TGDYRSHA 5 8 x 11 71 5780173 76046176 98107
TGDYRSHA 6 8 x 10 2 1378844 11407144 17891
TGDYRSHA 7 8 x 9 1 14197 114032 176
DKIQNTLH 4 8 x 12 4246 7699103 182549576 204866
DKIQNTLH 5 8 x 11 128 5800376 76736224 99013
DKIQNTLH 6 8 x 10 12 1386515 11494848 18156
DKIQNTLH 7 8 x 9 3 14224 114592 181
CGVTDYRD 4 8 x 12 3770 7689235 175394880 197515
CGVTDYRD 5 8 x 11 119 5477386 69166632 90362
CGVTDYRD 6 8 x 10 6 1014172 8471008 13135
CGVTDYRD 7 8 x 9 3 12080 97400 156
TDTNYYSE 4 8 x 12 1948 5693716 132544232 147679
TDTNYYSE 5 8 x 11 31 4284084 54504552 70914
TDTNYYSE 6 8 x 10 3 757684 6293528 9665
TDTNYYSE 7 8 x 9 2 13249 106600 163
GFPKSCCK 4 8 x 12 1988 8561404 192567528 216554
GFPKSCCK 5 8 x 11 71 5797430 71851208 94254
GFPKSCCK 6 8 x 10 9 1016408 8428872 13171
GFPKSCCK 7 8 x 9 3 10669 85664 133
EDCTPQRD 4 8 x 12 1619 7120572 163195952 183611
EDCTPQRD 5 8 x 11 27 5093444 64395880 83985
EDCTPQRD 6 8 x 10 4 918028 7662192 12041
EDCTPQRD 7 8 x 9 2 12125 97616 151
DKVNNEGC 4 8 x 12 2956 6620985 153457856 172165
DKVNNEGC 5 8 x 11 106 6031202 77347328 100749
DKVNNEGC 6 8 x 10 6 1252096 10380480 16494
DKVNNEGC 7 8 x 9 1 12779 102608 158
IKVMTIIE 4 8 x 12 3073 8010578 183887880 207567
IKVMTIIE 5 8 x 11 102 5657203 71743616 93875
IKVMTIIE 6 8 x 10 5 1199830 9958552 15774
IKVMTIIE 7 8 x 9 1 12269 98616 150
EMGVVAGI 4 8 x 12 9738 6620964 151417336 170051
EMGVVAGI 5 8 x 11 645 5974716 75807456 99402
EMGVVAGI 6 8 x 10 30 1116656 9328840 14681
EMGVVAGI 7 8 x 9 6 13947 112352 173
FGVACFQL 4 8 x 12 3018 7843521 180423248 201410
FGVACFQL 5 8 x 11 83 6714106 85098800 110549
FGVACFQL 6 8 x 10 11 1262447 10456680 16462
FGVACFQL 7 8 x 9 4 13572 109104 166

GIFLAYCL 4 8 x 12 6919 10406636 236248048 266080
GIFLAYCL 5 8 x 11 347 7328265 91937464 119946
GIFLAYCL 6 8 x 10 23 1261954 10485944 16317
GIFLAYCL 7 8 x 9 6 14092 113544 175
RAITNNQY 4 8 x 12 3927 8727897 200563856 224322
RAITNNQY 5 8 x 11 100 6627669 83660160 108568
RAITNNQY 6 8 x 10 4 1281739 10635872 16568
RAITNNQY 7 8 x 9 2 13127 105632 169
MAKNPPEN 4 8 x 12 2444 7621471 174284208 194555
MAKNPPEN 5 8 x 11 90 5561707 69879328 90738
MAKNPPEN 6 8 x 10 0 1039859 8683256 13499
MAKNPPEN 7 8 x 9 0 11726 94040 148
EDCHILNA 4 8 x 12 2003 6318919 149450376 166383
EDCHILNA 5 8 x 11 37 4139880 55476032 70817
EDCHILNA 6 8 x 10 0 1045736 8703624 13595
EDCHILNA 7 8 x 9 0 12540 100896 156
AFKSKKIC 4 8 x 12 7044 7322185 165653224 184953
AFKSKKIC 5 8 x 11 248 6595788 81625288 106462
AFKSKKIC 6 8 x 10 10 1132891 9445032 14883
AFKSKKIC 7 8 x 9 0 12776 102856 162
SLKICGLV 4 8 x 12 4614 9156011 211943344 237607
SLKICGLV 5 8 x 11 179 7958412 101159096 131459
SLKICGLV 6 8 x 10 4 1594978 13140632 20641
SLKICGLV 7 8 x 9 0 14825 119104 183
GILALTLI 4 8 x 12 22973 9170085 206913368 232286
GILALTLI 5 8 x 11 1481 8245284 102066224 133581
GILALTLI 6 8 x 10 119 1487742 12368816 19345
GILALTLI 7 8 x 9 0 14919 119920 186
LFWGSKHF 4 8 x 12 1586 8797327 199108072 223204
LFWGSKHF 5 8 x 11 47 5994684 74937816 97574
LFWGSKHF 6 8 x 10 0 1068735 8906840 13940
LFWGSKHF 7 8 x 9 0 13121 105488 162
PEVPKKAY 4 8 x 12 7081 7054507 160996936 180272
PEVPKKAY 5 8 x 11 172 4786076 60094648 78267
PEVPKKAY 6 8 x 10 6 1199904 9979896 15684
PEVPKKAY 7 8 x 9 0 14570 117272 179
MEHTFYSN 4 8 x 12 2007 6305374 147293488 163855
MEHTFYSN 5 8 x 11 71 4600205 59783312 76867
MEHTFYSN 6 8 x 10 0 896337 7540392 11639
MEHTFYSN 7 8 x 9 0 11173 89776 138
EKKKIYME 4 8 x 12 7326 5429228 121101376 136221
EKKKIYME 5 8 x 11 229 3270341 39220096 52103
EKKKIYME 6 8 x 10 11 684574 5690816 8899
EKKKIYME 7 8 x 9 0 13978 112552 173
DPVTRTEI 4 8 x 12 5613 8299260 193097232 215905
DPVTRTEI 5 8 x 11 236 6127024 79494080 102567
DPVTRTEI 6 8 x 10 6 1289090 10702304 16703

DPVTRTEI 7 8 x 9 0 13871 111648 170
RSGNGTDE 4 8 x 12 6863 7027562 163060344 180398
RSGNGTDE 5 8 x 11 276 6371266 82390144 108052
RSGNGTDE 6 8 x 10 4 1401680 11616968 18231
RSGNGTDE 7 8 x 9 0 15616 125696 192
LEVHDFKN 4 8 x 12 2858 9147707 211916736 237038
LEVHDFKN 5 8 x 11 60 6841587 87752264 113715
LEVHDFKN 6 8 x 10 1 1616535 13308616 21043
LEVHDFKN 7 8 x 9 0 14897 119808 186
YTGIYFVG 4 8 x 12 3865 6570377 149956880 167229
YTGIYFVG 5 8 x 11 104 5910730 74667688 96802
YTGIYFVG 6 8 x 10 2 1112676 9275680 14482
YTGIYFVG 7 8 x 9 0 14388 115816 180
QKCFIKTQ 4 8 x 12 1763 6362585 145906720 163036
QKCFIKTQ 5 8 x 11 28 4149584 52944856 68505
QKCFIKTQ 6 8 x 10 0 774003 6497464 9911
QKCFIKTQ 7 8 x 9 0 11345 91200 140
KVIPEFSE 4 8 x 12 6667 8861314 205382160 230398
KVIPEFSE 5 8 x 11 222 6321111 81921632 105874
KVIPEFSE 6 8 x 10 4 1384700 11477656 18082
KVIPEFSE 7 8 x 9 0 13260 106592 161
EEEIDENE 4 8 x 12 11429 5253025 117438016 131957
EEEIDENE 5 8 x 11 526 3303534 39701688 52490
EEEIDENE 6 8 x 10 32 349013 2922120 4439
EEEIDENE 7 8 x 9 3 9264 74808 117
ITTTFFEQ 4 8 x 12 3854 4780342 108495304 121255
ITTTFFEQ 5 8 x 11 66 3202817 39192752 51330
ITTTFFEQ 6 8 x 10 3 715479 5953192 9285
ITTTFFEQ 7 8 x 9 0 14061 113216 177
VIWVPAEK 4 8 x 12 3608 5564872 129263664 143821
VIWVPAEK 5 8 x 11 89 3710355 48058936 62147
VIWVPAEK 6 8 x 10 0 1040216 8697048 13574
VIWVPAEK 7 8 x 9 0 14518 116816 181
IENRDFLK 4 8 x 12 6552 8196585 194956056 216984
IENRDFLK 5 8 x 11 331 6077060 81362040 103658
IENRDFLK 6 8 x 10 8 1464375 12128768 19134
IENRDFLK 7 8 x 9 1 14754 118856 183
SKILEICD 4 8 x 12 10082 10648381 244767168 275135
SKILEICD 5 8 x 11 403 7938837 100555432 130996
SKILEICD 6 8 x 10 8 1419077 11741216 18637
SKILEICD 7 8 x 9 0 14675 118032 182
VTMYWINP 4 8 x 12 532 5819424 135784392 152847
VTMYWINP 5 8 x 11 18 4940743 63849440 82112
VTMYWINP 6 8 x 10 0 1083608 9008432 14147
VTMYWINP 7 8 x 9 0 12299 99040 153
LISVSELQ 4 8 x 12 11987 8914036 203726728 228462
LISVSELQ 5 8 x 11 614 8062419 101450200 132384

LISVSELQ 6 8 x 10 19 1571838 13038920 20648
LISVSELQ 7 8 x 9 0 17278 139040 214
MASPSRRLQ 4 9 x 14 13605 8499696 322260966 337727
MASPSRRLQ 5 9 x 13 656 6620523 173233602 194032
MASPSRRLQ 6 9 x 12 15 6021494 87900804 112349
MASPSRRLQ 7 9 x 11 6 1295459 12194379 18442
KPVITCFKS 4 9 x 14 4424 9530174 361139805 384311
KPVITCFKS 5 9 x 13 132 8733991 228522582 253698
KPVITCFKS 6 9 x 12 4 6311238 92075211 122098
KPVITCFKS 7 9 x 11 2 1329794 12414105 19718
LLIYTFIFW 4 9 x 14 6186 9264040 344479140 376089
LLIYTFIFW 5 9 x 13 183 7640763 193986693 223359
LLIYTFIFW 6 9 x 12 12 5251442 72454032 95260
LLIYTFIFW 7 9 x 11 7 924719 8673948 13156
TGVILLAVG 4 9 x 14 32131 11108791 417397716 434892
TGVILLAVG 5 9 x 13 2716 10639612 276993117 302487
TGVILLAVG 6 9 x 12 212 6864967 100926261 126192
TGVILLAVG 7 9 x 11 26 1496261 14035230 20904
WGKVSLENY 4 9 x 14 14653 11779651 447165126 458625
WGKVSLENY 5 9 x 13 514 8186942 217779102 235797
WGKVSLENY 6 9 x 12 19 5545052 83975679 103477
WGKVSLENY 7 9 x 11 6 1255126 11828583 17637
SLLNEKATN 4 9 x 14 18110 10633981 404905644 417034
SLLNEKATN 5 9 x 13 1032 8479101 222949044 243309
SLLNEKATN 6 9 x 12 31 6011217 87871905 109589
SLLNEKATN 7 9 x 11 4 1243776 11577960 17328
PFVLIATGT 4 9 x 14 19578 12338808 470363454 482777
PFVLIATGT 5 9 x 13 945 9569253 254606580 276181
PFVLIATGT 6 9 x 12 27 7078391 105567813 131145
PFVLIATGT 7 9 x 11 5 1449110 13576068 20359
IILLGTFGC 4 9 x 14 18483 9078007 335449332 345717
IILLGTFGC 5 9 x 13 738 6912719 174446892 192225
IILLGTFGC 6 9 x 12 37 4380585 60256476 76788
IILLGTFGC 7 9 x 11 6 1035363 9717318 14460
ATCRASAWM 4 9 x 14 5140 8401965 312165342 320662
ATCRASAWM 5 9 x 13 164 5964451 152372898 166543
ATCRASAWM 6 9 x 12 7 5360754 75480156 95037
ATCRASAWM 7 9 x 11 3 960569 9072531 13447
KLYAMFLTL 4 9 x 14 8096 10537559 395978949 407114
KLYAMFLTL 5 9 x 13 302 8875315 228760164 249465
KLYAMFLTL 6 9 x 12 37 7279325 102469977 129317
KLYAMFLTL 7 9 x 11 8 1339953 12556899 18843
FLVELVAAI 4 9 x 14 24112 8504159 319218273 328902
FLVELVAAI 5 9 x 13 1503 8161137 211464936 232763
FLVELVAAI 6 9 x 12 102 7400095 105980760 134134
FLVELVAAI 7 9 x 11 12 1489188 13942692 21108
GFVFRHEIK 4 9 x 14 2140 7714153 291953034 298261

GFVFRHEIK 5 9 x 13 59 6726894 175966101 191036
GFVFRHEIK 6 9 x 12 12 4667956 68323842 85069
GFVFRHEIK 7 9 x 11 8 1341237 12525921 18686
SFKNNYEKA 4 9 x 14 4846 7736901 295404453 302804
SFKNNYEKA 5 9 x 13 201 6441625 169439517 184655
SFKNNYEKA 6 9 x 12 5 5886871 86055732 107218
SFKNNYEKA 7 9 x 11 2 1267868 11873628 17687
KQYNSTGDY 4 9 x 14 6429 9558755 366421941 375423
KQYNSTGDY 5 9 x 13 225 7417275 197671140 213733
KQYNSTGDY 6 9 x 12 12 4885309 74310093 91743
KQYNSTGDY 7 9 x 11 4 1187541 11164257 16401
SHAVDKIQN 4 9 x 14 8110 10689298 408371022 418380
SHAVDKIQN 5 9 x 13 200 8514784 225567036 244010
SHAVDKIQN 6 9 x 12 16 6299317 93511197 116126
SHAVDKIQN 7 9 x 11 3 1395304 13057038 19369
LHCCGVTDY 4 9 x 14 3339 8469787 315223974 323806
LHCCGVTDY 5 9 x 13 67 6157488 158088537 172144
LHCCGVTDY 6 9 x 12 8 3711479 53411094 66883
LHCCGVTDY 7 9 x 11 5 1033521 9728937 14484
DWTDTNYYS 4 9 x 14 1823 5274230 198105993 202049
DWTDTNYYS 5 9 x 13 16 3697373 95033853 103649
DWTDTNYYS 6 9 x 12 4 3341372 47241360 59410
DWTDTNYYS 7 9 x 11 3 827717 7843284 11421
KGFPKSCCK 4 9 x 14 3366 8951382 331020171 338746
KGFPKSCCK 5 9 x 13 149 6469402 164505069 179307
KGFPKSCCK 6 9 x 12 18 5800750 81141912 101837
KGFPKSCCK 7 9 x 11 8 1057719 9908262 14879
EDCTPQRDA 4 9 x 14 3071 8693267 330833925 339043
EDCTPQRDA 5 9 x 13 115 7136024 187498386 202688
EDCTPQRDA 6 9 x 12 5 5140025 75357603 92906
EDCTPQRDA 7 9 x 11 4 1036018 9739620 14518
KVNNEGCFI 4 9 x 14 3432 9364901 352385271 361009
KVNNEGCFI 5 9 x 13 77 6952499 181988829 197919
KVNNEGCFI 6 9 x 12 3 4608468 67592106 84296
KVNNEGCFI 7 9 x 11 2 1332329 12429495 18674
VMTIIESEM 4 9 x 14 6469 8514736 319450050 328233
VMTIIESEM 5 9 x 13 190 6264171 162711810 177071
VMTIIESEM 6 9 x 12 10 5679263 81746361 102367
VMTIIESEM 7 9 x 11 4 1090950 10282896 16686
VVAGISFGV 4 9 x 14 20303 10555107 394032699 404239
VVAGISFGV 5 9 x 13 853 8013945 206262306 224713
VVAGISFGV 6 9 x 12 38 5581421 79183233 111872
VVAGISFGV 7 9 x 11 9 974769 9145863 13439
CFQLIGIFL 4 9 x 14 10973 10184933 377748594 387604
CFQLIGIFL 5 9 x 13 344 7718087 198318132 215446
CFQLIGIFL 6 9 x 12 23 5325777 75325662 94388
CFQLIGIFL 7 9 x 11 6 804256 7676919 11226

YCLSRAITN 4 9 x 14 9422 10853154 413308485 421212
YCLSRAITN 5 9 x 13 364 8148764 216557613 233165
YCLSRAITN 6 9 x 12 17 6012561 89670564 110474
YCLSRAITN 7 9 x 11 8 1298415 12209517 18115
MAKNPPENC 4 9 x 14 3247 7942359 298175094 304607
MAKNPPENC 5 9 x 13 176 7625871 197025300 213803
MAKNPPENC 6 9 x 12 0 5567138 79315272 99197
MAKNPPENC 7 9 x 11 0 1077498 10162188 15073
DCHILNAEA 4 9 x 14 6342 9258622 352295235 360708
DCHILNAEA 5 9 x 13 130 7515467 197293320 213544
DCHILNAEA 6 9 x 12 4 4055784 62109180 76029
DCHILNAEA 7 9 x 11 1 858344 8171505 11938
KSKKICKSL 4 9 x 14 7567 6283939 234559476 239708
KSKKICKSL 5 9 x 13 338 5635306 143493966 156364
KSKKICKSL 6 9 x 12 11 3477828 47812275 60338
KSKKICKSL 7 9 x 11 0 766278 7173306 10730
ICGLVFGIL 4 9 x 14 13973 10953892 406571148 417468
ICGLVFGIL 5 9 x 13 675 9239476 234864531 257147
ICGLVFGIL 6 9 x 12 34 6540238 91234782 115409
ICGLVFGIL 7 9 x 11 0 1026091 9696645 14442
LTLIVLFWG 4 9 x 14 13682 8971254 335934477 344088
LTLIVLFWG 5 9 x 13 772 8601701 222198786 242748
LTLIVLFWG 6 9 x 12 38 6023069 85621014 107616
LTLIVLFWG 7 9 x 11 2 1106924 10343673 15378
KHFWPEVPK 4 9 x 14 3114 7663100 288031617 294941
KHFWPEVPK 5 9 x 13 61 5446893 142742556 153940
KHFWPEVPK 6 9 x 12 1 3621752 53626104 65947
KHFWPEVPK 7 9 x 11 0 909220 8587314 12814
AYDMEHTFY 4 9 x 14 2517 8484309 320458275 326064
AYDMEHTFY 5 9 x 13 30 7501691 194572458 208991
AYDMEHTFY 6 9 x 12 1 5064508 73776681 91293
AYDMEHTFY 7 9 x 11 0 1157261 10838133 16049
NGEKKKIYM 4 9 x 14 12665 7335940 274098330 280156
NGEKKKIYM 5 9 x 13 654 7034159 180906768 197698
NGEKKKIYM 6 9 x 12 30 6341774 90185193 113425
NGEKKKIYM 7 9 x 11 0 1220665 11435913 16942
IDPVTRTEI 4 9 x 14 8899 10540388 398910825 409079
IDPVTRTEI 5 9 x 13 395 8558645 223986393 242526
IDPVTRTEI 6 9 x 12 21 6154078 90007650 112694
IDPVTRTEI 7 9 x 11 0 1299608 12205188 18410
RSGNGTDET 4 9 x 14 10691 9187549 347542236 355570
RSGNGTDET 5 9 x 13 379 7031961 183880089 199300
RSGNGTDET 6 9 x 12 4 6373756 93003687 116020
RSGNGTDET 7 9 x 11 0 1419341 13287843 19886
EVHDFKNGY 4 9 x 14 4645 9416166 359928639 370308
EVHDFKNGY 5 9 x 13 101 7284162 193804110 209406
EVHDFKNGY 6 9 x 12 2 5505597 82040544 101800

EVHDFKNGY 7 9 x 11 0 1247673 11698209 17345
GIYFVGLQK 4 9 x 14 7501 8358907 318358044 326270
GIYFVGLQK 5 9 x 13 165 7984940 212340177 229726
GIYFVGLQK 6 9 x 12 4 5513270 83686860 102942
GIYFVGLQK 7 9 x 11 0 1388323 13018293 19392
FIKTQIKVI 4 9 x 14 6588 8068520 303991740 312381
FIKTQIKVI 5 9 x 13 322 7740261 201216960 218731
FIKTQIKVI 6 9 x 12 10 5924073 84726846 106399
FIKTQIKVI 7 9 x 11 0 1138108 10692945 15799
EFSEPEEEI 4 9 x 14 13930 6971190 259117875 264285
EFSEPEEEI 5 9 x 13 708 6693549 170461305 185894
EFSEPEEEI 6 9 x 12 25 4696818 65155041 81761
EFSEPEEEI 7 9 x 11 0 1080748 10174284 15200
ENEEITTTF 4 9 x 14 9269 7101217 264538026 270941
ENEEITTTF 5 9 x 13 204 5051989 129052710 140868
ENEEITTTF 6 9 x 12 9 2944929 41098536 52051
ENEEITTTF 7 9 x 11 2 701534 6615036 9759
EQSVIWVPA 4 9 x 14 4450 9684381 368172216 376724
EQSVIWVPA 5 9 x 13 114 8938659 234276687 253775
EQSVIWVPA 6 9 x 12 3 6495630 94584231 117668
EQSVIWVPA 7 9 x 11 0 1347062 12573855 18648
KPIENRDFL 4 9 x 14 6933 10265792 396398151 423054
KPIENRDFL 5 9 x 13 251 8321491 224031816 253175
KPIENRDFL 6 9 x 12 15 6373267 96461316 123680
KPIENRDFL 7 9 x 11 1 1536472 14331195 21874
NSKILEICD 4 9 x 14 15503 12510339 475824366 523115
NSKILEICD 5 9 x 13 744 10124609 266589585 308100
NSKILEICD 6 9 x 12 35 6460880 96818031 124663
NSKILEICD 7 9 x 11 1 1478792 13816656 21234
VTMYWINPT 4 9 x 14 1732 7957296 300512394 323311
VTMYWINPT 5 9 x 13 23 5820421 152974260 172897
VTMYWINPT 6 9 x 12 0 4942056 72016911 89204
VTMYWINPT 7 9 x 11 0 1092650 10262376 15132
ISVSELQDF 4 9 x 14 17066 11292250 428638995 439961
ISVSELQDF 5 9 x 13 548 7742717 205334649 222802
ISVSELQDF 6 9 x 12 15 5335194 79505208 98416
ISVSELQDF 7 9 x 11 0 1632017 15241887 22828
EEGEDLHFP 4 9 x 14 11598 8985527 333721152 341649
EEGEDLHFP 5 9 x 13 447 5506481 142151076 154642
EEGEDLHFP 6 9 x 12 8 3521559 49993551 63000
EEGEDLHFP 7 9 x 11 0 906896 8496414 12547
NEKKGIEQN 4 9 x 14 11009 9211266 344433141 354647
NEKKGIEQN 5 9 x 13 399 7032708 180753669 197329
NEKKGIEQN 6 9 x 12 23 4566547 65450241 82244
NEKKGIEQN 7 9 x 11 1 1217635 11452959 17079
QWVVPQVKV 4 9 x 14 3286 5177710 191805228 196316
QWVVPQVKV 5 9 x 13 101 4953474 125945667 137239

QWVVPQVKV 6 9 x 12 5 3132192 43810632 54920
QWVVPQVKV 7 9 x 11 1 734904 7012287 10401
KTRHARQAS 4 9 x 14 5387 8293395 312369219 319454
KTRHARQAS 5 9 x 13 228 7958402 206881362 224120
KTRHARQAS 6 9 x 12 4 5192577 76169511 94551
KTRHARQAS 7 9 x 11 0 1248075 11696877 17482
EELPINDYT 4 9 x 14 8510 10057308 381189222 392202
EELPINDYT 5 9 x 13 258 8290189 215793189 235276
EELPINDYT 6 9 x 12 3 5851410 84359745 105164
EELPINDYT 7 9 x 11 0 1183846 11021157 16332
MASPSRRLQT 4 10 x 16 17152 8768461 484168430 480841
MASPSRRLQT 5 10 x 15 1015 8513952 362095840 370800
MASPSRRLQT 6 10 x 14 23 6629699 195407370 210335
MASPSRRLQT 7 10 x 13 8 6055247 100502010 120100
PVITCFKSVL 4 10 x 16 8521 11181671 620605400 606537
PVITCFKSVL 5 10 x 15 214 9015099 386944500 389160
PVITCFKSVL 6 10 x 14 10 7401969 221404950 234550
PVITCFKSVL 7 10 x 13 8 6404428 108019320 129816
IYTFIFWITG 4 10 x 16 3285 6629518 360331950 353888
IYTFIFWITG 5 10 x 15 135 6036746 251120810 253757
IYTFIFWITG 6 10 x 14 28 5775190 165704500 176951
IYTFIFWITG 7 10 x 13 9 5225723 82462930 101230
ILLAVGIWGK 4 10 x 16 29489 12005276 651518450 647108
ILLAVGIWGK 5 10 x 15 1903 11632604 485003290 491680
ILLAVGIWGK 6 10 x 14 115 9107449 262935070 282010
ILLAVGIWGK 7 10 x 13 11 6508134 103455170 127497
SLENYFSLLN 4 10 x 16 20076 11464199 624687790 621031
SLENYFSLLN 5 10 x 15 790 11123764 464735910 472412
SLENYFSLLN 6 10 x 14 28 9446479 270362090 288964
SLENYFSLLN 7 10 x 13 6 7625241 118727020 145049
KATNVPFVLI 4 10 x 16 12959 12702330 707362210 697361
KATNVPFVLI 5 10 x 15 524 11034553 473512140 478389
KATNVPFVLI 6 10 x 14 39 9114371 272722440 291641
KATNVPFVLI 7 10 x 13 10 6604648 112386060 135313
TGTVIILLGT 4 10 x 16 32197 11467354 617344450 612247
TGTVIILLGT 5 10 x 15 1150 7872609 326665840 329514
TGTVIILLGT 6 10 x 14 80 7520391 215617930 229503
TGTVIILLGT 7 10 x 13 19 5156369 81578500 99459
GCFATCRASA 4 10 x 16 7244 9981682 541510220 526602
GCFATCRASA 5 10 x 15 217 8045541 334649340 340415
GCFATCRASA 6 10 x 14 22 7684875 221055850 235449
GCFATCRASA 7 10 x 13 14 5311439 85218860 102498
MLKLYAMFLT 4 10 x 16 7587 9544109 518541310 507057
MLKLYAMFLT 5 10 x 15 349 9255202 385444500 387609
MLKLYAMFLT 6 10 x 14 12 6615380 191071550 201429
MLKLYAMFLT 7 10 x 13 6 5075388 80149900 97040
VFLVELVAAI 4 10 x 16 29957 8776115 477466050 470069

VFLVELVAAI 5 10 x 15 1879 8516098 355719680 364378
VFLVELVAAI 6 10 x 14 150 8162806 235360270 261786
VFLVELVAAI 7 10 x 13 27 5666856 90329970 116319
GFVFRHEIKN 4 10 x 16 5949 10026032 550365360 597270
GFVFRHEIKN 5 10 x 15 126 7720906 327056230 330118
GFVFRHEIKN 6 10 x 14 17 6734562 197839140 207832
GFVFRHEIKN 7 10 x 13 12 4684271 77676130 92864
FKNNYEKALK 4 10 x 16 15651 7767604 430412220 430529
FKNNYEKALK 5 10 x 15 1033 7542718 322201760 323979
FKNNYEKALK 6 10 x 14 41 5341855 160409950 168253
FKNNYEKALK 7 10 x 13 4 4010632 67356860 80769
YNSTGDYRSH 4 10 x 16 9156 10591269 580184620 583271
YNSTGDYRSH 5 10 x 15 467 10261999 432147150 465290
YNSTGDYRSH 6 10 x 14 23 8251126 240384090 284187
YNSTGDYRSH 7 10 x 13 8 5685894 93686900 113006
VDKIQNTLHC 4 10 x 16 10678 12255928 682873000 689503
VDKIQNTLHC 5 10 x 15 357 10042420 432178610 437777
VDKIQNTLHC 6 10 x 14 23 8280895 247724410 275047
VDKIQNTLHC 7 10 x 13 11 6475658 109129800 142365
GVTDYRDWTD 4 10 x 16 5914 10631974 574884500 588636
GVTDYRDWTD 5 10 x 15 216 10280952 426534890 429800
GVTDYRDWTD 6 10 x 14 5 8280454 236705390 252127
GVTDYRDWTD 7 10 x 13 3 6540308 102419980 124653
NYYSEKGFPK 4 10 x 16 10207 10903239 597823830 599556
NYYSEKGFPK 5 10 x 15 428 8439838 358317740 362332
NYYSEKGFPK 6 10 x 14 18 6366337 187400060 198657
NYYSEKGFPK 7 10 x 13 6 4089832 68356180 81630
CCKLEDCTPQ 4 10 x 16 6194 9379591 511927330 507930
CCKLEDCTPQ 5 10 x 15 196 9102330 381155390 384610
CCKLEDCTPQ 6 10 x 14 9 7160777 205863180 218222
CCKLEDCTPQ 7 10 x 13 5 4743101 75610050 91536
DADKVNNEGC 4 10 x 16 10098 9542464 525171930 520926
DADKVNNEGC 5 10 x 15 478 9264670 392003020 399428
DADKVNNEGC 6 10 x 14 21 7533083 219270990 240399
DADKVNNEGC 7 10 x 13 5 5097655 82746110 101387
IKVMTIIESE 4 10 x 16 7792 8632949 473917050 467620
IKVMTIIESE 5 10 x 15 268 8369499 353415880 356029
IKVMTIIESE 6 10 x 14 17 8026651 234912820 249224
IKVMTIIESE 7 10 x 13 5 5699183 93448740 112239
GVVAGISFGV 4 10 x 16 28527 10898560 590159840 582284
GVVAGISFGV 5 10 x 15 1491 8384696 347692530 354424
GVVAGISFGV 6 10 x 14 88 6215710 176973500 191178
GVVAGISFGV 7 10 x 13 13 5584979 88246190 110648
CFQLIGIFLA 4 10 x 16 16000 10545363 573016980 588282
CFQLIGIFLA 5 10 x 15 815 10206321 426308160 430603
CFQLIGIFLA 6 10 x 14 44 7734973 225125090 239224
CFQLIGIFLA 7 10 x 13 8 5379990 87273140 105115

CLSRAITNNQ 4 10 x 16 15196 13966806 769337230 835883
CLSRAITNNQ 5 10 x 15 704 11697575 495207770 538953
CLSRAITNNQ 6 10 x 14 33 9423351 275977710 293178
CLSRAITNNQ 7 10 x 13 6 6521891 107885590 129953
MAKNPPENCE 4 10 x 16 4995 10220693 556116390 553256
MAKNPPENCE 5 10 x 15 236 7943583 331513720 333036
MAKNPPENCE 6 10 x 14 6 7626259 219037850 231999
MAKNPPENCE 7 10 x 13 0 5567531 88222650 106522
CHILNAEAFK 4 10 x 16 15294 12329924 678851700 671123
CHILNAEAFK 5 10 x 15 455 9984473 423278290 433676
CHILNAEAFK 6 10 x 14 2 7313105 217033980 243579
CHILNAEAFK 7 10 x 13 1 5408440 89960710 108244
KKICKSLKIC 4 10 x 16 11042 9808091 527889390 522822
KKICKSLKIC 5 10 x 15 454 6288378 261325410 262454
KKICKSLKIC 6 10 x 14 6 3943528 113556280 120130
KKICKSLKIC 7 10 x 13 0 3588526 57063600 69021
LVFGILALTL 4 10 x 16 43000 13338131 728834230 728701
LVFGILALTL 5 10 x 15 2744 10566342 445027220 462815
LVFGILALTL 6 10 x 14 277 10106862 296115110 356037
LVFGILALTL 7 10 x 13 7 7532131 123244610 170960
VLFWGSKHFW 4 10 x 16 4001 13478942 732962320 774554
VLFWGSKHFW 5 10 x 15 200 11232373 468166690 471073
VLFWGSKHFW 6 10 x 14 2 8645795 249219960 294860
VLFWGSKHFW 7 10 x 13 0 5990697 96635220 119830
EVPKKAYDME 4 10 x 16 11557 11098084 606805230 620716
EVPKKAYDME 5 10 x 15 294 9725326 406879870 411491
EVPKKAYDME 6 10 x 14 5 7070569 205279370 218007
EVPKKAYDME 7 10 x 13 0 6418876 103908220 125829
TFYSNGEKKK 4 10 x 16 17210 12267135 674552830 669388
TFYSNGEKKK 5 10 x 15 746 9910236 420316140 422171
TFYSNGEKKK 6 10 x 14 31 7465681 222418280 234464
TFYSNGEKKK 7 10 x 13 2 5574581 93417220 111451
YMEIDPVTRT 4 10 x 16 10519 11263843 622030070 610839
YMEIDPVTRT 5 10 x 15 407 8901670 380271840 381217
YMEIDPVTRT 6 10 x 14 26 7058247 210589680 221970
YMEIDPVTRT 7 10 x 13 0 5005286 85026390 101146
IFRSGNGTDE 4 10 x 16 14454 10856067 601277330 589331
IFRSGNGTDE 5 10 x 15 576 10523839 449952920 450889
IFRSGNGTDE 6 10 x 14 13 8716366 258892950 273125
IFRSGNGTDE 7 10 x 13 0 6130826 104360780 123898
LEVHDFKNGY 4 10 x 16 7937 13063445 724191240 780418
LEVHDFKNGY 5 10 x 15 224 10850575 463628190 519701
LEVHDFKNGY 6 10 x 14 6 9176523 271998020 287584
LEVHDFKNGY 7 10 x 13 0 6916531 115241030 137607
GIYFVGLQKC 4 10 x 16 11785 11989687 655708550 655976
GIYFVGLQKC 5 10 x 15 266 8364626 354996480 357317
GIYFVGLQKC 6 10 x 14 7 7989185 236985100 251475

GIYFVGLQKC 7 10 x 13 0 5519167 93748760 111879
IKTQIKVIPE 4 10 x 16 10397 9089278 497973810 494761
IKTQIKVIPE 5 10 x 15 525 6798159 287427170 287681
IKTQIKVIPE 6 10 x 14 17 6528616 191149780 201777
IKTQIKVIPE 7 10 x 13 0 5955676 97376610 117066
SEPEEEIDEN 4 10 x 16 23215 7797563 423791070 418211
SEPEEEIDEN 5 10 x 15 1420 5699290 236952960 237159
SEPEEEIDEN 6 10 x 14 88 5476119 156259110 165345
SEPEEEIDEN 7 10 x 13 11 4934900 77536420 93882
EITTTFFEQS 4 10 x 16 7836 7199658 391574090 382126
EITTTFFEQS 5 10 x 15 185 6979831 291023480 292347
EITTTFFEQS 6 10 x 14 14 5463381 155812600 164955
EITTTFFEQS 7 10 x 13 0 4928267 77307180 94184
IWVPAEKPIE 4 10 x 16 12847 12345444 672321850 661083
IWVPAEKPIE 5 10 x 15 497 10153401 423871890 427623
IWVPAEKPIE 6 10 x 14 7 7829913 225832170 240392
IWVPAEKPIE 7 10 x 13 0 5054273 82375730 99251
RDFLKNSKIL 4 10 x 16 27339 13596257 751622590 737145
RDFLKNSKIL 5 10 x 15 1373 11022642 469857030 473573
RDFLKNSKIL 6 10 x 14 41 9214905 271850180 289395
RDFLKNSKIL 7 10 x 13 0 6789634 112836030 135377
ICDNVTMYWI 4 10 x 16 3495 10343646 566485740 554496
ICDNVTMYWI 5 10 x 15 84 9214394 386210430 389068
ICDNVTMYWI 6 10 x 14 0 7060638 204697220 219012
ICDNVTMYWI 7 10 x 13 0 4677329 76504790 92264
PTLISVSELQ 4 10 x 16 28803 13116325 723000030 717218
PTLISVSELQ 5 10 x 15 1696 12726248 540604130 557265
PTLISVSELQ 6 10 x 14 60 10276099 303568470 329634
PTLISVSELQ 7 10 x 13 2 7455404 124350020 152407
FEEEGEDLHF 4 10 x 16 20175 7264452 393218310 393081
FEEEGEDLHF 5 10 x 15 1606 5393062 223006380 228691
FEEEGEDLHF 6 10 x 14 95 5150681 146930840 159268
FEEEGEDLHF 7 10 x 13 2 2847834 45713760 56395
ANEKKGIEQN 4 10 x 16 16212 11947400 651941930 658137
ANEKKGIEQN 5 10 x 15 596 9709542 406253010 422064
ANEKKGIEQN 6 10 x 14 35 7288242 211702930 230058
ANEKKGIEQN 7 10 x 13 1 6630483 107052670 133085
QWVVPQVKVE 4 10 x 16 5559 5356908 290593750 290855
QWVVPQVKVE 5 10 x 15 170 5185815 215856580 222406
QWVVPQVKVE 6 10 x 14 6 4962421 142562010 154611
QWVVPQVKVE 7 10 x 13 1 3152945 50478680 63155
TRHARQASEE 4 10 x 16 10197 7973869 437491110 439456
TRHARQASEE 5 10 x 15 498 7731975 326269940 337679
TRHARQASEE 6 10 x 14 13 6210364 180522430 195429
TRHARQASEE 7 10 x 13 2 5629694 91508240 113121
LPINDYTENG 4 10 x 16 10009 13025829 722123160 727137
LPINDYTENG 5 10 x 15 219 10780044 460849150 479919

LPINDYTENG 6 10 x 14 0 9350259 277078850 302219
LPINDYTENG 7 10 x 13 0 7082930 118051850 145996
EFDPMLDERG 4 10 x 16 9299 10836263 594548300 597814
EFDPMLDERG 5 10 x 15 442 10509943 443344970 455575
EFDPMLDERG 6 10 x 14 10 6980831 207502320 219284
EFDPMLDERG 7 10 x 13 0 5662061 93992190 113039
CCIYCRRGNR 4 10 x 16 2412 5102125 276647390 269125
CCIYCRRGNR 5 10 x 15 87 4936748 205564340 207157
CCIYCRRGNR 6 10 x 14 0 3181949 92200950 97361
CCIYCRRGNR 7 10 x 13 0 2886036 46553920 56235
CRRVCEPLLG 4 10 x 16 10029 7821890 428030410 421077
CRRVCEPLLG 5 10 x 15 209 6032222 253867510 254805
CRRVCEPLLG 6 10 x 14 4 3878686 113982870 120951
CRRVCEPLLG 7 10 x 13 0 3535647 58521590 70131
YPYPYCYQGG 4 10 x 16 1446 3103529 167366310 163226
YPYPYCYQGG 5 10 x 15 40 2998352 124155670 124304
YPYPYCYQGG 6 10 x 14 0 2461067 69759770 73123
YPYPYCYQGG 7 10 x 13 0 2215691 34430920 41526
MASPSRRLQT 8 10 x 12 6 1418861 14856800 20724
MASPSRRLQT 9 10 x 11 4 17544 176630 243
PVITCFKSVL 8 10 x 12 4 1554192 16146070 22500
PVITCFKSVL 9 10 x 11 2 16474 165810 233
IYTFIFWITG 8 10 x 12 6 1003781 10574990 14729
IYTFIFWITG 9 10 x 11 4 14743 148600 209
ILLAVGIWGK 8 10 x 12 6 1186133 12333110 17301
ILLAVGIWGK 9 10 x 11 2 20719 208550 291
SLENYFSLLN 8 10 x 12 4 1470970 15308080 21549
SLENYFSLLN 9 10 x 11 2 17386 174890 243
KATNVPFVLI 8 10 x 12 6 1716218 17810020 24873
KATNVPFVLI 9 10 x 11 4 17861 179900 250
TGTVIILLGT 8 10 x 12 5 889560 9317810 13032
TGTVIILLGT 9 10 x 11 3 19205 193690 272
GCFATCRASA 8 10 x 12 6 977746 10264850 14378
GCFATCRASA 9 10 x 11 1 12301 123800 177
MLKLYAMFLT 8 10 x 12 4 1314877 13758780 19225
MLKLYAMFLT 9 10 x 11 2 18007 181290 252
VFLVELVAAI 8 10 x 12 10 1506688 15742130 22002
VFLVELVAAI 9 10 x 11 3 19110 192910 271
GFVFRHEIKN 8 10 x 12 6 1451546 15096200 21177
GFVFRHEIKN 9 10 x 11 0 15527 155890 217
FKNNYEKALK 8 10 x 12 3 1256652 13182100 18432
FKNNYEKALK 9 10 x 11 1 16062 161760 224
YNSTGDYRSH 8 10 x 12 5 1248106 13096020 18286
YNSTGDYRSH 9 10 x 11 3 16550 166700 233
VDKIQNTLHC 8 10 x 12 4 1569315 16306370 22931
VDKIQNTLHC 9 10 x 11 2 17210 173380 242
GVTDYRDWTD 8 10 x 12 3 1256013 13071100 18209

GVTDYRDWTD 9 10 x 11 2 14877 149900 207
NYYSEKGFPK 8 10 x 12 4 782139 8227830 11291
NYYSEKGFPK 9 10 x 11 2 16817 169800 237
CCKLEDCTPQ 8 10 x 12 3 640475 6719520 9269
CCKLEDCTPQ 9 10 x 11 2 6529 65940 95
DADKVNNEGC 8 10 x 12 3 986478 10274410 14184
DADKVNNEGC 9 10 x 11 2 17977 181230 262
IKVMTIIESE 8 10 x 12 3 1381724 14411790 20221
IKVMTIIESE 9 10 x 11 1 16725 168060 232
GVVAGISFGV 8 10 x 12 6 960402 10037100 14143
GVVAGISFGV 9 10 x 11 4 21380 215760 300
CFQLIGIFLA 8 10 x 12 6 904934 9615430 13401
CFQLIGIFLA 9 10 x 11 4 13746 138640 199
CLSRAITNNQ 8 10 x 12 6 1502579 15646640 21918
CLSRAITNNQ 9 10 x 11 4 16420 165380 236
MAKNPPENCE 8 10 x 12 0 1079967 11344250 15768
MAKNPPENCE 9 10 x 11 0 13222 132760 186
CHILNAEAFK 8 10 x 12 0 982245 10419130 14369
CHILNAEAFK 9 10 x 11 0 12284 123610 177
KKICKSLKIC 8 10 x 12 0 773708 8075710 11329
KKICKSLKIC 9 10 x 11 0 9367 94450 133
LVFGILALTL 8 10 x 12 0 1688553 17634280 24930
LVFGILALTL 9 10 x 11 0 22097 222570 317
VLFWGSKHFW 8 10 x 12 0 1501227 15528050 21813
VLFWGSKHFW 9 10 x 11 0 15145 152400 208
EVPKKAYDME 8 10 x 12 0 1363622 14201700 20024
EVPKKAYDME 9 10 x 11 0 15534 156290 287
TFYSNGEKKK 8 10 x 12 0 1255428 13163100 18250
TFYSNGEKKK 9 10 x 11 0 15766 158730 219
YMEIDPVTRT 8 10 x 12 0 1176339 12393440 17116
YMEIDPVTRT 9 10 x 11 0 16358 164830 235
IFRSGNGTDE 8 10 x 12 0 1498411 15652640 21917
IFRSGNGTDE 9 10 x 11 0 17781 179100 246
LEVHDFKNGY 8 10 x 12 0 1895139 19574720 27516
LEVHDFKNGY 9 10 x 11 0 18310 184210 264
GIYFVGLQKC 8 10 x 12 0 1433234 14978500 21350
GIYFVGLQKC 9 10 x 11 0 18387 185120 267
IKTQIKVIPE 8 10 x 12 0 1376813 14367550 20183
IKTQIKVIPE 9 10 x 11 0 16792 169360 238
SEPEEEIDEN 8 10 x 12 0 1263180 13221700 18542
SEPEEEIDEN 9 10 x 11 0 18571 187010 264
EITTTFFEQS 8 10 x 12 0 1231885 12865090 18140
EITTTFFEQS 9 10 x 11 0 14210 143040 197
IWVPAEKPIE 8 10 x 12 0 1054243 11112330 15563
IWVPAEKPIE 9 10 x 11 0 13537 136600 261
RDFLKNSKIL 8 10 x 12 0 1385224 14531710 20138
RDFLKNSKIL 9 10 x 11 0 18000 181480 251

ICDNVTMYWI 8 10 x 12 0 994995 10446650 14407
ICDNVTMYWI 9 10 x 11 0 12825 129180 182
PTLISVSELQ 8 10 x 12 0 1788793 18638020 26280
PTLISVSELQ 9 10 x 11 0 19355 194970 291
FEEEGEDLHF 8 10 x 12 0 770012 8096450 11368
FEEEGEDLHF 9 10 x 11 0 18811 189910 273
ANEKKGIEQN 8 10 x 12 0 1391426 14529060 20390
ANEKKGIEQN 9 10 x 11 0 15042 151260 223
QWVVPQVKVE 8 10 x 12 0 842717 8944630 12543
QWVVPQVKVE 9 10 x 11 0 13085 131940 185
TRHARQASEE 8 10 x 12 0 1113262 11726320 16394
TRHARQASEE 9 10 x 11 0 17241 173610 241
LPINDYTENG 8 10 x 12 0 1762676 18282240 25714
LPINDYTENG 9 10 x 11 0 18085 181860 253
EFDPMLDERG 8 10 x 12 0 1314725 13742910 19250
EFDPMLDERG 9 10 x 11 0 16188 163050 227
CCIYCRRGNR 8 10 x 12 0 510716 5389430 7472
CCIYCRRGNR 9 10 x 11 0 5477 55320 100
CRRVCEPLLG 8 10 x 12 0 611524 6458880 8988
CRRVCEPLLG 9 10 x 11 0 15511 156630 224
YPYPYCYQGG 8 10 x 12 0 503545 5280720 7385
YPYPYCYQGG 9 10 x 11 0 11802 118990 172
MASPSRRLQ 8 9 x 10 4 15901 144036 217
MASPSRRLQ 9 9 x 9 2 112 855 2
KPVITCFKS 8 9 x 10 1 14643 132552 192
KPVITCFKS 9 9 x 9 0 123 927 2
LLIYTFIFW 8 9 x 10 4 9700 87975 134
LLIYTFIFW 9 9 x 9 2 181 1494 3
TGVILLAVG 8 9 x 10 4 18106 163980 255
TGVILLAVG 9 9 x 9 0 111 819 2
WGKVSLENY 8 9 x 10 6 15925 144468 211
WGKVSLENY 9 9 x 9 2 106 801 2
SLLNEKATN 8 9 x 10 2 19383 175563 252
SLLNEKATN 9 9 x 9 0 68 432 1
PFVLIATGT 8 9 x 10 1 15308 138564 200
PFVLIATGT 9 9 x 9 1 105 783 2
IILLGTFGC 8 9 x 10 4 9524 86436 131
IILLGTFGC 9 9 x 9 2 185 1521 4
ATCRASAWM 8 9 x 10 2 14625 132678 195
ATCRASAWM 9 9 x 9 1 209 1719 3
KLYAMFLTL 8 9 x 10 4 15253 137862 199
KLYAMFLTL 9 9 x 9 0 63 387 1
FLVELVAAI 8 9 x 10 6 18756 170073 245
FLVELVAAI 9 9 x 9 2 114 873 2
GFVFRHEIK 8 9 x 10 0 14017 126603 182
GFVFRHEIK 9 9 x 9 0 59 351 1
SFKNNYEKA 8 9 x 10 1 13975 126603 190

SFKNNYEKA 9 9 x 9 0 69 441 4
KQYNSTGDY 8 9 x 10 1 13389 121185 173
KQYNSTGDY 9 9 x 9 1 106 792 2
SHAVDKIQN 8 9 x 10 2 15189 137664 201
SHAVDKIQN 9 9 x 9 1 104 783 2
LHCCGVTDY 8 9 x 10 2 13608 123300 176
LHCCGVTDY 9 9 x 9 1 99 729 2
DWTDTNYYS 8 9 x 10 1 11362 103131 151
DWTDTNYYS 9 9 x 9 0 106 783 2
KGFPKSCCK 8 9 x 10 5 13694 124020 182
KGFPKSCCK 9 9 x 9 1 176 1413 3
EDCTPQRDA 8 9 x 10 2 13827 125208 182
EDCTPQRDA 9 9 x 9 1 103 774 2
KVNNEGCFI 8 9 x 10 1 13498 121941 174
KVNNEGCFI 9 9 x 9 0 55 315 1
VMTIIESEM 8 9 x 10 2 13812 125253 182
VMTIIESEM 9 9 x 9 1 105 783 2
VVAGISFGV 8 9 x 10 4 10462 94986 137
VVAGISFGV 9 9 x 9 2 194 1611 3
CFQLIGIFL 8 9 x 10 4 11982 108720 159
CFQLIGIFL 9 9 x 9 2 94 693 2
YCLSRAITN 8 9 x 10 4 13602 123318 181
YCLSRAITN 9 9 x 9 2 105 792 2
MAKNPPENC 8 9 x 10 0 12755 115164 165
MAKNPPENC 9 9 x 9 0 69 441 2
DCHILNAEA 8 9 x 10 0 11115 100566 148
DCHILNAEA 9 9 x 9 0 96 684 2
KSKKICKSL 8 9 x 10 0 15300 138897 200
KSKKICKSL 9 9 x 9 0 256 2133 4
ICGLVFGIL 8 9 x 10 0 12831 116370 169
ICGLVFGIL 9 9 x 9 0 113 837 2
LTLIVLFWG 8 9 x 10 0 20456 185526 272
LTLIVLFWG 9 9 x 9 0 254 2115 4
KHFWPEVPK 8 9 x 10 0 11522 104409 158
KHFWPEVPK 9 9 x 9 0 90 639 2
AYDMEHTFY 8 9 x 10 0 13542 122409 181
AYDMEHTFY 9 9 x 9 0 90 630 1
NGEKKKIYM 8 9 x 10 0 13606 123156 180
NGEKKKIYM 9 9 x 9 0 117 873 2
IDPVTRTEI 8 9 x 10 0 16054 145692 212
IDPVTRTEI 9 9 x 9 0 113 837 2
RSGNGTDET 8 9 x 10 0 16665 150966 220
RSGNGTDET 9 9 x 9 0 114 846 2
EVHDFKNGY 8 9 x 10 0 14321 129681 186
EVHDFKNGY 9 9 x 9 0 103 756 2
GIYFVGLQK 8 9 x 10 0 17058 154557 222
GIYFVGLQK 9 9 x 9 0 164 1296 2

```
FIKTQIKVI 8 9 x 10 0 14226 129114 187
FIKTQIKVI 9 9 x 9 0 106 783 2
EFSEPEEEI 8 9 x 10 0 14913 135189 270
EFSEPEEEI 9 9 x 9 0 236 1953 3
ENEEITTTF 8 9 x 10 0 15098 136971 200
ENEEITTTF 9 9 x 9 0 199 1629 3
EQSVIWVPA 8 9 x 10 0 14102 127701 183
EQSVIWVPA 9 9 x 9 0 109 801 1
KPIENRDFL 8 9 x 10 0 15397 139329 205
KPIENRDFL 9 9 x 9 0 101 738 4
NSKILEICD 8 9 x 10 0 16237 147006 213
NSKILEICD 9 9 x 9 0 116 864 2
VTMYWINPT 8 9 x 10 0 13240 120006 173
VTMYWINPT 9 9 x 9 0 98 711 5
ISVSELQDF 8 9 x 10 0 19486 176616 252
ISVSELQDF 9 9 x 9 0 115 855 2
EEGEDLHFP 8 9 x 10 0 9734 88164 130
EEGEDLHFP 9 9 x 9 0 214 1746 3
NEKKGIEQN 8 9 x 10 0 14529 131679 190
NEKKGIEQN 9 9 x 9 0 85 585 2
QWVVPQVKV 8 9 x 10 0 11249 102060 149
QWVVPQVKV 9 9 x 9 0 152 1197 3
KTRHARQAS 8 9 x 10 0 13982 126495 184
KTRHARQAS 9 9 x 9 0 105 774 2
EELPINDYT 8 9 x 10 0 9213 83538 121
EELPINDYT 9 9 x 9 0 166 1323 2
```

The persistent results set

```
file name /local/pj\_test\_n2/elaTests/Prot/ensemblProteins batch size 10
query thresh DPsize Matches nodes seen
depth calculated timeMs
MASPS 3 5 x 7 253556 1482502 7493725 1624480
MASPS 4 5 x 6 7705 9957 49690 33372
RLQTK 3 5 x 7 154572 1615771 8146475 1581165
RLQTK 4 5 x 6 4972 8782 43815 15857
VITCF 3 5 x 7 103503 1587286 8021675 1845468
VITCF 4 5 x 6 218 8747 43640 19958
SVLLI 3 5 x 7 551218 1764665 8861480 2583183
SVLLI 4 5 x 6 17526 10220 51005 7522
TFIFW 3 5 x 7 86871 1268109 6436675 1450560
TFIFW 4 5 x 6 551 8607 42950 4151
TGVIL 3 5 x 7 239852 1864335 9385850 1581369
TGVIL 4 5 x 6 11413 9243 46120 26989
AVGIW 3 5 x 7 235654 1761698 8877865 2041409
AVGIW 4 5 x 6 6423 10281 51310 28581
```

KVSLE 3 5 x 7 335204 1886158 9481975 2255924
KVSLE 4 5 x 6 8723 9943 49620 25337
YFSLL 3 5 x 7 381151 1494152 7544075 1718443
YFSLL 4 5 x 6 9512 10065 50230 6083
MASPSR 4 6 x 8 17581 1824121 11137368 1588655
MASPSR 5 6 x 7 544 13336 80058 28575
LQTKPV 4 6 x 8 9192 1991232 12118554 1548989
LQTKPV 5 6 x 7 156 13021 78144 26689
TCFKSV 4 6 x 8 1444 1566185 9574446 517083
TCFKSV 5 6 x 7 9 10398 62592 1685
LIYTFI 4 6 x 8 4091 1688689 10301442 1543940
LIYTFI 5 6 x 7 103 11068 66432 15296
WITGVI 4 6 x 8 8734 1461500 8956248 1234223
WITGVI 5 6 x 7 302 11575 69534 1966
LAVGIW 4 6 x 8 22116 2296312 13943562 2070834
LAVGIW 5 6 x 7 574 14266 85596 67656
KVSLEN 4 6 x 8 18920 2220637 13493934 1972871
KVSLEN 5 6 x 7 659 13886 83250 27868
FSLLNE 4 6 x 8 31015 2026700 12332538 1838469
FSLLNE 5 6 x 7 739 14341 85980 3686
ATNVPF 4 6 x 8 9175 2120991 12918534 1554119
ATNVPF 5 6 x 7 181 14637 87888 18999
MASPSRR 5 7 x 9 1704 1904288 13624331 1163110
MASPSRR 6 7 x 8 8 15053 105700 43534
QTKPVIT 5 7 x 9 448 1909837 13640564 1394202
QTKPVIT 6 7 x 8 29 15581 109564 3688
FKSVLLI 5 7 x 9 1172 2225993 15850828 1225658
FKSVLLI 6 7 x 8 29 13659 95711 42469
TFIFWIT 5 7 x 9 64 1344727 9659566 1281364
TFIFWIT 6 7 x 8 5 11663 81907 2871
VILLAVG 5 7 x 9 3798 2267295 16140761 1571327
VILLAVG 6 7 x 8 169 17173 120442 45114
WGKVSLE 5 7 x 9 810 1847046 13187356 1139873
WGKVSLE 6 7 x 8 36 14400 101164 2052
YFSLLNE 5 7 x 9 1188 2094002 14957040 1507712
YFSLLNE 6 7 x 8 12 16156 113435 3046
ATNVPFV 5 7 x 9 604 2178115 15536913 1556360
ATNVPFV 6 7 x 8 8 16226 114100 19852
IATGTVI 5 7 x 9 1296 2209088 15770174 1701941
IATGTVI 6 7 x 8 65 17413 122206 63256
MASPSRRL 6 8 x 10 54 2263244 18515912 1567403
MASPSRRL 7 8 x 9 4 17259 138720 42883
TKPVITCF 6 8 x 10 43 2160804 17655928 1523076
TKPVITCF 7 8 x 9 10 16730 134592 16929
SVLLIYTF 6 8 x 10 51 2669988 21718520 2266510
SVLLIYTF 7 8 x 9 6 18239 146472 3863
FWITGVIL 6 8 x 10 103 1838415 15062512 1091421

FWITGVIL 7 8 x 9 19 13638 109720 2096
AVGIWGKV 6 8 x 10 16 2177875 17755520 1900312
AVGIWGKV 7 8 x 9 0 14718 118120 37167
LENYFSLL 6 8 x 10 51 2194416 17901424 1597877
LENYFSLL 7 8 x 9 6 16023 128776 33737
EKATNVPF 6 8 x 10 26 2674272 21760960 1673232
EKATNVPF 7 8 x 9 5 16522 132592 6155
LIATGTVI 6 8 x 10 108 2597999 21183392 1792756
LIATGTVI 7 8 x 9 5 16942 136032 3425
LLGTFGCF 6 8 x 10 49 1280528 10457160 660330
LLGTFGCF 7 8 x 9 7 9559 76776 1798
MASPSRRLQ 6 9 x 12 112 22918911 283692348 3954111
MASPSRRLQ 7 9 x 11 8 2523731 23251338 1773687
MASPSRRLQ 8 9 x 10 3 19218 173817 50034
KPVITCFKS 6 9 x 12 26 21432371 266154345 4306590
KPVITCFKS 7 9 x 11 4 2388630 21934359 2016994
KPVITCFKS 8 9 x 10 2 17367 157149 46973
LLIYTFIFW 6 9 x 12 50 16173210 190771353 3362900
LLIYTFIFW 7 9 x 11 8 1422008 13089069 1201663
LLIYTFIFW 8 9 x 10 3 8593 77760 3047
MASPSRRLQ 7 9 x 11 8 2523731 23251338 1546242
MASPSRRLQ 8 9 x 10 3 19218 173817 59940
KPVITCFKS 7 9 x 11 4 2388630 21934359 1823703
KPVITCFKS 8 9 x 10 2 17367 157149 36305
LLIYTFIFW 7 9 x 11 8 1422008 13089069 1237973
LLIYTFIFW 8 9 x 10 3 8593 77760 4766
TGVILLAVG 7 9 x 11 47 2597780 23900922 1953380
TGVILLAVG 8 9 x 10 9 18341 165825 31476
WGKVSLENY 7 9 x 11 12 2202845 20299941 1300038
WGKVSLENY 8 9 x 10 9 17748 160821 3319
SLLNEKATN 7 9 x 11 10 1956088 17929755 1513786
SLLNEKATN 8 9 x 10 6 21284 192555 15681
PFVLIATGT 7 9 x 11 19 2771447 25485219 1697728
PFVLIATGT 8 9 x 10 2 20837 188550 21038
IILLGTFGC 7 9 x 11 8 1615726 14868567 1217626
IILLGTFGC 8 9 x 10 3 11108 100431 6255
ATCRASAWM 7 9 x 11 10 1951447 18027882 1341129
ATCRASAWM 8 9 x 10 4 19339 175167 17754
MASPSRRLQT 7 10 x 13 11 23250727 326887160 4744493
MASPSRRLQT 8 10 x 12 6 2795725 28639370 2416050
MASPSRRLQT 9 10 x 11 3 21169 212870 182571
PVITCFKSVL 7 10 x 13 7 22975414 327534200 4923826
PVITCFKSVL 8 10 x 12 3 2863418 29201760 2332767
PVITCFKSVL 9 10 x 11 1 18917 190070 117180
IYTFIFWITG 7 10 x 13 9 17795660 237289970 4129212
IYTFIFWITG 8 10 x 12 6 1835094 18927000 1692004
IYTFIFWITG 9 10 x 11 6 17839 179620 55507

ILLAVGIWGK 7 10 x 13 61 21878659 294641210 4141784
ILLAVGIWGK 8 10 x 12 18 1843168 18828700 1420536
ILLAVGIWGK 9 10 x 11 6 24165 243360 69731
SLENYFSLLN 7 10 x 13 22 24906281 330786630 5390062
SLENYFSLLN 8 10 x 12 9 2446027 25030800 2268657
SLENYFSLLN 9 10 x 11 6 20430 205440 50340
KATNVPFVLI 7 10 x 13 13 23495631 337762230 4783923
KATNVPFVLI 8 10 x 12 6 3148975 32118120 2199986
KATNVPFVLI 9 10 x 11 3 21540 216410 185122
TGTVIILLGT 7 10 x 13 25 17262695 231382440 5986231
TGTVIILLGT 8 10 x 12 6 1517318 15573770 2570789
TGTVIILLGT 9 10 x 11 3 18849 189600 71548
GCFATCRASA 7 10 x 13 37 18947860 259804970 3873482
GCFATCRASA 8 10 x 12 10 1841140 18940930 1140265
GCFATCRASA 9 10 x 11 4 16024 161180 44954
MLKLYAMFLT 7 10 x 13 21 15143280 204715200 3194328
MLKLYAMFLT 8 10 x 12 18 2100934 21541150 1575215
MLKLYAMFLT 9 10 x 11 12 19992 201260 77794
MASPSRRLQTK 7 11 x 15 16 27657805 808146713 5935127
MASPSRRLQTK 8 11 x 14 9 23541892 371462817 4975578
MASPSRRLQTK 9 11 x 13 6 3059399 34485935 2279079
MASPSRRLQTK 10 11 x 12 3 22985 254265 185881
VITCFKSVLLI 7 11 x 15 7 27189026 797415553 6276299
VITCFKSVLLI 8 11 x 14 4 19070477 299570095 4578039
VITCFKSVLLI 9 11 x 13 2 2826139 31802551 2629181
VITCFKSVLLI 10 11 x 12 0 19405 214368 74616
TFIFWITGVIL 7 11 x 15 31 19479446 564852574 4675691
TFIFWITGVIL 8 11 x 14 14 15209985 232200210 3346721
TFIFWITGVIL 9 11 x 13 9 2376841 26880810 1906652
TFIFWITGVIL 10 11 x 12 3 19445 215347 65010
MASPSRRLQTK 9 11 x 13 6 3059399 34485935 1713060
MASPSRRLQTK 10 11 x 12 3 22985 254265 69025
MASPSRRLQTK 11 11 x 11 3 122 1166 94
VITCFKSVLLI 9 11 x 13 2 2826139 31802551 1887892
VITCFKSVLLI 10 11 x 12 0 19405 214368 49792
VITCFKSVLLI 11 11 x 11 0 126 1166 126
TFIFWITGVIL 9 11 x 13 9 2376841 26880810 1447311
TFIFWITGVIL 10 11 x 12 3 19445 215347 16634
TFIFWITGVIL 11 11 x 11 0 74 594 40
AVGIWGKVSLE 9 11 x 13 6 3263409 36604612 2104211
AVGIWGKVSLE 10 11 x 12 0 22176 244827 68428
AVGIWGKVSLE 11 11 x 11 0 80 660 2
YFSLLNEKATN 9 11 x 13 9 2959200 33386452 1612433
YFSLLNEKATN 10 11 x 12 3 23199 256806 14038
YFSLLNEKATN 11 11 x 11 3 123 1177 71
PFVLIATGTVI 9 11 x 13 5 2887792 32608136 1611608
PFVLIATGTVI 10 11 x 12 2 23893 264539 25650

PFVLIATGTVI 11 11 x 11 1 125 1199 41
LLGTFGCFATC 9 11 x 13 6 1541040 17369022 1267862
LLGTFGCFATC 10 11 x 12 6 11415 126247 3078
LLGTFGCFATC 11 11 x 11 3 203 2090 77
ASAWMLKLYAM 9 11 x 13 10 1806105 20351628 1389239
ASAWMLKLYAM 10 11 x 12 5 21565 238788 9130
ASAWMLKLYAM 11 11 x 11 2 191 1980 54
LTLVFLVELVA 9 11 x 13 2 1792422 20221949 1370643
LTLVFLVELVA 10 11 x 12 1 24260 268433 7835
LTLVFLVELVA 11 11 x 11 0 210 2123 21
MASPSRRLQTKP 9 12 x 15 9 23552458 406385340 5150389
MASPSRRLQTKP 10 12 x 14 6 3098043 38171280 2340939
MASPSRRLQTKP 11 12 x 13 3 24046 290232 185079
MASPSRRLQTKP 12 12 x 12 3 122 1284 302
ITCFKSVLLIYT 9 12 x 15 6 17133963 303546864 3771345
ITCFKSVLLIYT 10 12 x 14 2 2541958 31380192 1789035
ITCFKSVLLIYT 11 12 x 13 1 21989 265440 84910
ITCFKSVLLIYT 12 12 x 12 0 99 948 54
IFWITGVILLAV 9 12 x 15 31 11106905 197337984 2379287
IFWITGVILLAV 10 12 x 14 22 2282771 28309116 1312396
IFWITGVILLAV 11 12 x 13 11 24906 300864 98769
IFWITGVILLAV 12 12 x 12 3 260 2964 64
IWGKVSLENYFS 9 12 x 15 18 18014388 323728248 4445064
IWGKVSLENYFS 10 12 x 14 12 2693256 33183336 1638770
IWGKVSLENYFS 11 12 x 13 6 21414 258840 14646
IWGKVSLENYFS 12 12 x 12 3 114 1188 78
LNEKATNVPFVL 9 12 x 15 15 23921346 409506576 5266883
LNEKATNVPFVL 10 12 x 14 9 3076558 37779240 2405323
LNEKATNVPFVL 11 12 x 13 6 24460 295188 115373
LNEKATNVPFVL 12 12 x 12 0 63 516 5
ATGTVIILLGTF 9 12 x 15 7 19725647 329466828 4827025
ATGTVIILLGTF 10 12 x 14 4 3056287 37640856 2362097
ATGTVIILLGTF 11 12 x 13 1 26892 324588 99996
ATGTVIILLGTF 12 12 x 12 1 129 1356 64
CFATCRASAWML 9 12 x 15 16 19364463 328028076 4168845
CFATCRASAWML 10 12 x 14 10 2330488 28798080 1530234
CFATCRASAWML 11 12 x 13 6 19385 234804 4866
CFATCRASAWML 12 12 x 12 2 166 1848 60
LYAMFLTLVFLV 9 12 x 15 33 21529198 352370364 4806001
LYAMFLTLVFLV 10 12 x 14 8 2568901 31686216 2251112
LYAMFLTLVFLV 11 12 x 13 3 22872 276312 48861
LYAMFLTLVFLV 12 12 x 12 1 201 2220 90
LVAAIVGFVFRH 9 12 x 15 22 22728479 374501556 5506253
LVAAIVGFVFRH 10 12 x 14 15 3310026 40709484 2793171
LVAAIVGFVFRH 11 12 x 13 9 27041 326556 108360
LVAAIVGFVFRH 12 12 x 12 3 124 1296 94
MASPSRRLQTKPV 9 13 x 17 12 27708233 974766325 6268957

MASPSRRLQTKPV 10 13 x 16 9 23966566 457688608 5252873
MASPSRRLQTKPV 11 13 x 15 6 3388126 45218784 2378682
MASPSRRLQTKPV 12 13 x 14 3 25888 338546 150758
MASPSRRLQTKPV 13 13 x 13 3 122 1404 103
TCFKSVLLIYTFI 9 13 x 17 17 27668266 965991949 6809818
TCFKSVLLIYTFI 10 13 x 16 12 16451737 316912336 3856834
TCFKSVLLIYTFI 11 13 x 15 3 2449287 32897319 1472837
TCFKSVLLIYTFI 12 13 x 14 1 21000 274937 71484
TCFKSVLLIYTFI 13 13 x 13 1 137 1599 69
WITGVILLAVGIW 9 13 x 17 34 29136374 999177712 7549167
WITGVILLAVGIW 10 13 x 16 26 18486944 342918511 4812742
WITGVILLAVGIW 11 13 x 15 14 2354832 31758025 1680974
WITGVILLAVGIW 12 13 x 14 9 24904 326885 110192
WITGVILLAVGIW 13 13 x 13 3 120 1339 49
KVSLENYFSLLNE 9 13 x 17 21 39107922 1321559070 10047181
KVSLENYFSLLNE 10 13 x 16 12 26604031 481226798 6913588
KVSLENYFSLLNE 11 13 x 15 9 2989526 39920712 2795986
KVSLENYFSLLNE 12 13 x 14 6 23970 313599 127741
KVSLENYFSLLNE 13 13 x 13 3 119 1365 56
ATNVPFVLIATGT 9 13 x 17 13 35484938 1236715714 8388798
ATNVPFVLIATGT 10 13 x 16 10 25032262 470054390 5658701
ATNVPFVLIATGT 11 13 x 15 6 3380200 45110832 2464148
ATNVPFVLIATGT 12 13 x 14 2 28727 376025 110412
ATNVPFVLIATGT 13 13 x 13 1 163 1924 60
IILLGTFGCFATC 9 13 x 17 17 23654756 780894959 5895267
IILLGTFGCFATC 10 13 x 16 12 13616118 238134416 3773371
IILLGTFGCFATC 11 13 x 15 6 1877081 25080224 1647758
IILLGTFGCFATC 12 13 x 14 3 13368 174850 82497
IILLGTFGCFATC 13 13 x 13 3 179 2145 75
ASAWMLKLYAMFL 9 13 x 17 41 22784167 765632868 5554953
ASAWMLKLYAMFL 10 13 x 16 26 16268479 289953183 3915069
ASAWMLKLYAMFL 11 13 x 15 10 2019456 26936650 1817662
ASAWMLKLYAMFL 12 13 x 14 5 25193 329797 106322
ASAWMLKLYAMFL 13 13 x 13 2 225 2834 86
LVFLVELVAAIVG 9 13 x 17 16 22363083 758829344 5511472
LVFLVELVAAIVG 10 13 x 16 12 18769693 339359553 4493937
LVFLVELVAAIVG 11 13 x 15 7 2926366 39159055 2438355
LVFLVELVAAIVG 12 13 x 14 2 33186 435175 107123
LVFLVELVAAIVG 13 13 x 13 1 251 3133 69
VFRHEIKNSFKNN 9 13 x 17 20 27914822 963658280 6751226
VFRHEIKNSFKNN 10 13 x 16 14 17713835 335090899 4005942
VFRHEIKNSFKNN 11 13 x 15 11 2634511 35217585 1833897
VFRHEIKNSFKNN 12 13 x 14 3 21584 282867 71293
VFRHEIKNSFKNN 13 13 x 13 1 119 1339 83
MASPSRRLQTKPVI 13 14 x 15 3 27846 392294 97425
MASPSRRLQTKPVI 12 14 x 16 6 3668880 52736376 1706174
CFKSVLLIYTFIFW 13 14 x 15 0 20681 291256 88479

CFKSVLLIYTFIFW 12 14 x 16 5 2373934 34398056 947869
TGVILLAVGIWGKV 13 14 x 15 3 26294 370734 181770
TGVILLAVGIWGKV 12 14 x 16 6 3110778 44832788 1891285
LENYFSLLNEKATN 13 14 x 15 6 30706 432782 179169
LENYFSLLNEKATN 12 14 x 16 9 3297642 47428976 1574032
PFVLIATGTVIILL 13 14 x 15 2 28065 395864 126318
PFVLIATGTVIILL 12 14 x 16 5 3107630 44879016 1579834
TFGCFATCRASAWM 13 14 x 15 4 25187 356118 102554
TFGCFATCRASAWM 12 14 x 16 7 2550883 36828442 1490778
KLYAMFLTLVFLVE 13 14 x 15 2 25623 361158 172465
KLYAMFLTLVFLVE 12 14 x 16 5 2829815 40713470 1595049
VAAIVGFVFRHEIK 13 14 x 15 6 34420 485884 184132
VAAIVGFVFRHEIK 12 14 x 16 12 2397103 34489224 1265711
SFKNNYEKALKQYN 13 14 x 15 0 22865 321678 138587
SFKNNYEKALKQYN 12 14 x 16 1 2965029 42742224 1665541
MASPSRRLQTKPVIT 14 15 x 16 3 29299 442275 138379
MASPSRRLQTKPVIT 13 15 x 17 6 3719683 57379770 1893848
MASPSRRLQTKPVIT 12 15 x 18 9 24271792 546638490 5111573
MASPSRRLQTKPVIT 14 15 x 16 3 29299 442275 138712
MASPSRRLQTKPVIT 13 15 x 17 6 3719683 57379770 1689940
FKSVLLIYTFIFWIT 14 15 x 16 3 25125 379230 126822
FKSVLLIYTFIFWIT 13 15 x 17 6 2961877 45801300 1665766
VILLAVGIWGKVSLE 14 15 x 16 6 31835 480990 188278
VILLAVGIWGKVSLE 13 15 x 17 12 3619111 55729710 2176277
YFSLLNEKATNVPFV 14 15 x 16 3 29072 439110 147784
YFSLLNEKATNVPFV 13 15 x 17 9 3529929 54526185 1573808
IATGTVIILLGTFGC 14 15 x 16 2 34252 517530 163508
IATGTVIILLGTFGC 13 15 x 17 4 3405670 52716645 1761034
ATCRASAWMLKLYAM 14 15 x 16 4 30454 460710 68562
ATCRASAWMLKLYAM 13 15 x 17 10 2816028 43632975 1398288
LTLVFLVELVAAIVG 14 15 x 16 1 33904 512700 103292
LTLVFLVELVAAIVG 13 15 x 17 2 2373099 36644295 1325548
VFRHEIKNSFKNNYE 14 15 x 16 3 24565 371685 155089
VFRHEIKNSFKNNYE 13 15 x 17 9 2844224 43970970 1511976
ALKQYNSTGDYRSHA 14 15 x 16 2 29206 440850 174898
ALKQYNSTGDYRSHA 13 15 x 17 4 3924442 60233355 2051614
MASPSRRLQTKPVITC 15 16 x 17 3 30893 497488 144283
MASPSRRLQTKPVITC 14 16 x 18 6 3844508 63316144 1864483
MASPSRRLQTKPVITC 13 16 x 19 9 24315568 588325456 5344261
KSVLLIYTFIFWITGV 15 16 x 17 6 28014 451056 177254
KSVLLIYTFIFWITGV 14 16 x 18 9 3397703 55878400 2724093
KSVLLIYTFIFWITGV 13 16 x 19 12 27173043 622350800 7173091
LLAVGIWGKVSLENYF 15 16 x 17 6 19287 310704 143054
LLAVGIWGKVSLENYF 14 16 x 18 6 3138778 51363936 2316249
LLAVGIWGKVSLENYF 13 16 x 19 9 24065718 566423280 5749267
LLNEKATNVPFVLIAT 15 16 x 17 1 16325 262656 115815
LLNEKATNVPFVLIAT 14 16 x 18 7 2437486 40067808 1635917

```
LLNEKATNVPFVLIAT 13 16 x 19 14 19333665 453640704 4517261
TVIILLGTFGCFATCR 15 16 x 17 6 29904 482432 231766
TVIILLGTFGCFATCR 14 16 x 18 12 3314655 54684752 2581623
TVIILLGTFGCFATCR 13 16 x 19 15 17925122 415554432 4890096
SAWMLKLYAMFLTLVF 15 16 x 17 3 30762 495728 196578
SAWMLKLYAMFLTLVF 14 16 x 18 7 3036962 50165568 2100518
SAWMLKLYAMFLTLVF 13 16 x 19 13 19415239 454845936 4575836
VELVAAIVGFVFRHEI 15 16 x 17 6 36055 581600 264109
VELVAAIVGFVFRHEI 14 16 x 18 16 3527005 58098272 2644039
VELVAAIVGFVFRHEI 13 16 x 19 19 20981552 477206960 5191384
NSFKNNYEKALKQYNS 15 16 x 17 0 29502 476032 177352
NSFKNNYEKALKQYNS 14 16 x 18 0 3039832 50257920 1880065
NSFKNNYEKALKQYNS 13 16 x 19 2 20981809 491634848 4835016
GDYRSHAVDKIQNTLH 15 16 x 17 0 29625 476720 289287
GDYRSHAVDKIQNTLH 14 16 x 18 2 3825949 62846112 1890601
GDYRSHAVDKIQNTLH 13 16 x 19 2 20533157 520957280 4750196
```

SQL commands used

```
create table transient (
protein VARCHAR2 (20),
thresh number,
query number,
hits number,
nodesSeen number,
columnsDone number,
timeMs number
);

create table persistent (
protein VARCHAR2 (20),
thresh number,
query number,
hits number,
nodesSeen number,
columnsDone number,
timeMs number
);
====================

TOTAL RUN TIME IN HOURS

select SUM(timeMs)/(1000*60*60)
from persistent;
SUM(TIMEMS)/(1000*60*60)
------------------------
```

```
              136.407996   i.e 5.7 days

select SUM(timeMs)/(1000*60*60)
from transient;
SUM(TIMEMS)/(1000*60*60)
------------------------
              38.7503706   i.e. 1.5 days
-----------------------------------------------

set pages 40;
set lin 180;


------------------------------------------------------------
FIRST COLUMNS based on maxima

create view pCOLUMNS as select
p.query q,
p.thresh th,
min(p.columnsDone) mini,
max(p.columnsDone) maxi,
ROUND(avg(p.columnsDone)) aver,
ROUND(max(p.columnsDone)/200000000,4) maxDIV200M
from persistent p
group by p.query, p.thresh;

create view tCOLUMNS as select
p.query q,
p.thresh th,
min(p.columnsDone) mini,
max(p.columnsDone) maxi,
ROUND(avg(p.columnsDone)) aver,
ROUND(max(p.columnsDone)/200000000,4) maxDIV200M
from transient p
group by p.query, p.thresh;

select * from pcolumns;
select * from tcolumns;
-------------------------------------------------------
NOW HITS based on averages

create view pHITS as select
p.query q,
p.thresh th,
min(p.hits) mini,
max(p.hits) maxi,
ROUND(avg(p.hits)) aver,
```

```
ROUND(avg(p.hits)*(300/query)) avg\_300
from persistent p
group by p.query, p.thresh;

create view tHITS as select
p.query q,
p.thresh th,
min(p.hits) mini,
max(p.hits) maxi,
ROUND(avg(p.hits)) aver,
ROUND(avg(p.hits)*(300/query)) avg\_300
from transient p
group by p.query, p.thresh;


select * from thits;
select * from phits;
-------------------------------------------------------
NOW TIMES based on averages

create view pTIMES as select
p.query q,
p.thresh th,
min(p.timeMs) min,
max(p.timeMs) max,
ROUND(avg(p.timeMs)) avg,
ROUND(avg(p.timeMs)*(300/p.query)) avg\_300,
ROUND(avg(p.timeMs)/(p.query*1000*200),4) sec2Mb
from persistent p
group by p.query, p.thresh;

create view tTIMES as select
p.query q,
p.thresh th,
min(p.timeMs) min,
max(p.timeMs) max,
ROUND(avg(p.timeMs)) avg,
ROUND(avg(p.timeMs)*(300/p.query)) avg\_300,
ROUND(avg(p.timeMs)/(p.query*1000*200),4) sec2Mb
from transient p
group by p.query, p.thresh;

select * from ptimes;
select * from ttimes;
```

SQL output

```
SQL> select SUM(timeMs)/(1000*60*60)
  2  from persistent;

SUM(TIMEMS)/(1000*60*60)
------------------------
              136.407996

SQL> select SUM(timeMs)/(1000*60*60)
  2  from transient;

SUM(TIMEMS)/(1000*60*60)
------------------------
              38.7503706

SQL> set pages 40;
SQL> set lin 180;
SQL> select * from pcolumns;

         Q         TH       MINI       MAXI       AVER MAXDIV200M
---------- ---------- ---------- ---------- ---------- ----------
         5          3    6436675    9481975    8249977      .0474
         5          4      42950      51310      47598      .0003
         6          4    8956248   13943562   11641847      .0697
         6          5      62592      87888      77719      .0004
         7          5    9659566   16140761   14263059      .0807
         7          6      81907     122206     107137      .0006
         8          6   10457160   21760960   18001259      .1088
         8          7      76776     146472     124644      .0007
         9          6  190771353  283692348  246872682     1.4185
         9          7   13089069   25485219   19755152      .1274
         9          8      77760     192555     150067       .001
        10          7  204715200  337762230  283422668     1.6888
        10          8   15573770   32118120   23200178      .1606
        10          9     161180     243360     199979      .0012
        11          7  564852574  808146713  723471613     4.0407
        11          8  232200210  371462817  301077707     1.8573
        11          9   17369022   36604612   28906699       .183
        11         10     126247     268433     230633      .0013
        11         11        594       2123       1351          0
        12          9  197337984  409506576  336096871     2.0475
        12         10   28309116   40709484   34184200      .2035
        12         11     234804     326556     285869      .0016
        12         12        516       2964       1513          0
        13          9  758829344 1321559070  974136247     6.6078
        13         10  238134416  481226798  363482077     2.4061
```

```
         13         11   25080224   45218784   35699910      .2261
         13         12     174850     435175     316965      .0022
         13         13       1339       3133       1898         0
         14         12   34398056   52736376   42116508      .2637
         14         13     291256     485884     378641      .0024
         15         12  546638490  546638490  546638490     2.7332
         15         13   36644295   60233355   50801498      .3012
         15         14     371685     517530     448736      .0026
         16         13  415554432  622350800  510104411     3.1118
         16         14   40067808   63316144   54075435      .3166
         16         15     262656     581600     448268      .0029

36 rows selected.

SQL> select * from tcolumns;

         Q         TH       MINI       MAXI       AVER MAXDIV200M
---------- ---------- ---------- ---------- ---------- ----------
         5          4      21405      53535      42959      .0003
         5          5        115        695        384         0
         6          4    3181482    8960148    6045120      .0448
         6          5      30180      79446      62425      .0004
         6          6        138       1026        536         0
         7          4   33492781   90720238   60637158      .4536
         7          5    3640049   11597537    7548249       .058
         7          6      35371     119357      84012      .0006
         7          7        238       1358        678         0
         8          4  108495304  251024912  178571276     1.2551
         8          5   39192752  102245968   75437310      .5112
         8          6    2922120   13570984    9841472      .0679
         8          7      74808     139040     108745      .0007
         9          4  191805228  475824366  343272622     2.3791
         9          5   95033853  276993117  193006027      1.385
         9          6   41098536  105980760   78171406      .5299
         9          7    6615036   15241887   10962888      .0762
         9          8      83538     185526     128722      .0009
         9          9        315       2133        957         0
        10          4  167366310  769337230  562480802     3.8467
        10          5  124155670  540604130  371848620      2.703
        10          6   69759770  303568470  210464641     1.5178
        10          7   34430920  124350020   89022147      .6218
        10          8    5280720   19574720   12701967      .0979
        10          9      55320     222570     160751      .0011
        11          3  883686199 1506901275 1228614912     7.5345
        11          4  588209424 1282432382  927304596     6.4122
        11          5  337938843  889693266  643200278     4.4485
        11          6  252268302  577024657  419560898     2.8851
```

```
        11          7  168544189  312397712  242837030      1.562
        11          8   79405403  137446474  114782868      .6872
        11          9   15113879   20830898   17682355      .1042
        11         10     193424     214797     207629      .0011

33 rows selected.

SQL> select * from thits;

         Q         TH       MINI       MAXI       AVER    AVG\_300
---------- ---------- ---------- ---------- ---------- ----------
         5          4         28       3403        896       53759
         5          5          0        151         24        1465
         6          4        167       6647       2048      102398
         6          5          0        239         53        2642
         6          6          0         11          2         101
         7          4        420      12600       3888      166649
         7          5          7        688        143        6141
         7          6          0         50          7         288
         7          7          0          5          1          47
         8          4        532      22973       6035      226300
         8          5         18       1481        244        9165
         8          6          0        119         12         449
         8          7          0         12          2          73
         9          4       1732      32131       9354      311799
         9          5         16       2716        408       13584
         9          6          0        212         20         666
         9          7          0         26          3         109
         9          8          0          6          1          46
         9          9          0          2          0          16
        10          4       1446      43000      13488      404628
        10          5         40       2744        633       18982
        10          6          0        277         33         990
        10          7          0         27          5         143
        10          8          0         10          2          67
        10          9          0          4          1          33
        11          3     279711     732887     516487    14086021
        11          4      13353      44715      27134      740014
        11          5        542       2506       1377       37555
        11          6         30        116         62        1677
        11          7          9         31         15         400
        11          8          6         16         10         268
        11          9          4          8          6         164
        11         10          1          4          3          86

33 rows selected.
```

```
SQL> select * from phits;

         Q         TH       MINI       MAXI       AVER    AVG\_300
---------- ---------- ---------- ---------- ---------- ----------
         5          3      86871     551218     260176   15610540
         5          4        218      17526       7449     446953
         6          4       1444      31015      13585     679267
         6          5          9        739        363      18150
         7          5         64       3798       1232      52781
         7          6          5        169         40       1719
         8          6         16        108         56       2088
         8          7          0         19          7        258
         9          6         26        112         63       2089
         9          7          4         47         12        406
         9          8          2          9          4        136
        10          7          7         61         23        687
        10          8          3         18          9        273
        10          9          1         12          5        147
        11          7          7         31         18        491
        11          8          4         14          9        245
        11          9          2         10          6        164
        11         10          0          6          2         66
        11         11          0          3          1         36
        12          9          6         33         17        436
        12         10          2         22         10        244
        12         11          1         11          5        128
        12         12          0          3          2         44
        13          9         12         41         21        490
        13         10          9         26         15        341
        13         11          3         14          8        185
        13         12          1          9          4         87
        13         13          1          3          2         46
        14         12          1         12          6        133
        14         13          0          6          3         62
        15         12          9          9          9        180
        15         13          2         12          7        136
        15         14          1          6          3         60
        16         13          2         19         11        198
        16         14          0         16          7        135
        16         15          0          6          3         65

36 rows selected.

SQL> select * from ptimes;

         Q         TH        MIN        MAX        AVG    AVG\_300     SEC2MB
---------- ---------- ---------- ---------- ---------- ---------- ----------
```

```
         5          3    1450560    2583183    1853556  111213340     1.8536
         5          4       4151      33372      18650    1119000      .0187
         6          4     517083    2070834    1541020   77051017     1.2842
         6          5       1685      67656      21380    1069000      .0178
         7          5    1139873    1701941    1393505   59721652      .9954
         7          6       2052      63256      25098    1075629      .0179
         8          6     660330    2266510    1563657   58637154      .9773
         8          7       1798      42883      16450     616887      .0103
         9          6    3362900    4306590    3874534  129151122     2.1525
         9          7    1201663    2016994    1551996   51733192      .8622
         9          8       3047      59940      24716     823856      .0137
        10          7    3194328    5986231    4574149  137224470     2.2871
        10          8    1140265    2570789    1957363   58720897      .9787
        10          9      44954     185122      94972    2849157      .0475
        11          7    4675691    6276299    5629039  153519245     2.5587
        11          8    3346721    4975578    4300113  117275800     1.9546
        11          9    1267862    2629181    1768264   48225389      .8038
        11         10       3078     185881      49093    1338902      .0223
        11         11          2        126         58       1594         0
        12          9    2379287    5506253    4480121  112003033     1.8667
        12         10    1312396    2793171    2047009   51175214      .8529
        12         11       4866     185079      84540    2113500      .0352
        12         12          5        302         90       2253         0
        13          9    5511472   10047181    6975204  160966254     2.6828
        13         10    3773371    6913588    4742562  109443736     1.8241
        13         11    1472837    2795986    2058922   47513587      .7919
        13         12      71293     150758     104202    2404672      .0401
        13         13         49        103         72       1667         0
        14         12     947869    1891285    1524030   32657793      .5443
        14         13      88479     184132     141211    3025950      .0504
        15         12    5111573    5111573    5111573  102231460     1.7039
        15         13    1325548    2176277    1704810   34096198      .5683
        15         14      68562     188278     140532    2810648      .0468
        16         13    4517261    7173091    5225156   97971683     1.6329
        16         14    1635917    2724093    2181954   40911642      .6819
        16         15     115815     289287     193278    3623954      .0604

36 rows selected.

SQL> select * from ttimes;

         Q         TH        MIN        MAX        AVG    AVG_\300     SEC2MB
---------- ---------- ---------- ---------- ---------- ---------- ----------
         5          4         42        175         89       5346      .0001
         5          5          1         19          2        117         0
         6          4       5732      16116      10869     543447      .0091
         6          5         54        143        111       5537      .0001
```

```
         6          6          0          2          1         65          0
         7          4      47112     124548      83872    3594513      .0599
         7          5       5996      19783      12734     545756      .0091
         7          6         60        197        141       6036      .0001
         7          7          1          3          2         70          0
         8          4     121255     283049     200358    7513423      .1252
         8          5      51330     134168      98365    3688685      .0615
         8          6       4439      21566      15459     579707      .0097
         8          7        117        214        168       6309      .0001
         9          4     196316     523115     354320   11810657      .1968
         9          5     103649     308100     211009    7033630      .1172
         9          6      52051     134134      98294    3276467      .0546
         9          7       9759      22828      16402     546735      .0091
         9          8        121        272        189       6307      .0001
         9          9          1          5          2         77          0
        10          4     163226     835883     563602   16908063      .2818
        10          5     124304     557265     379482   11384457      .1897
        10          6      73123     356037     227065    6811953      .1135
        10          7      41526     170960     108653    3259584      .0543
        10          8       7385      27516      17774     533213      .0089
        10          9         95        317        230       6885      .0001
        11          3     825973    1412062    1151716   31410444      .5235
        11          4     555693    1211841     874742   23856595      .3976
        11          5     325368     855454     617572   16842868      .2807
        11          6     248479     569079     413156   11267891      .1878
        11          7     173910     323956     250632    6835405      .1139
        11          8      92343     158147     132290    3607914      .0601
        11          9      22927      28407      24405     665582      .0111
        11         10        261        392        297       8100      .0001

33 rows selected.

SQL> spool off
```

Bibliography

[1] ISO/IEC 8824-1:1995. Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation, 1995. http://www.iso.ch/cate/d16289.html.

[2] P. Ferragina A. Crauser. On constructing suffix arrays in external memory. In *ESA'99. 7th Annual European Symposium. Proceedings*, pages 224–235, 1999.

[3] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[4] M.D. Adams et al. The Genome Sequence of Drosophila melanogaster. *Science*, 287:2185–2195, 2000.

[5] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[6] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. In *SOFSEM'99*, LNCS 1725, pages 291–306, 1999.

[7] S.F. Altschul et al. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–10, 1990.

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

[9] A. Marian amd S. Abiteboul, G. Cobena, and L. Mignet. Change-Centric Management of Versions in an XML Warehouse. In *Proc. 27th Conf. on Very Large Databases*, pages 581–590. Morgan Kaufmann, 2001.

[10] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[11] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software Practice and Experience*, 25(2):129–141, 1995.

[12] S. Andrews. Design and Implementation of a Relational Database and Graphical User Interface to Store Microarray Data. Master's thesis, Department of Computing Science, University of Glasgow, 2001.

[13] M. Annamalai et al. Indexing Images in Oracle8i. In *SIGMOD 2000*, pages 539–547, 2000.

[14] A. Apostolico and Z. Galil, editors. *Pattern Matching Algorithms*. OUP, 1997.

[15] Artemis, 1999. http://www.sanger.ac.uk/Software/Artemis/index.shtml.

[16] M. Ashburner et al. Gene ontology: tool for the unification of biology. *Nat Genet*, 25:25–9, 2000.

[17] M. Atkinson and M. Jordan. Providing Orthogonal Persistence for Java. *Lecture Notes in Computer Science*, 1445, 1998.

[18] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[19] M. P. Atkinson, K. Chisholm, and P. Cockshott. PS-algol: An algol with a persistent heap. In *ACM SIGPLAN Notices*, volume 17, July 1982.

[20] M.P. Atkinson. Persistence and Java — a Balancing Act. In *Proceedings of the ECOOP Symposium on Objects and Databases, Sophia Antipolis, France June 2000*, number 1944 in Lecture Notes in Computing Science, pages 1–32. Springer-Verlag, 2000.

[21] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM Sigmod Record*, 25(4), 1996.

[22] M.P. Atkinson and M.J. Jordan. Issues Raised by Three Years of Developing PJama. In *Database Theory - ICDT'99*, volume 1540 of *Lecture Notes in Computer Science*, 1999.

[23] M.P. Atkinson and M.J. Jordan. A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Technical Report TR-2000-90, Sun Microsystems Laboratories Inc and Department of Computing Science, University of Glasgow, 901 San Antonio Road, Palo Alto, CA 94303, USA and Glasgow G12 8QQ, Scotland, 2000.

[24] M.P. Atkinson and R. Welland, editors. *Fully Integrated Data Environments*. Springer-Verlag, 1999.

[25] R. Baeza-Yates and G. Navarro. A Hybrid Indexing Method for Approximate String Matching. *Journal of Discrete Algorithms, JDA*, 1:205–239, 2001.

[26] R. Baeza-Yates, G. Navarro, E. Sutinen, and J. Tarhio. Indexing Methods for Approximate Text Retrieval. Technical report, University of Chile, 1997.

[27] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, Reading, US, 1999.

[28] R. A. Baeza-Yates. Text retrieval: Theory and practice. In Jan van Leeuwen, editor, *Proceedings of the IFIP 12th World Computer Congress. Volume 1: Algorithms, Software, Architecture*, pages 465–476, Amsterdam, The Netherlands, September 1992. Elsevier Science Publishers.

[29] R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[30] R.A. Baeza-Yates and G.H. Gonnet. All-against-all sequence matching. Technical report, Dept. of Computer Science, Universidad de Chile, 1990. ftp://sunsite.dcc.uchile.cl/pub/users/rbaeza/papers/all-all.ps.gz.

[31] P.G. Baker, C.A. Goble, S. Bechhofer, N.W. Paton, R. Stevens, and A. Brass. An ontology for bioinformatics applications. *Bioinformatics*, 15:510–520, 1999.

[32] J. B.L. Bard, R. A. Baldock, and D. R. Davidson. Elucidating the Genetic Networks of Development: A Bioinformatics Approach. *Genome Research*, 8:859–863, 1998. http://genex.hgu.mrc.ac.uk/.

[33] P. Bieganski. *Genetic Sequence Data Retrieval and Manipulation based on Generalised Suffix Trees*. PhD thesis, University of Minnesota, USA, 1995.

[34] M. Bishop, editor. *Guide to Human Genome Computing*. Academic Press Inc., 1998.

[35] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas. The Smallest Automaton Recognizing the Subwords of a Text. *TCS*, 40:31–55, 1985.

[36] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM, ; ACM CR 8810-0785*, 34(3), July 1987.

[37] A. Brazma, I. Jonassen, J. Vilo, and E. Ukkonen. Predicting Gene Regulatory Elements in Silico on a Genomic Scale. *Genome Research*, 8:1202–1215, 1998.

[38] A. Brazma, A. Robinson, G. Cameron, and M. Ashburner. One-stop shop for microarray data. *Nature*, 403:699–700, 2000.

[39] S.E. Brenner, C. Chothia, and T.J.P. Hubbard. Assessing sequence comparison methods with reliable structurally identified distant evolutionary relationships. *PNAS*, 95:6073–6078, 1998.

[40] A. L. Brown, R. Morrison, D. S. Munro, A. Dearle, and J. Rosenberg. A Layered Persistent Architecture for Napier88. In J. Rosenberg and J. L. Keedy, editors, *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, Workshops in Computing, pages 155–172, London, May 8–11 1990. Springer Verlag.

[41] P. Buneman, S. B. Davidson, K. Hart, G. C. Overton, and L. Wong. A data transformation system for biological data sources. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases*, pages 158–169, Zurich, Switzerland, 11–15 September 1995. Morgan Kaufmann.

[42] P. Buneman, S.B. Davidson, and A. Watters. A semantics for complex objects and approximate queries. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 305–314, Austin, Texas, 21–23 March 1988.

[43] P. Buneman, S. Khanna, and W. C. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT 2001*, LNCS 1973, pages 316–330, 2001.

[44] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram Based Database Searching Using a Suffix Array. In S. Istrail, P. Pevzner, and M. Waterman, editors, *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 77–83, Lyon, France, 1999. ACM Press.

[45] M. J. Carey, D. J. Dewitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):383–394, June 1994.

[46] L. Carr, W. Hall, S. Bechhofer, and C.A. Goble. Conceptual linking: ontology-based open hypermedia. In *WWW 2001*, pages 334–342, 2001.

[47] R. G. G. Cattell and D. K. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, San Francisco, 1997.

[48] C. Charras and T. Lecroq. Exact String Matching Algorithms. http://www-igm.univ-mlv.fr/ lecroq/string/.

[49] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the 7th ACM PODS*, pages 34 – 43, 1998.

[50] I. Chen, A. Kosky, V. Markowitz, and E. Szeto. Developing and accessing scientific databases with the OPM data management tools. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 580–580, Washington - Brussels - Tokyo, April 1997. IEEE.

[51] I. A. Chen and V. M. Markowitz. An Overview of the Object-Protocol Model (OPM) and OPM data management tools. Technical Report LBL-33706, Lawrence Berkeley national Laboratory, 1995.

[52] J. M. Cheng, N. M. Mattos, D. D. Chamberlin, and L. G. DeMichiel. Extending relational database technology for new applications. *IBM Systems Journal*, 33(2):264–279, 1994. G321-5542.

[53] I. Chumakov et al. Continuum of overlapping clones spanning the entire human chromosome 21q. *Nature*, 359:380–386, 1992.

[54] D. R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[55] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, Atlanta, Georgia, 28–30 January 1996.

[56] A. L. Cobbs. Fast approximate matching using suffix trees. In Zvi Galil and E. Ukkonen, editors, *Combinatorial Pattern Matching, 6th Annual Symposium*, volume 937 of *Lecture Notes in Computer Science*, pages 41–54, Espoo, Finland, 5-7 July 1995. Springer.

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

[58] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proc. 27th Conf. on Very Large Databases*, pages 341–350. Morgan Kaufmann, 2001.

[59] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[60] S.D. Cox. A PJama Implementation of Efficient DNA or Protein Sequence Storage and Retrieval. Master's thesis, Department of Computing Science, University of Glasgow, 1999.

[61] S. Davidson, J. Crabtree, B. Brunk, J. Schug, V. Tannen, C. Overton, and C. Stoeckert. K2/kleisli and gus: Experiments in integrated access to genomic data sources. *IBM Systems Journal, March 2001*, 2001. http://http://db.cis.upenn.edu.

[62] S. Davidson and A. Kosky. WOL: A language for Database Transformations and Constraints. In *Proc. 13th Int. Conf. Data Eng. (ICDE'97)*, pages 55–66, Washington - Brussels - Tokyo, 1997. IEEE.

[63] S. Davidson, G. C. Overton, V. Tannen, and L. Wong. BioKleisli: a digital library for biomedical researchers. *Int. Journal on Digital Libraries*, 1:36–53, 1997.

[64] M. Dayhoff, R. Eck, and C. Park. A model of evolutionary change in proteins. In M. Dayhoff, editor, *Atlas of Protein Sequence and Structure*, volume 5. Silver Springs, MD: National Biomedical Research Foundation, 1972.

[65] J.M. Delabar, N. Creau, P.M. Sinet, O. Ritter, S.E. Antonarakis, M. Burmeister, A. Chakravarti, D. Nizetic, M. Ohki, and D. Patterson. Report of the fourth international workshop on human chromosome 21. *Genomics*, 18:735–745, 1993.

[66] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research*, 27:2369–2376, 1999.

[67] B. Dorohonceanu and C. G. Nevill-Manning. Accelerating protein classification using suffix trees. In *Proc. Intl. Conf. on Intelligent Systems for Molecular Biology ISMB00*, pages 128–133, 2000.

[68] E. Dunning. *Finding Structure In Text, Genome And Other Symbolic Sequences*. PhD thesis, Department of Cmputer Science, University of Sheffield, 1998.

[69] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis. Probabilistic models of proteins and nucleic acids.* CUP, 1998.

[70] R. Durbin and J. T. Mieg. A C. elegans Database, 1991. http://www.acedb.org.

[71] B. Eckel. *Thinking in Java*. Prentice-Hall PTR, 1998.

[72] M.B. Eisen, n P.T. Spellma, P.O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proc Natl Acad Sci U S A*, 95:14863–8, 1998.

[73] T. Eki, M. Abe, K. Furuya, I. Ahmad, N. Fujishima, H. Kishida, A. Shiratori, T. Onozaki, K. Yokoyama, D. Le Paslier, D. Cohen, F. Hanaoka, and Y. Murakami. A long-range physical map of human Chromosome 21q22.1 band from the YAC continuum. *Mammalian Genome*, 7:303–311, 1996.

[74] A.J. Enright and C.A. Ouzounis. Functional associations of proteins in entire genomes by means of exhaustive detection of gene fusion. *Genome Biology*, 2(9), 2001. http://genomebiology.com/2001/2/9/research/0034.

[75] D. Ensor and I. Stevenson. *Oracle8 Design Tips*. O'Reilly & Associates, Inc., 1997.

[76] T. Etzold, A. Ulyanov, and P. Argos. SRS: Information retrieval system for molecular biology data banks. *Methods in Enzymology*, 266:114–128, 1996.

[77] B. Ewing and P. Green. Base-calling of automated sequencer traces using Phred. II. Error probabilities. *Genome Res*, 8(3):186–94, 1998.

[78] B. Ewing, L. Hillier, M.C. Wendl, and P. Green. Base-calling of automated sequencer traces using Phred. I. Accuracy assessment. *Genome Res*, 8(3):175–85, 1998.

[79] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottlenect in suffix tree construction. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[80] L. Fernando, B. Seibel, and S. Lifschitz. A Genome Databases Framework. In *DEXA 2001*, pages 319–329, 2001.

[81] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, March 1999.

[82] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In IEEE, editor, *41st Annual Symposium on Foundations of Computer Science: proceedings: 12–14 November, 2000, Redondo Beach, California*, pages 390–398, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. IEEE Computer Society Press.

[83] R.D. Fleischmann, M.D. Adams, O. White, R.A. Clayton, E.F. Kirkness, A.R. Kerlavage, C.J. Bult, J.F. Tomb, B.A. Dougherty, J.M. Merrick, K. McKenney, G. Sutton, W. FitzHugh, C. Fields, J. D. Gocayne, J. Scott, R. Shirley, L.-I. Liu, A. Glodek, J. M. Kelley, J. F. Weidman, C. A. Phillips, T. Spriggs, E. Hedblom, M. D. Cotton, T. R. Utterback, M. C. Hanna, D. T. Nguyen, D. M. Saudek, R. C. Brandon, L. D. Fine, J. L. Fritchman, J. L. Fuhrmann, N. S. M. Geoghagen, C. L. Gnehm, L. A. McDonald, K. V. Small, C. M. Fraser, H. O. Smith, and J. C. Venter. Whole-genome random sequencing and assembly of Haemophilus influenzae Rd. *Science*, 269(5223):496–512, 1995.

[84] L. Florea, G. Hartzell, Z. Zhang, G. Rubin, and W. Miller. A computer program for aligning a cdna sequence with a genomic dna sequence. *Genome Research*, 8:967–974, 1998.

[85] I. Foster and C. Kesselman (eds.). *The Grid: Blueprint for a new Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[86] William B. Frakes and R. Baeza-Yates. *Information Retrieval Data Structures and Algorithms.* Prentice Hall, 1992.

[87] H. Fraser. Genome Annotation and Comparison System, M.Res. Dissertation, University of Glasgow, Institute of Biomedical and Life Sciences and Department of Computing Science, 2000.

[88] K. Gardiner, S. Graw, H. Ichikawa, M. Ohki, A. Joetham, P. Gervy, I. Chumakov, and D. Patterson. Yac analysis and minimal tiling path construction for chromosome 21q. *Somat. Cell Mol. Genet.*, pages 399–414, 1995.

[89] S. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: a unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[90] D. Gilbert. Readseq sequence conversion program, 1999. http://bimas.cit.nih.gov/-molbio/readseq/.

[91] R. Goldman and J. Widom. Approximate DataGuides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, 1999.

[92] G. H. Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley, New York, 1 edition, 1984.

[93] J. Gosling, B. Joy, G. Steele, and G. Brancha. *The Java Language Specification*. Addison-Wesley, 2000.

[94] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[95] E. Grant. A Microarray Database. Master's thesis, Department of Computing Science, University of Glasgow, 2001.

[96] S.L. Graw, K. Gardiner, K. Halljohnson, I. Hart, A. Joetham, K. Walton, D. Donaldson, and D. Patterson. Molecular analysis and breakpoint definition of a set of human chromosome 21 somatic cell hybrids. *Somat. Cell Mol. Genet.*, pages 415–428, 1995.

[97] A. Grigoriev, A. Levin, and H. Lehrach. A distributed environment for physical map construction. *Bioinformatics*, 14(3):252–258, 1998.

[98] J. Groet et al. Bacterial contig map of the 21q11 region associated with Alzheimer's disease and abnormal myelopoiesis in Down syndrome. *Genome Research*, 8:385–98, 1998.

[99] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[100] C. G. Hamilton. Recovery Management for Sphere: Recovering A Persistent Object Store. Technical Report TR-1999-51, University of Glasgow, Department of Computing Science, December 1999.

[101] N.J. Harding. Software to search a protein sequence database and automatically update a local cache of data for a biological e-laboratory. Master's thesis, Department of Computing Science, University of Glasgow, 2001.

[102] E. Harley, A.J. Bonner, and N. Goodman. Revealing hidden interval graph structure in STS-content data. *Bioinformatics*, 15(4):278–288, 1999.

[103] M. Hattori et al. The DNA sequence of human chromosome 21. *Nature Genetics*, 405:311–319, 2000.

[104] S. Henikoff and J.G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc Natl Acad Sci U S A*, 89(22):10915–9, 1992.

[105] T. Hildmann et al. A contiguous 3-Mb sequence-ready map in the S3-MX region on 21q22.2 based on high-throughput nonisotopic library screenings. *Genome Research*, 9:360–72, 1999.

[106] D.W. Hood, M.E. Deadman, T. Allen, H. Masoud, M. A Masoud, J.R. Brisson, R. Fleischmann, J.C. Venter, J.C. Richards, and E.R. Moxon. Use of the complete genome sequence information of Haemophilus influenzae strain Rd to investigate lipopolysaccharide biosynthesis. *Mol Microbiol*, 22(5):951–65, 1996.

[107] A.L. Hosking and J. Chen. PM3: An Orthogonally Persistent Systems Programming Language - Design, Implementation, Performance. *Proceedings VLDB99*, 1999.

[108] R.A. Hoskins et al. A BAC-based physical map of the major autosomes of Drosophila melanogaster. *Science*, 287:2271–4, 2000.

[109] X. Huang and W. Miller. A Time-Efficient, Linear-Space Local Similarity Algorithm. *Advances in Applied Mathematics*, 12:337–357, 1991.

[110] D.A. Huffman. A method for the construction of minimim redundancy codes. *Proc.of the IRE*, 40, September 1952.

[111] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative Workflows that Support Easy Modification and Dynamic Browsing. In *International Joint Conference on Work Activities Coordination and Collaboration (WACC) 1999*, pages 69–78, 1999. http://www-db.research.bell-labs.com/project/ vortex/index.html.

[112] K. Humphreys, G. Demetriou, and R. Gaizauskas. Two Applications of Information Extraction to Biological Science Journal Articles: Enzyme Interactions and Protein Structures. In *Proceedings of the Pacific Symposium on Biocomputing (PSB-2000), Honolulu, Hawaii, USA, January 4-9*, pages 505–516, 2000.

[113] E. Hunt. PJama Stores and Suffix Tree Indexing for Bioinformatics Applications, 2000. 10th PhD Workshop at ECOOP'00, http://www.inf.elte.hu/∼phdws/-timetable.html.

[114] E. Hunt and M. Atkinson. Design and Implementation of a Genetics Database using Java and Orthogonal Persistence, August 3-4 1998. Poster at Objects in Bioinformatics, Cambridge, Hinxton, 1998.

[115] E. Hunt and M. Atkinson. Design and Implementation of a Genetics Database using Java and Orthogonal Persistence, June25-26 1998. Poster at the BBSCRC Workshop: Technologies for Functional Genomics, Warwick 1998.

[116] E. Hunt and M. Atkinson. PJama: Databases of Indexed Sequence and Mapping Data, April26-28 2000. Poster at Genes, Proteins and Computers Conference, Chester College of Higher Education.

[117] E. Hunt, M. Atkinson, and R. Irving. Indexing the whole genome, 2001. Poster and oral presentation at Workshop 9 (Genome Informatics) at Human Genome Meeting 2001 (HGM2001), April 19-22, 2001, Edinburgh, http://dcs.gla.ac.uk/ẽla/HGM-abstract.html.

[118] E. Hunt, M. Atkinson, R.W. Irving, I. Darroch, and D. Leader. Visual data exploration and editing using Java, 1999. Poster at the Conference on Datamining in Bioinformatics, Towards *in silico* Biology, Hinxton, Cambridge, 10-12 November 1999.

[119] E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *Proc. 27th Conf. on Very Large Databases*, pages 139–148. Morgan Kaufmann, 2001.

[120] E. Hunt, R. W. Irving, and M. Atkinson. Persistent Suffix Trees and Suffix Binary Search Trees as DNA Sequence Indexes. Technical report, University of Glasgow, Department of Computing Science, October 2000. TR-2000-63, http://www.dcs.gla.ac.uk/∼ela.

[121] E. Hunt, H. Lehrach, and M-L. Yaspo. Physical map integration using a relational database: the example of the Human Chromosome 21 DB, 2000. Unpublished manuscript, reproduced in Appendix A.

[122] E. Hunt, M-L. Yaspo, I. Szulzedsky, M. Nguyen, X. Kong, J. O'Brian, and H. Lehrach. Creating an integrated Chromosome 21 map using ACEDB and ORACLE, 1997. Poster, Chromosome 21 Workshop, Berlin, 1997.

[123] H. Ichikawa, F. Hosoda, Y. Arai, K. Shimizu, M. Ohira, and M. Ohki. A noti restriction map of the entire long arm of human chromosome 21. *Nature Genetics*, 4:361–366, 1993.

[124] Y. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*, pages 174–185, San Francisco, September 1999. Morgan Kaufmann.

[125] R.W. Irving and L. Love. The Suffix Binary Search Tree and Suffix AVL Tree. Technical Report TR-2000-54, University of Glasgow, Department of Computing Science, 2000. http://www.dcs.gla.ac.uk/research/algorithms/sbst/publications/-SBST_report.ps.

[126] R.W. Irving and L. Love. Suffix Binary Search Trees and Suffix Arrays. Technical Report TR-2001-82, University of Glasgow, Department of Computing Science, 2001. http://www.dcs.gla.ac.uk/research/algorithms/sbst/publications/SA_report.ps.

[127] H. V. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 403–414, 2000.

[128] R. Japp. First Year Report, July 2001. Department of Computing Science, University of Glasgow.

[129] T.K. Jenssen, A. Laegreid, J. Komorowski, and E. Hovig. A literature network of human genes for high-throughput analysis of gene expression. *Nat Genet.*, 28(1):21–28, 2001.

[130] A. Jones. A Database for Storing the Results of 2D-PAGE Experiments. Master's thesis, Department of Computing Science, University of Glasgow, 2001.

[131] T. Kahveci and A.K. Singh. An Efficient Index Structure for String Databases. In *VLDB'01*, pages 351–360. Morgan and Kaufmann, 2001.

[132] R. Kanth. Indexing Medium-dimensionality Data in Oracle. In *SIGMOD 1999*, pages 521–522, 1999.

[133] W. J. Kent and D. Haussler. GigAssembler: An Algorithm for the Initial Assembly of the Human Genome Working Draft. Technical report, Department of Biology and Howard Hughes Medical Institute, Department of Computer Science, University of California at Santa Cruz, Santa Cruz, CA 95064 USA, 2000. Technical Report UCSC-CRL-00-17, December 27, 2000, http://genome.ucsc.edu/goldenPath/algo.html.

[134] M. Knox. Genetic Map Browser. Master's thesis, Department of Computing Science, University of Glasgow, 1999.

[135] D. Knuth. *The Art of Computer Programming, volume 3, Sorting and Searching*. Addison-Wesley, 1973.

[136] E.V. Koonin, R.L. Tatusov, and M.Y. Galperin. Beyond complete genomes: from sequence to structure and function. *Curr Opin Struct Biol*, pages 355–63, 1998. http://www.ncbi.nlm.nih.gov/COG/.

[137] R.N. Kostoff and R.A. DeMarco. Extracting information from the literature by text mining. *Anal Chem*, 73(13):370A–378A, 2001.

[138] S. Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29:1149–1171, 1999.

[139] S. Kurtz and C. Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, pages 426–427, 1999.

[140] N. J. Larsson. *Structures of String Matching and Data Compression.* PhD thesis, Department of Computer Science, Lund University, 1999.

[141] C. Letondal. Pise, a tool to generate Web interfaces for Molecular Biology programs. http://www-alt.pasteur.fr/∼letondal/Pise/.

[142] V. I. Levenstein. Binary codes capable of correcting insertions and reversals. *Sov. Phys. Dokl.*, 10:707–10, 1966.

[143] B. Lewis, B. Mathiske, and N. Gafter. Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine. In *Proceedings of the Nineth International Workshop on Persistent Object Systems*, LNCS 2135, pages 19–34. Springer-Verlag, September 2000.

[144] K. Loney and G. Koch. *Oracle 8i: The Complete Reference*. Osborne McGraw-Hill, 2000.

[145] A. Louis, E. Ollivier, J-C. Aude, and J-L. Risler. Massive Sequence Comparisons as a Help in Annotating Genomic Sequences. *Genome Research*, 11, 2001. June 12th 2001, published online in advance of paper publication.

[146] N.M. Luscombe, D. Greenbaum, and M. Gerstein. What is bioinformatics? A proposed definition and overview of the field. *Methods Inf Med*, 40(4):346–58, 2001.

[147] M.G. Maass. Linear Bidirectional On-Line Construction of Affix Trees. In R. Giancarlo and Sankoff D., editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching, CPM 2000*, LNCS 1848, pages 320–334. Springer-Verlag, 2000.

[148] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, October 1993.

[149] L. Marsan and M-F. Sagot. Extracting structured motifs using a suffix tree – Algorithms and application to promoter consensus identification. In *Proceedings of the fourth annual international conference on Computational molecular biology RECOMB00*, 2000. to appear.

[150] G.T. Marth, I. Korf, M.D. Yandell, R.T. Yeh, Z. Gu, H. Zakeri, N.O. Stitziel, L. Hillier, P.Y. Kwok, and W.R. Gish. A general approach to single-nucleotide polymorphism discovery. *Nat Genet*, 23(4):452–6, 1999.

[151] S.M. Maurer, R.B. Firestone, and C.R. Scriver. Science's neglected legacy. *Nature*, 405:117–120, 2000.

[152] R. McCool. *The Common Gateway Interface*. NCSA, 1.1 edition, 1994.

[153] E.M. McCreight. A space-economic suffix tree construction algorithm. *Journal of the A.C.M.*, 23(2):262–272, April 1976.

[154] R. McNaughton. *Elementary Computability, Formal Languages, and Automata*. Prentice-Hall, 1982.

[155] G.P. McSorley. Design and Implementation of Improvements to the Sphere System, October 2000. manuscript.

[156] H.W. Mewes and K. Heumann. Genome Analysis: Pattern Search in Biological Macromolecules. In *LNCS*, volume 937, pages 261–285, 1995.

[157] C. Miller, J. Gurd, and A. Brass. A RAPID algorithm for sequence database comparisons: application to the identification of vector contamination in the EMBL databases. *Bioinformatics*, 15:111–121, 1999.

[158] W. Miller. Comparison of genomic dna sequences: solved and unsolved problems. *Bioinformatics*, 17(5):391–397, 2001.

[159] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proc. 24th VLDB Conf.*, pages 122–133, 1998.

[160] K. Monostori, A. Zaslavsky, and H. Schmidt. Suffix Vector: Space- and Time-Efficient Alternative to Suffix Trees. In *Proceedings of the Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, 2002. to appear.

[161] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Jrnl. A.C.M.*, 15(4):514–534, October 1968.

[162] D. W. Mount. *Bioinformatics. Sequence and Genome Analysis.* Cold Spring Harbor Laboratory Press, 2001.

[163] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th Annual Symposium on Foundations of Computer Science: October 20–22, 1997, Miami Beach, Florida*, pages 118–126. IEEE Computer Society Press, 1997.

[164] J. I. Munro, V. Raman, and S.S. Rao. Space Efficient Suffix Trees. *Journal of Algorithms*, 39:205–222, 2001.

[165] E.W. Myers et al. A whole-genome assembly of drosophila. *Science*, 287:2196–2204, 2000.

[166] http://www.mysql.com/.

[167] P. Nadkarni. Mapmerge: merge genomic maps. *Bioinformatics*, 14(4):310–316, 1998.

[168] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33(1):31–88, 2000.

[169] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electonic Journal*, 1(2), 1998.

[170] G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. In M. Crochemore and M. Paterson, editors, *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching*, number 1645 in Lecture Notes in Computer Science, pages 163–185, Warwick University, UK, 1999. Springer-Verlag, Berlin.

[171] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing Text with Approximate q-grams. In Giancarlo R. and Sankoff D., editors, *Combinatorial Pattern Matching 2000, 11th Annual Symposium*, LNCS 1848, pages 350–365. Springer, 2000.

[172] D. Nizetic, L. Gellen, R.M.J. Hamvas, R. Mott, A. Grigoriev, R. Vatcheva, G. Zehetner, M-L. Yaspo, A. Dutriaux, C. Lopes, J. Delabar, C. Van Broeckhoven, M.C. Potier, and H. Lehrach. An integrated yac-overlap and cosmid-pocket map of the human chromosome 21. *Hum. Mol. Genet.*, 3:759–770, 1994.

[173] Object Protocol Method. http://gizmo.lbl.gov/opm.html.

[174] I. Ounis and M. Pasca. Modeling, indexing and retrieving images using conceptual graphs. In *Proceedings of the 9th DEXA International Conference on Database and EXpert Systems Applications*, pages 226–239, Vienna, Austria, August 1998.

[175] P. Buneman and A. Deutsch and W. Fan and H. Liefke and A. Sahuguet and W.C. Tan. Beyond XML Query Languages. In *Query Language Workshop (QL'98)*, Nov 1998.

[176] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proc Natl Acad Sci U S A*, 85:2444–8, 1988.

[177] P.A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, Cambridge, MA, 2000.

[178] T. Printezis. *Management of Long-Running High-Performance Persistent Object Stores*. PhD thesis, Department of Computing Science, University of Glasgow, 2000.

[179] T. Printezis and M. P. Atkinson. An Efficient Promotion Algorithm for Persistent Object Systems, 2000. Submitted to *Software – Practice and Experience*.

[180] T. Printezis, M. P. Atkinson, L. Daynès, S. Spence, and P. Bailey. The Design of a new Persistent Object Store for PJama. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, August 1997.

[181] K.D. Pruitt and D.R. Maglott. RefSeq and LocusLink: NCBI gene-centered resources. *Nucleic Acids Res*, 29:137–140, 2001.

[182] M. R. Pullan, M. F. Watson, J.B. Kennedy, C. Raguenaud, and R. Hyman. The Prometheus Taxonomic Model: a practical approach to representing multiple taxonomies. *Taxon*, 49:55–75, 2000.

[183] E. Pustułka-Hunt and D. Jack. Case study: Use of computer tools in locating a human disease gene. Technical report, University of Glasgow, Department of Computing Science, March 1999. TR-1999-28, http://www.dcs.gla.ac.uk/∼ela.

[184] E. Pustułka-Hunt, D. Jack, G. F. Hogg, and D. G. Monckton. Case study: CGT repeat expansion modeling using a Java applet and its PJama extension providing persistent storage for genetics data. Technical report, University of Glasgow, Department of Computing Science, April 1999. TR-1999-28, http://www.dcs.gla.ac.uk/∼ela.

[185] S. Quicke. Gene machine. *Computer Weekly*, 2001. 11th January 2001.

[186] P. Riley. Suffix tree optimisation. Technical report, Department of Computing Science, University of Glasgow, April 2001. Department of Computing Science, University of Glasgow.

[187] E. Rocke. Using Suffix Trees for Gapped Motif Discovery. In R. Giancarlo and D. Sankoff, editors, *CPM00*, LNCS 1848, pages 335–349, Montréal, Canada, 2000. Springer-Verlag, Berlin.

[188] P. Ross-Macdonald et al. Large-scale analysis of the yeast genome by transposon tagging and gene disruption. *Nature*, 402:413–418, 1999.

[189] G. M. Rubin. Around the Genomes. The Drosophila Genome Project. *Genome Research*, pages 71–79, 1996. http://www.fruitfly.org/publications/ Around_the_Genomes.-html.

[190] K. Rutherford, J. Parkhill, J. Crook, T. Horsnell, P. Rice, M-A. Rajandream, and B. Barrell. Artemis: sequence visualisation and annotation. *Bioinformatics*, 16:944–945, 2000.

[191] .R Sachidanandam et al. A map of human genome sequence variation containing 1.42 million single nucleotide polymorphisms. *Nature*, 409(6822):928–33, 2001.

[192] Sahuguet, A. and Azavant, F. Looking at the Web through XML glasses. In *CoopIs'99*, 1999.

[193] Sahuguet, A. and Azavant, F. Web Ecology: Recycling HTML pages as XML documents using W4F. In *WebDB'99*, 1999.

[194] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, September 1977.

[195] G.D. Schuler. Sequence Mapping by Electronic PCR. *Genome Res*, 7(5):541–550, 1997.

[196] G.D. Schuler. Electronic PCR: bridging the gap between genome mapping and genome sequencing. *Trends Biotechnol*, 16(11):456–9, 1998.

[197] G.D. Schuler et al. A Gene map of the Human Genome. *Science*, 274:540–558, 1996.

[198] J. Schultz, R.R. Copley, T. Doerks, C.P. Ponting, and P. Bork. SMART: a web-based tool for the study of genetically mobile domains. *Nucleic Acids Res.*, 28(1):231–4, 2000. http://smart.embl-heidelberg.de/.

[199] S. Schwartz, Z. Zhang, K.A. Frazer, A. Smit, C. Riemer, J. Bouck, R. Gibbs, R. Hardison, and W. Miller. PipMaker–a web server for aligning two genomic DNA sequences. *Genome Res.*, 10(4):577–86, 2000.

[200] M. Senger. AppLab, CORBA-Java based Application Wrapper. http://-industry.ebi.ac.uk/applab/.

[201] P. Seshadri. PREDATOR: A resource for database research. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(1):16–20, 1998.

[202] T. Shibuya. Generalization of a Suffix Tree for RNA Structural Pattern Matching. In *Algorithm theory — SWAT 2000: 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5–7, 2000: proceedings*, volume 1851, pages 393–406. Springer-Verlag Inc., 2000.

[203] T.A. Smith and M.S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 284, 1981.

[204] C. Soderlund, S. Humphrey, A. Dunhum, and L. French. Contigs built with fingerprints, markers and FPC V4.7. *Genome Research*, 10:1772–1787, 2000. http://www.sanger.ac.uk/Software/fpc/.

[205] E. L. L Sonnhammer and R. Durbin. A workbench for large scale sequence homology analysis. *Comput. Applic. Biosci.*, 10:301–307, 1994.

[206] N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *Proceedings of the International Conference on Supercomputing (ICS-98)*, pages 141–148, New York, July 13–17 1998. ACM press.

[207] J. Srinivasan et al. Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i. In *ICDE 2000*, pages 91–100, 2000.

[208] R. Stevens and C. Miller. Wrapping and Interoperating Bioinformatics Resources Using CORBA. *Briefings in Bioinformatics*, 2000. To appear.

[209] E.A Stewart et al. An STS-based radiation hybrid map of the human genome. *Genome Res.*, pages 422–433, 1997.

[210] N.E. Stone, Fan J.B, N. Willour, L.A. Pennacchio, J.A. Warrington, A. Hu, A. de la Chapelle, A.E. Lehesjoki, D.R. Cox, and R.M. Myers. Construction of a 750-kb bacterial clone contig and restriction map in the region of human chromosome 21 containing the progressive myoclonus epilepsy gene. *Genome Res.*, pages 218–225, 1997.

[211] S. Sundara, Y. Hu, T. Chorma, and J. Srinivasan. Developing an Indexing Scheme for XML Document Collections using the Oracle8i Extensibility Framework. In *VLDB'01*, pages 701–702. Morgan and Kaufmann, 2001.

[212] W. Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEETIT: IEEE Transactions on Information Theory*, 39, 1993.

[213] I. Szulzewsky, E. Hunt, M. Nguyen, B. Korn, B. Roehrdanz, H. Lehrach, and M-L. Yaspo. An integrated transcript map for the whole human chromosome 21, 1997.

[214] A. Tanner. Genetic Map Browser. Master's thesis, Department of Computing Science, University of Glasgow, 1999.

[215] J. Tarhio and E. Ukkonen. Boyer-Moore approach to approximate string matching. In J. R. Gilbert and R. Karlsson, editors, *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*, LNCS 447, pages 348–359, 1990.

[216] R.L. Tatusov, D.A. Natale, I.V. Garkavtsev, T.A. Tatusova, U.T. Shankavaram, B.S. Rao, B. Kiryutin, M.Y. Galperin, N.D. Fedorova, and E.V. Koonin. The cog database: new developments in phylogenetic classification of proteins from complete genomes. *Nucleic Acids Res*, pages 22–28, 2001. http://www.ncbi.nlm.nih.gov/COG/.

[217] T.A. Tatusova and T.L. Madden. BLAST 2 Sequences, a new tool for comparing protein and nucleotide sequences. *FEMS Microbiol Lett.*, 174(2):247–50, 1999.

[218] K. Thompson. Regular expression search algorithm. *Comm. ACM*, 11:419–422, 1968.

[219] W.F. Tichy. Should computer scientists experiment more? 16 reasons to avoid experimentation. *IEEE Computer*, 31(5):32–40, 1998.

[220] W.F. Tichy, P. Lukowicz, L. Precht, and E.A. Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software*, 28(1):9–18, 1995.

[221] T. Troup. Integrated Map Display Applet. Master's thesis, Department of Computing Science, University of Glasgow, 1999.

[222] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–212, 1992.

[223] E. Ukkonen. Approximate string matching over suffix trees. *CPM93*, 684:228–242, 1993.

[224] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, 1995. TR A-1993-1, Department of Computing Science, University of Helsinki, Finland.

[225] A. Vanet, L. Marsan, A. Labigne, and M-F. Sagot. Inferring Regulatory Elements from a Whole genome. An Analysis of *Heliobacter pylori* σ^{80} Family of Promoter Signals. *J. Mol. Biol.*, 297:335–353, 2000.

[226] J. C. Venter et al. The Sequence of the Human Genome. *Science*, 291:1304–1351, 2001.

[227] J. Viksna and D. Gilbert. Pattern matching and pattern discovery algorithms for protein topologies. In *Algorithms in Bioinformatics: First International Workshop, WABI 2001 Proceedings*, LNCS 2149, pages 98–111, 2001.

[228] L. Wall, R.L. Schwartz, T. Christiansen, and S. Potter. *Programming Perl*. Nutshell Handbook. O'Reilly & Associates, 2nd edition, 1996.

[229] B.C. Warboys, P. Kawalek, I. Robertson, and R.M. Greenwood. *Business Information Systems: a Process Approach*. McGraw-Hill, 1999.

[230] M. Waterman. *Introduction to Computational Biology. Maps, sequences and genomes*. Chapman & Hall, 1995.

[231] M.S. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA–rRNA comparisons. *Journal of Molecular Biology*, 197:723–728, 1987.

[232] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

[233] S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner. *JDBC(TM) API Tutorial and Reference, Second Edition: Universal Data Access for the Java(TM) 2 Platform (Java Series)* . Addison-Wesley, 1999.

[234] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[235] M-L. Yaspo, L. Gellen, R. Mott, B. Korn, D. Nizetic, A-M. Poustka, and H. Lehrach. Model for a transcript map of human chromosome 21 - isolation of new coding sequences from exon and enriched cdna libraries. *Hum. Mol. Genet.*, pages 1291–1304, 1995.

[236] M-L. Yaspo, I. Szulzewsky, E. Hunt, M. Nguyen, X. Kong, J. O'Brian, and H. Lehrach. The Chromosome 21 transcript map - How far from completion with more than 3000 potential coding sequences. Impact for the Molecular Genetics of Down's syndrome. 1997. Poster at the Genome Mapping and Sequencing Conference, Cold Spring Harbour, New York, 1997.

[237] B.E. Young. Suffix Binary Search Trees in Java. Master's thesis, Department of Computing Science, University of Glasgow, 2000.

[238] J. Zhang and T.L. Madden. PowerBLAST: a new network BLAST application for interactive or automated sequence analysis and annotation. *Genome Res*, 7(6):649–56, 1997.

[239] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A Greedy Algorithm for Aligning DNA Sequences. *Journal of Computational Biology*, 7:203–214, 2000.